第十六期：Android 底层驱动

eoe特刊

目录

【Android 系统架构及其驱动研究】

1.1 Android 系统架构

Android 是为移动设备设计的软件平台，包括操作系统、中间件和一些关键应用。AndroidSDK 提供了必须的工具和进行应用开发所必须的 Java 接口 API。

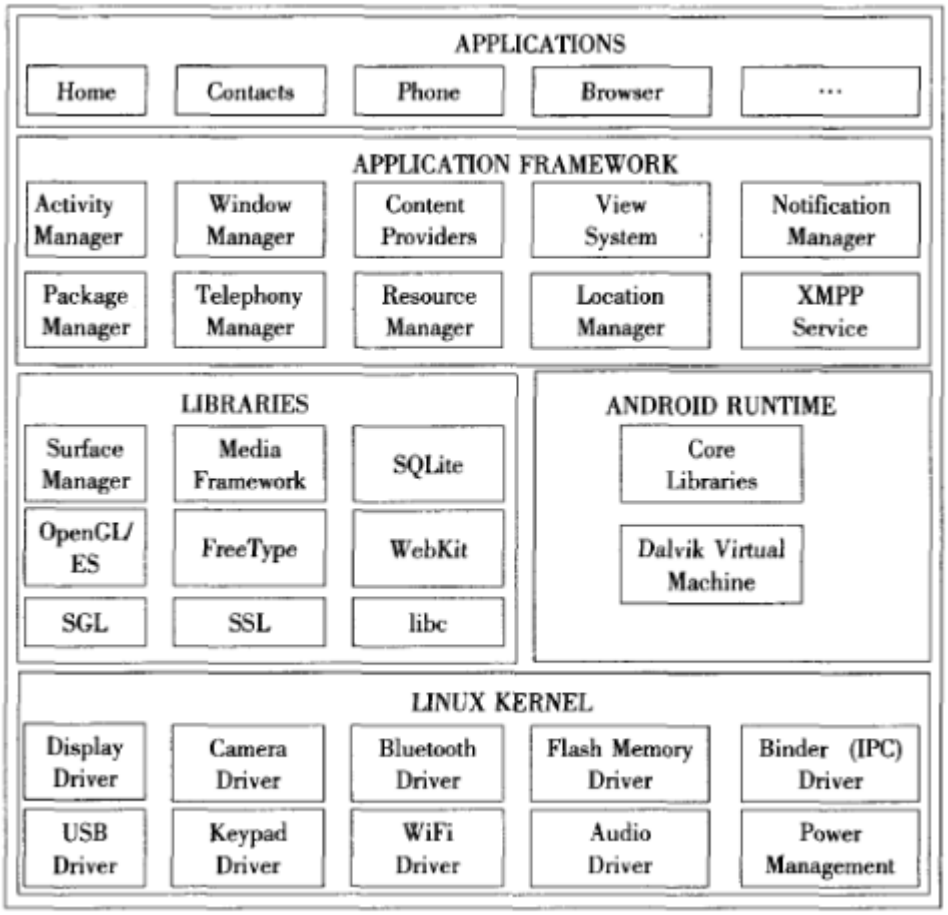

图 1.1 Android 框架图

Android 是一个开放的软件系统，为用户提供了丰富的移动设备开发功能，从下至上包括 4 个层次，如图 1 所示。其中第一层是 Linux 内核层，包括 Linux 操作系统及驱动，依赖于 Linux2.6 内核，不支持 linux2.4 内核。如 Android1.0(release 一 1.0)使用 inux2.6.25，Android1.6(sdk 一 1.6)使用 Linux2.6.29。除了标准的 Linux 内核外，Android 系统还增加了 Binder IPC 驱动、WiFi 驱动、蓝牙驱动等驱动程序，为系统运行提供了基础性支持。

第二层是核心的扩展类库，如 SQLite、WebKit、OpenGL 等，它们可以通过 JAVA 本地调用 JNI(Java Native Interface)的接口函数实现和上层之间的通信。该层由 Android 的 Java 虚拟机 Dalvik 和基础的 Java 库为 Java 运行环境提供了 Java 编程语言核心库的大多数功能。

第三层是包含所有开发所用的 SDK 类库和某些未公开接口类库的框架层，是整个 Android 平台核心机制的体现。

第四层是应用层。系统部分应用和第三方开发的应用都是位于这个层次上，但两者不完全相同，其中系统应用会用一些隐藏的类，而第三方的应用，是基于 SDK 基础上开发的。一般 Android 开发是在 SDK 基础上用 Java 编写应用程序，但本机开发程序包 NDK 提供了应用层穿越 Java 框架层直接和底层包含了 JNI 接口的 C / C++库直接通信的方法。

其中第一层由 C 语言实现，第二层由 C 和 c+十实现，第三、四层主要由 Java 代码实现。从 Linux 操作系统的角度来看，第一、二层次之间是内核空间与用户空间的分界线，第一层运行于内核空间，第二、三、四层运行于用户空间。第二、三层之间，是本地代码层和 Java 代码层的接口。第三、四层之间是系统 API 接口。

1.2 Android 代码结构

Android 代码包括 3 个部分：

① 核心工程(Core Project 文件夹)是建立 Android 系统的基础，在根目录的各个文件夹中；

② 扩展工程(External Project 文件夹中)是使用其他开源项目扩展功能；

③ 包(Package)提供 Android 的应用程序和服务。其中既包含了原始 Android 的目标机代码，还包括了主机编译工具、仿真环境等。

代码包经过解压缩后，第一级别的目录和文件如下所示：

```
Mydroid/
—— Makefile (全局的 Makefile)
—— bionic (这里面是一些基础的库的源代码)
—— bootloader (引导加载器)
—— build (目录的内容不是目标所用的代码，而是编译和配置所需要的脚本和工具)
—— dalvik (JAVA 虚拟机)
—— development (程序开发所需要的模板和工具)
—— external (目标机器使用的一些库)
—— frameworks (应用程序的框架层)
—— hardware (与硬件相关的库)
—— kernel (Linux2. 6 的源代码)
—— packages (Android 的各种应用程序)
—— prebuilt (Android 在各种平台下编译的预置脚本)
—— recovery (与目标的恢复功能相关)
—— system (Android 的底层的一些库)
```

编译完成后，将在根目录中生成一个 out 文件夹，所有生成的 Android 代码结构内容均放置在这个文件夹中。out 文件夹如下所示：

```
Out/
—— CaseCheck. txt
—— casecheck. txt
—— host
—— common
```

—— linux—x86
—— taeget
—— common
—— product

主要的两个目录为 host 和 target，前者表示在主机(x86)生成的工具，后者表示目标机(默认为 ARMy5)运行的内容。

1.3 Android 专用驱动

Android 中内核的结构和标准 Linux2. 6 内核基本相同，但增加了部分私有内容。Android 在标准的 Linux 内核中的驱动主要分成两种类型：**Android 专用驱动**和 **Android 使用的设备驱动**。其中主要的专用驱动包括：

Ashmen：匿名共享内存驱动；
Logger：轻量级的 log 驱动；
Binder：基于 OpenBinder 系统的驱动，为 Android 平台提供 IPC 支持；
Android Power Management(PM)电源管理模块；
Low Memory Killer：在缺少内存的情况下，杀死进程；
Android PMEM：物理内存驱动。

1.3.1 Ashmen

即匿名共享内存(Anonymous Shared Memo—ry)，为用户空间程序提供分配内存的机制，实现类似 malloc 的功能。其设备节点名称为：/dev/ashmen. 主设备号为 10，次设备号动态生成。其驱动程序在内核中的头文件和代码路径如下：

Kernel/include/linux/ashmen.H
Kernel/mm/ashmen.c

在用户空间 C libutil 库对 Ashmen 进行封装并提供接口；

System/core/include/eutils/ashmen.H —— 简单封装头文件；
System/core/libcutils/ashmen—dev.c —— 匿名共享内存在用户空间的调用封装；
System/core/libcutils/ashmen—host.c —— 没有使用。

用于为 Android 系统提供内存分配功能。

1.3.2 Binder

为用户层程序提供进程间通信(IPC)支持，Android 整个系统的运行依赖 Binder 驱动。其设备节点名称为：/dev/binder，主设备号为 10，次设备号动态生成。Binder 驱动程序在内核中的头文件和代码路径如下：

Kernel/include/linux/binder.h
Kernel/drivers/misc/binder.c

在用户空间 libutil 工具库和 Service Manager 守护进程中调用 Binder 接口提供对整个系统的支持。

Frameworks/base/cmds/servicemanager/ —— Server Manager 守护进程的实现；

Framewoks/base/include/utils/ —— Binder 驱动在用户空间的封装接 1：7；

Frameworks/base/libs/utils/ —— Binder 驱动在用户空间的封装实现。

Binder 是 Android 中主要使用的 IPC 方式，使用时通常只需要按照模板定义相关的类即可，不需要直接调用 Binder 驱动程序的设备节点。

1.3.3 Logger

该驱动程序为用户层程序提供 log 支持，作为一个工具使用。在用户空间中有 3 个设备节点：/dev/log/main、/dev/log/event、/dev/log/radio，主设备号为 10，次设备号动态生成。Logger 驱动在内核中的头文件和代码路径如下：

Kern el/include/linux/logger.h

Kernel/driver/mise/logger.C

在 Android 的用户空间 logcat 程序调用 Logger 驱动：

SysteH cOre / logcat / ，— — 可执行程序。

Logcat 是一个可执行程序，用于提取系统 log 信息，是系统的一个辅助工具。

1.4 Linux 设备驱动在 Android 上的使用分析

Linux 设备驱动程序，在为智能手机定制的 Android 操作系统中得到了广泛的应用，下面对几个比较有特色的设备驱动的使用情况进行分析。

1.4.1 Framebuffer 显示驱动

Framebuffer 驱动在 Linux 中是标准的显示设备的驱动。对于 PC 系统，它是显卡的驱动；对于嵌入式 SOC 处理器系统，它是 LCD 控制器或者其他显示控制器的驱动。它是一个字符设备，在文件系统中设备节点通常是/dev/fbx。每个系统可以有多个显示设备，依次用/dev/fb0、/dev/fb1 等来表示。在 Android 系统中主设备号为 29，次设备号递增生成。

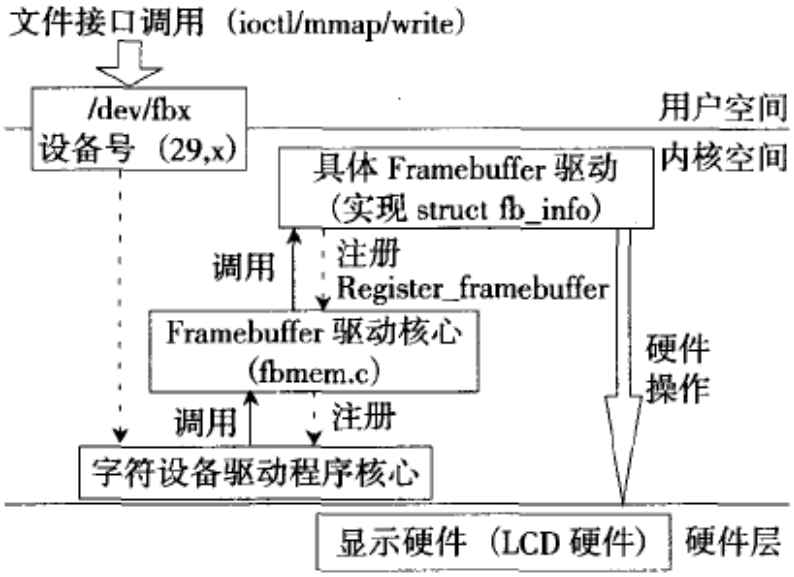

图 1.2 Framebuffer 显示驱动架构图

该驱动在用户空间一般使用 ioctl、mmap 等文件系统接口进行操作，ioctl 用于获得和设置

信息，mmap 将 Framebuffer 的内存映射到用户空间。驱动也可以直接支持 write 操作，用写的方式输出显示内容。显示驱动的架构如图 1.2 所示。

驱动的主要头文件位于 include/linux/fb.h 文件中；驱动核心实现位于 drivers/video/fb.mem. c 文件中，驱动中核心的数据接口是 fb—info 在 fb. h 中定义，如下所示。

```
Struct fb—info{
Int node;
Int flags;
Struct fb_var_screeninfo var;  /*变化屏幕信息*/
Struet fb_fix_screeninfo fix;  /*固定屏幕信息*/
Struct fb_ops fbops;  /*Framebuffer 驱动的操作$ /
......
}
```

该结构包含了驱动主要信息，struct fb_var_screeninfo 和 Street fb_fix_screeninfo 两个数据结构对应 FBIOGET_VSCREENINFO 和 FBIO_GET_FSCREENINFO 这两个 ioctl 从用户空间获得的显示信息。Fb—ops 表示 Framebuffer 驱动的操作，通常通过以下函数进行注册：

```
Int register_framebuffer(struct fb_info * fb_info);
```

具体的 Framebufer 驱动需要实现 fb_info 结构，实现 fb_ops 中的各个函数指针。从驱动程序的用户空间进行 ioctl 调用时，会转换成调用其中的函数。

Android 对 Framebuffer 驱动的使用方式是标准的，在/dev/graphie/中的 Framebuffer 设备节点由 init 进程自动创建，被 libui 库调用。Android 的 GUI 系统中，通过调用 Framebuffer 驱动的标准接口，实现显示设备的抽象。

1.4.2 Event 输入设备驱动

Input 驱动程序是 Linux 输入设备的驱动程序，分为游戏杆(joystick)、鼠标(mouse 和 mice)和事件设备(Event queue)3 种驱动程序。其中事件驱动程序是目前通用的程序，可支持键盘、鼠标、触摸屏等多种输入设备。Input 驱动程序的主设备号是 13，每一种 Input 设备从设备号占用 5 位，3 种从设备号分配是：游戏杆 0～61；Mouse 鼠标 33～62；Mice 鼠标 63；事件设备 64～95，各个具体的设备在 misc、touchscreen、keyboard 等目录中。

Event 设备在用户空间使用 read、ioctl、poll 等文件系统的接口操作，read 用于读取输入信息，ioctl 用于获取和设置信息，poll 用于用户空间的阻塞，当内核有按键等中断时，通过在中断中唤醒内核的 poll 实现。Event 输入驱动的架构如图 1.3 所示。

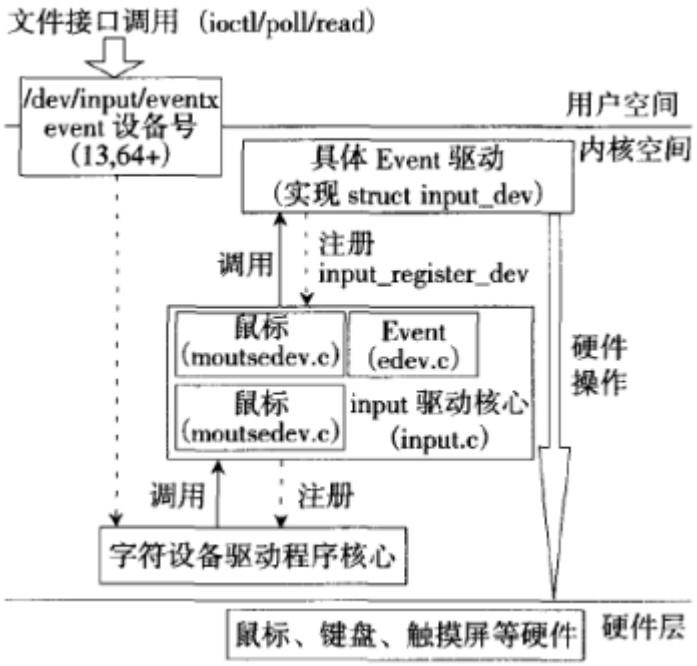

图 1.3 Event 输入驱动架构图

Input 驱动程序的头文件在 include/linux/input.h 中。Input 驱动程序的核心和 Event 部分代码，分别位于 Drivers/input/input.C 和 Drivers/input/evdev.C 中。其中 Input.h 中定义了 structinput_dev 结构，它表示 Input 驱动程序的各种信息，对于 Event 设备分为同步设备、键盘相对设备(鼠标)、绝对设备(触摸屏)等。Event 驱动程序需要定义 struct input_dev 结构体，并且通过 input_register_device()函数进行注册。

```
Int_must_chect input_register_device(struct input_dev * );
```

Input 设备驱动在内核 menuconfig 配置时，配置选项为“Device Drivers”—>“Input Device Drivers”。Event 驱动程序配置对应的文件是 driver/input/Kconfig，其配置选项是“Event Interface”，各个具体设备的接口在各自下面进行支持。

Android 使用 Event 驱动作为标准的输入设备，在 GUI 系统中打开 Event 驱动程序的设备节点，通常的输入设备是鼠标和触摸屏。Android 的 init 进程在/dev/input 自动建立 Event 设备的节点，被 libui 库调用作为系统的输入。

1.4.2 ALSA 音频驱动

高级 Linux 声音体系 ALSA(Advanced Linux Sound Architecture)是为音频系统提供驱动的 Linux 内核组件，以替代原先的开发声音系统 OSS。它是一个完全开放源代码的音频驱动程序集，除了像 OSS 那样提供一组内核驱动程序模块之外，ALSA 还专门为简化应用程序的编写提供相应的函数库，与 OSS 提供的基于 ioctl 等原始编程接口相比，ALSA 函数库使用起来要更加方便一些。

利用该函数库，开发人员可以方便、快捷地开发出自己的应用程序，细节则留给函数库进行内部处理。所以虽然 ALSA 也提供了类似于 OSS 的系统接口，但建议应用程序开发者使用音频函数库，而不是直接调用驱动函数。

ALSA 驱动的主设备号为 116，次设备号由各个设备单独定义，主要的设备节点如下：

/dev/snd/contmlCX —— 主控制；
/dev/snd/pcmXXXc —— PCM 数据通道；
/dev/snd/seq —— 顺序器；
/dev/snd/timer —— 定义器。

在用户空间中，ALSA 驱动通常配合 ALsA 库使用，库通过 ioctl 等接口调用 ALSA 驱动程序的设备节点。对于 AIJSA 驱动的调用，调用的是用户空间的 ALsA 库的接口，而不是直接调用 ALSA 驱动程序。ALSA 音频驱动的架构如图 1.4 所示。

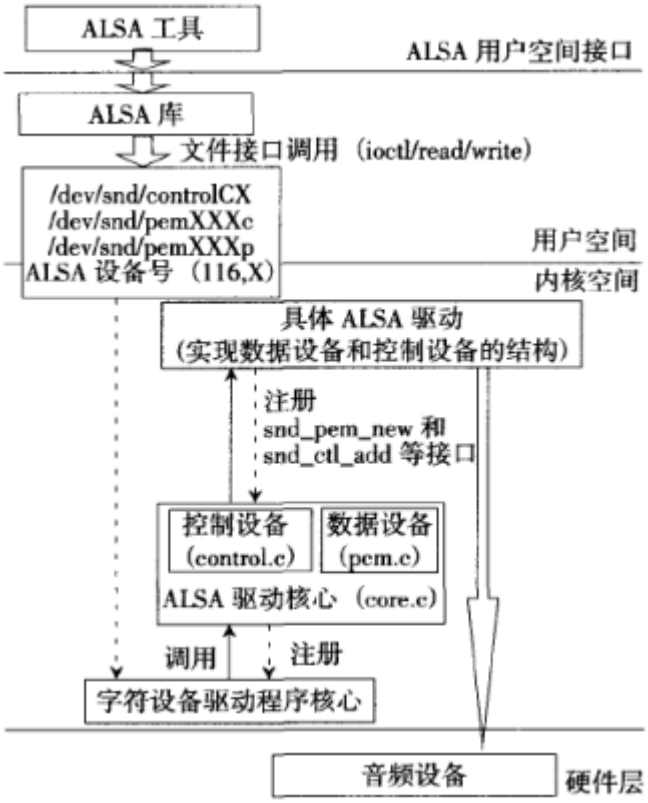

图 1.4　ALSA 音频驱动架构图

ALSA驱动程序的主要头文件是 include/sound./sound.h，驱动核心数据结构和具体驱动的注册函数是 include/sound/core.h，驱动程序的核心实现是Sound/core/sound.c文件。

ALSA 驱动程序使用下面的函数注册控制和设备。

```
int snd_pcm_new (struct snd_card * card, char * id, int device, int playback_count,
int capture_count, struct snd_ pcm ** rpcm);

int snd ctl_add(struct snd_card * card, struct snd_kcontrol * kcontrol);
```

ALSA 音频驱动在内核进行 menuconfig 配置时，配置选项为“Device Drivers”>“Sound card support”—>“Advanced Linux Sound Architecture”。子选项包含了 Generic sound devices(通用声音设备)、ARM 体系结构支持，以及兼容 OSS 的几个选项。ALsA 音频驱动配置对应的文件是

sound/core/Kconfig。

Android 没有直接使用 ALSA 驱动，可以基于 A-LSA 驱动和 ALSA 库实现 Android Audio 的硬件抽象层；ALSA 库调用内核的 ALSA 驱动，Audio 的硬件抽象层调用 ALSA 库。

1.5 Android 比起 Linux 的七点优势。

1、Alarm ： 是 Android 系统中在标准 RTC 驱动上开发的一个新的驱动，提供了一个定时器，用于把设备从睡眠状态唤醒。
2、Ashmem/pmem ： 使得进程间能够共享大块的内存（如图标）。它为内核提供了一种回收这些使用完的共享内存块的办法。
3、Binder ： 解决了标准 Linux 进程间通信会花费许多开销，并有安全漏洞的问题。
4、Power management ： 提供更多的电源管理策略；使用唤醒锁来管理电源。
5、Low memory killer ： 当内存不够的时候，该策略会试图结束一个进程。
6、Kernel debugger
7、Logger ： 使得调试信息可以输入到一个内存块中。

【Android 底层驱动概述】

2.1 Android 底层驱动的详细内容。（eoe 社区用户笔名：无心）

Android 底层的驱动其实就是 linux 驱动，所以在这里就给大家描述一下 linux 的驱动原理。在看本节之前，读者应该要对 linux 系统有一定的了解，否则看起来比较吃力。

顾名思义，驱动程序是用来控制计算机外围设备的，Linux 系统将所有的外围设备都高度地抽象成一些字节的序列，并且以文件的形式来表示这些设备，稍微解释一下，linux 和我们常用的 windows 在文件上面不一样，linux 系统上面所有的东西都是以文件的形式存在，比如文本文件，目录文件。而 windows 则是以多种文件格式存在，比如 exe 文件，txt 文件等等。我们可以来看一下 Linux 的 I/O 子系统（图 2.1）。

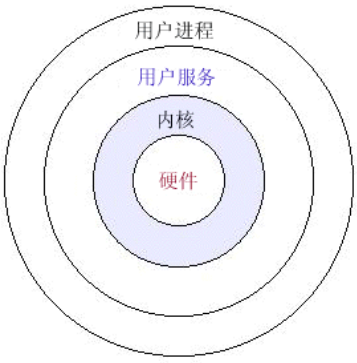

图 2.1 Linux 的 I/O 子系统

由图 2.1 可知，内核是所有程序离硬件最近的一个，所以它可以直接与硬件交互。内核紧紧地包围在硬件周围，内核是一些软件包的组合，它们可以直接访问系统的硬件，包括处理器、内存和 I/O 设备。而用户进程则通过内核提供的用户服务来和内核通讯，从而间接地控制系统硬件。

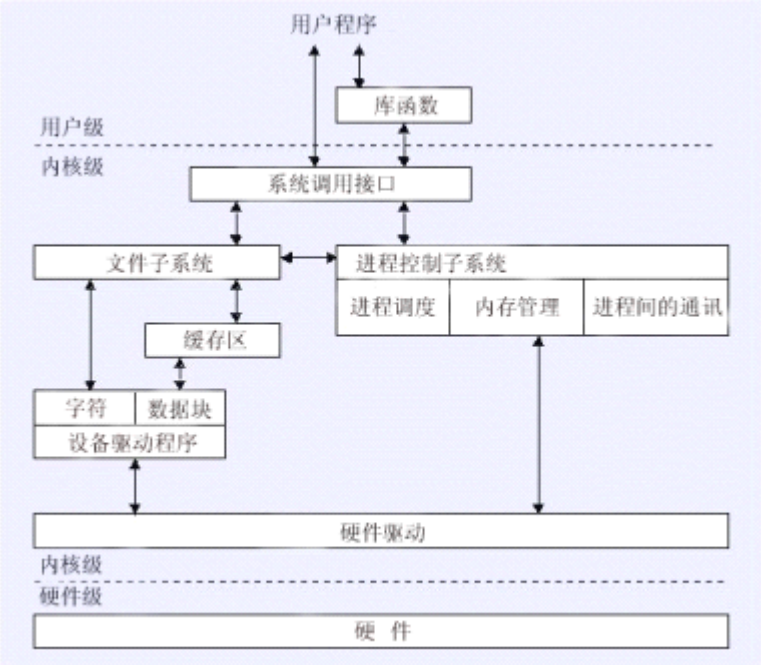

图 2.2 内核提供的标准系统调用

图 2.2 显示了用户级的程序,使用内核提供的标准系统调用,来与内核通讯，一般的系统调用有：open()，read()，write()，ioctl()，close() 等等。

Linux 的内核是一个有机的整体。每一个用户进程运行时都好像有一份内核的拷贝，每当用户进程使用系统调用时，都自动地将运行模式从用户级转为内核级，此时进程在内核的地址空间中运行。

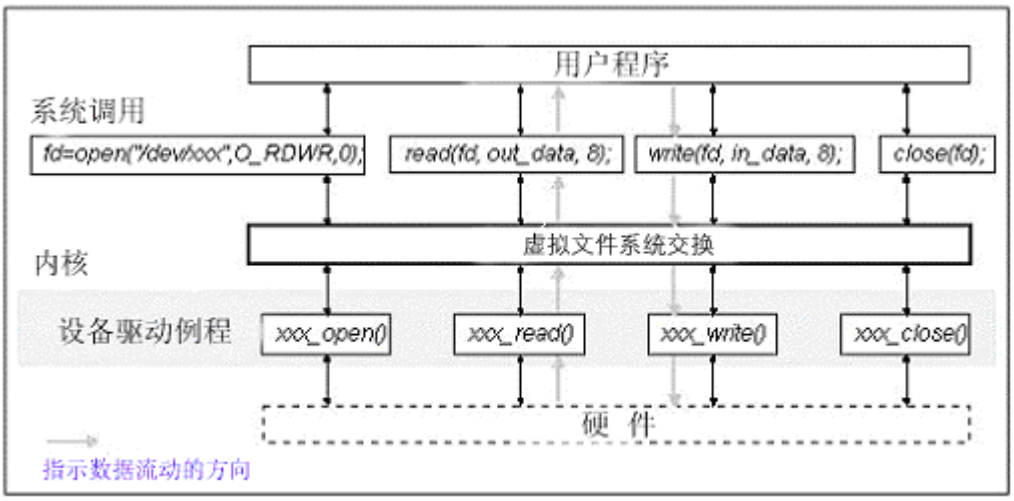

图 2.3 基本系统调用的实例

Linux 内核使用"设备无关"的 I/O 子系统来为所有的设备服务。每个设备都提供标准接口给内核，从而尽可能地隐藏了自己的特性。图 2.3 展示了用户程序使用一些基本的系统调用从设备读取数据并且将它们存入缓冲的例子。我们可以看到，每当一个系统调用被使用时，内核就转到相应的设备驱动例程来操纵硬件。

每个设备在 Linux 系统上看起来都像一个文件，它们存放在/dev 目录中并被称为"特殊文件"或是"设备节点"。大家可以使用 ls -l /dev/lp* 来得到以下的输出：

```
crw-rw-rw 1 root root 6, 0 April 23 1994 /dev/lp0
```

这行输出表示 lp0 是一个字符设备（属性字段的第一个字符是'c'），主设备号是 6，次设备号是 0。主设备号用来向内核表明这一设备节点所代表的驱动程序的类型（比如：主设备号是 3 的块设备是 IDE 磁盘驱动程序，而主设备号为 8 的块设备是 SCSI 磁盘驱动程序）。

每个驱动程序负责管理它所驱动的几个硬件实例，这些硬件实例则由次设备号来表示（例如:次设备号为 0 的 SCSI 磁盘代表整个也可以说是"第一个"SCSI 磁盘，而次设备号为 1 到 15 的磁盘代表此 SCSI 磁盘上的 15 个分区）。

到此eoer 们应该对Linux 的设备有所了解了吧,下面就可以开始我们的正题"设备驱动程序"。

设备驱动程序是一组由内核中的相关子例程和数据组成的 I/O 设备软件接口。每当内核意识到要对某个设备进行特殊的操作时，它就调用相应的驱动例程。这就使得控制从用户进程转移到了驱动例程，当驱动例程完成后，控制又被返回至用户进程。图 2.4 就显示了以上的过程。

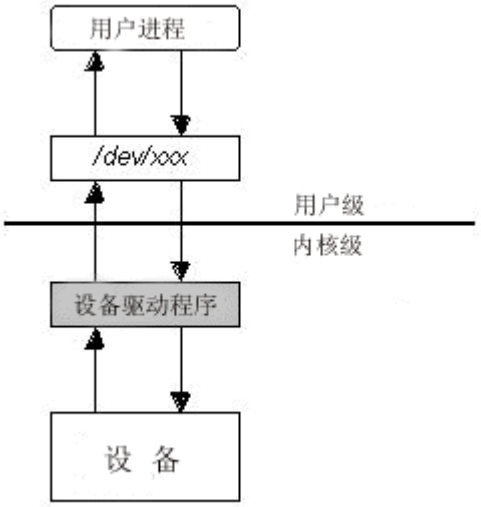

图 2.4　用户进程与驱动例程之间的转移过程

2.1.1　设备驱动程序的作用

驱动程序是一个小型的系统级程序，它能够使特定的硬件和软件与操作系统建立联系，让操作系统能够正常运行并启用该设备。如果您正准备添加某些新的设备，操作系统不会知道如何处理它。但是当您安装了驱动程序后，操作系统就可以正确的判断出它是什么设备，更重要的是：它知道了如何使用这个新设备。

2.1.2　Linux 设备驱动程序完成 4 个功能：

1、对设备初始化和释放
2、把数据从内核传送到硬件和从硬件读取数据
3、读取应用程序传送给设备文件的数据和回送应用程序请求的数据
4、检测和处理设备出现的错误。

2.1.3　设备驱动程序都具有以下 3 个特性：

1、具有一整套的和硬件设备通讯的例程，并且提供给操作系统一套标准的软件接口。
2、具有一个可以被操作系统动态地调用和移除的自包含组件。
3、可以控制和管理用户程序和物理设备之间的数据流。

2.2　字符设备和块设备

接下来我们来了解一下字符设备和块设备，它们是 Linux 系统中两种主要的外围设备。

我们常见的磁盘是块设备，而终端和打印机是字符设备。块设备被用户程序通过系统缓冲来访问。特别是系统内存分配和管理进程就没有必要来充当从外设读写的数据传输者了。

正好与之相反的是，字符设备直接与用户程序进行通讯，而且两者似乎没有缓冲区。Linux 的传输控制机制会根据用户程序的需要来正确地操纵内存和磁盘等外设来取得数据。在 Linux 系

统中字符设备驱动器被保存为/usr/src/linux/drivers/char 目录中。

下面我们重点介绍字符设备驱动程序的开发方法。

首先了解一下 Linux 的内核编程环境。大家都知道，linux 系统是一个多任务多进程的系统，所以每个 Linux 用户进程都在一个独立的系统空间中运行着，与系统区和其他用户进程相隔离。这样就保护了一个用户进程的运行环境，以免被其他用户进程所破坏。

与这种情况正相反的是，设备驱动程序运行在内核模式，它们具有很大的自由度。这些设备驱动程序都是被假设为正确和可靠的，它们是内核的一部分，可以处理系统中断请求和访问外围设备，同时它们有效地处理中断请求以便系统调度程序保持系统需求的平衡。所以设备驱动程序可以脱离系统的限制来使用系统区，比如系统的缓冲区等等。

一个设备驱动程序同时包括中断和同步区域。其中中断区域处理实时事件并且被设备的中断所驱动；而同步区域则组成了设备的剩余部分，处理进程的同步事件。所以，当一个设备需要一些软件服务时，就发出一个"中断"，然后中断处理器得到产生中断的原因同时进行相应的动作。

一个 Linux 进程可能会在事件发生之前一直等待下去。例如，一个进程可能会在运行中等待一些写入硬件设备的信息的到来。其中一种方式是进程可以使用 sleep()和 wakeup()这两个系统调用，进程先使自己处于睡眠状态，等待事件的到来，一旦事件发生，进程即可被唤醒。

举个例子来说:interruptible_sleep_on(&dev_wait_queue)函数使进程睡眠并且将此进程的进程号加到进程睡眠列表 dev_wait_queue 中，一旦设备准备好后，设备发出一个中断，从而导致设备驱动程序中相应的例程被调用，这个驱动程序例程处理完一些设备要求的事宜后会发出一个唤醒进程的信号，通常使用 wake_up_interruptible(&dev_wait_queue)函数，它可以唤醒 dev_wait_queue 所示列表中的所有进程。

特别要注意的是，如果两个和两个以上的进程共享一些公共数据区时，我们必须将之视为临界区，临界区保证了进程间互斥地访问公共数据。在 Linux 系统中我们可以使用 cli()和 sti()两个内核例程来处理这种互斥，当一个进程在访问临界区时可以使用 cli()来关闭中断，离开时则使用 sti()再将中断打开，就像下面的写法：

```
cli()
  临界区
sti()
```

2.3 Linux 下的 VFS

VFS 的作用就是采用标准的 Unix 系统调用读写位于不同物理介质上的不同文件系统。VFS 是一个可以让 open()、read()、write()等系统调用不用关心底层的存储介质和文件系统类型就可以工作的粘合层。在古老的 DOS 操作系统中，要访问本地文件系统之外的文件系统需要使用特殊的工具才能进行。而在 Linux 下，通过 VFS，一个抽象的通用访问接口屏蔽了底层文件系统和物理介质的差异性。

每一种类型的文件系统代码都隐藏了实现的细节。因此，对于 VFS 层和内核的其它部分而言，每一种类型的文件系统看起来都是一样的。

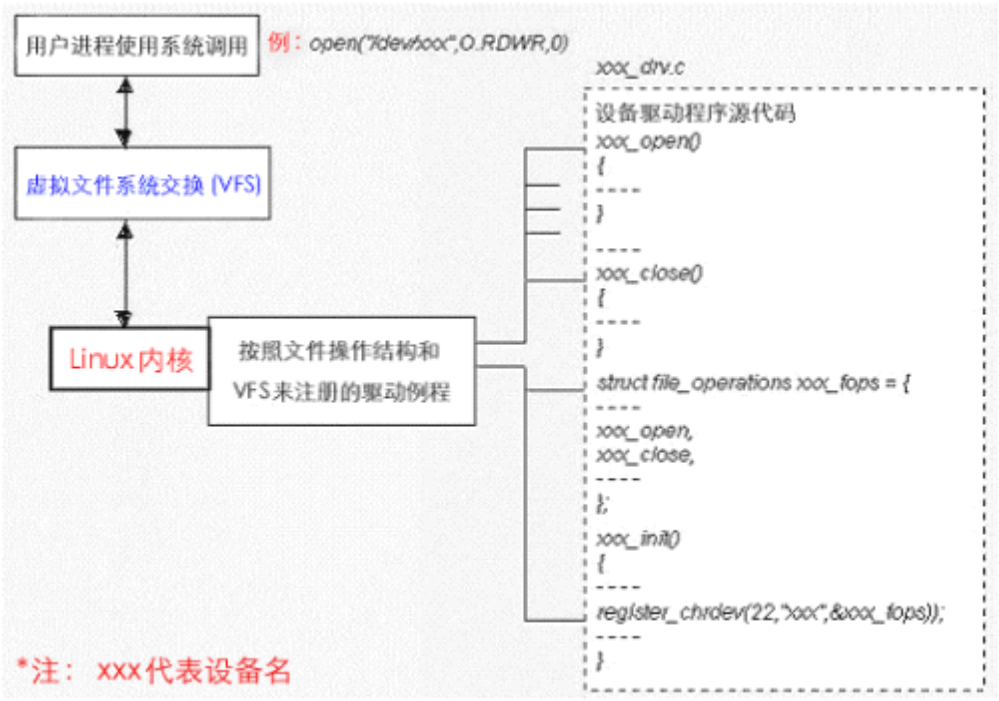

图 2.5 文件操作结构的定义

图 2.5 中的"文件操作结构"在/usr/include/linux/fs.h 文件中定义，此结构包含了驱动程序中的函数列表。图上的初始化例程 xxx_init()根据 VFS 和设备的主设备号来注册"文件操作结构"。

下面是一些设备驱动程序的支撑函数：

add_timer()： 定时间一过，可以引发函数的执行；
cli()： 关闭中断，阻止中断的捕获；
end_request()： 当一个请求被完成或被撤销时被执行；
free_irq()： 释放一个先前被 request_irq()和 irqaction()捕获的的中断请求；
get_fs*()： 允许一个设备驱动程序访问用户区数据（一块不属于内核的内存区）；
inb(), inb_p()： 从一个端口读取一个字节，其中 inb_p() 会一直阻塞直到从端口得到字节为止；
irqaction()： 注册一个中断；
IS_*(inode)： 测试 inode 是否在一个被 mount 了的文件系统上；
kfree*()： 放先前被 kmalloc()分配的内存区；
kmalloc()： 分配大于 4096 个字节的大块内存区；
MAJOR()： 返回设备的主设备号。
MINOR()： 返回设备的次设备号。
memcpy_*fs()： 在用户区和内核区之间复制大块的内存。
outb(), outb_p()： 向一个端口写一个字节，其中 outb_p()一直阻塞直到写字节成功为止。
printk()： 内核使用的 printf()版本；
put_fs*()： 允许设备驱动程序将数据写入用户区；
register_*dev()： 在内核中注册一个设备；

request_irq()：向内核申请一个中断请求 IRQ，如果成功则安装一个中断请求处理器。
select_wait()：将一个进程加到相应 select_wait 队列中；
***sleep_on()：**使进程睡眠以等待事件的到来，并且将 wait_queue 入口点加到列表中以便事件到来时将进程唤醒；
sti()：和 cti()相对应，恢复中断捕获；
sys_get*()：系统调用，得到进程的有关信息；
wake_up*()：唤醒先前被*sleep_on() 睡眠的进程；

【Android 驱动类别】(eoeAndroid 社区 ID：crazy)

3.1 Android 专用驱动 Ashmem、binder、logger

Android 专用驱动分类

1）**Ashmen：** 匿名共享内存驱动，通过内核的机制,为用户空间程序提供分配内存的机制，设备节点/dev/ashmem。

2）**Binder：** 基于 OpenBinder 驱动，为 Android 平台提供 IPC(进程间通信)的支持，Android 整个系统运行依赖 Binder 驱动，对应的文件(binder.c)。

3）**Logger：** 轻量级的 Log 驱动，为用户提供 log 的支持,此驱动作为工作使用节点 /dev/log/main、/dev/log/event、/dev/log/radio，对应的文件(logger.c)。

4）**Android Power Management(能源管理)：** 轻量级的能源管理,基于 Linux 的能源管理。

5）**Low Memory Killer：** 在缺少内存的情况下杀死进程。

6）**Android PMEM：** 物理内存驱动。

7）**电源管理：** Power(power.c)

8）**闹钟管理：** Alarm(alarm.c)

9）**内存控制台：** Ram_console(ram_console.c)

10）**时钟控制台：** Timed_gpio(timed_gpio.c)

3.2 Android 设备驱动

1、framebuffer 驱动

显示驱动使用 framebuffer 驱动节点/dev/fb0 /dev/graphics/b0

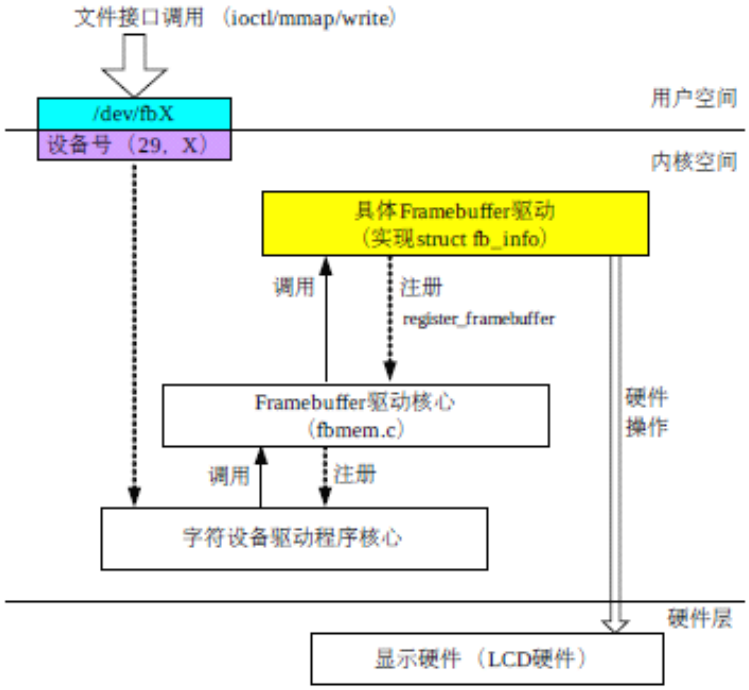

图 3.1 Framebuffer 显示驱动

2、event 输入设备驱动

通常使用 Input 设备中的 Event 设备，节点/dev/input/eventx。

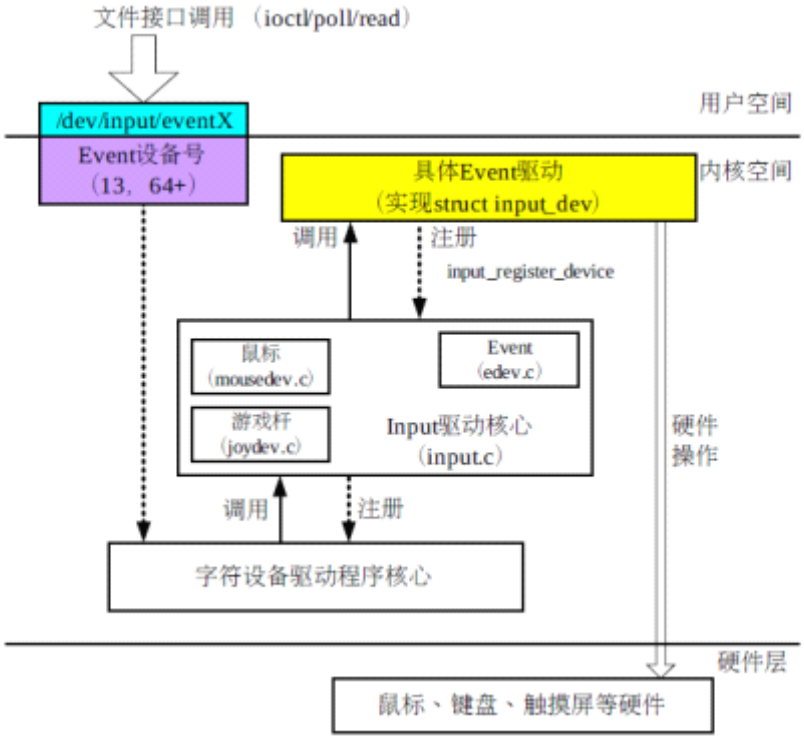

图 3.2 Event 输入设备驱动

3、v412 摄像头：视频驱动

摄像头(Camera)，视频驱动通常使用 Video For Linux 节点/dev/video/videox。

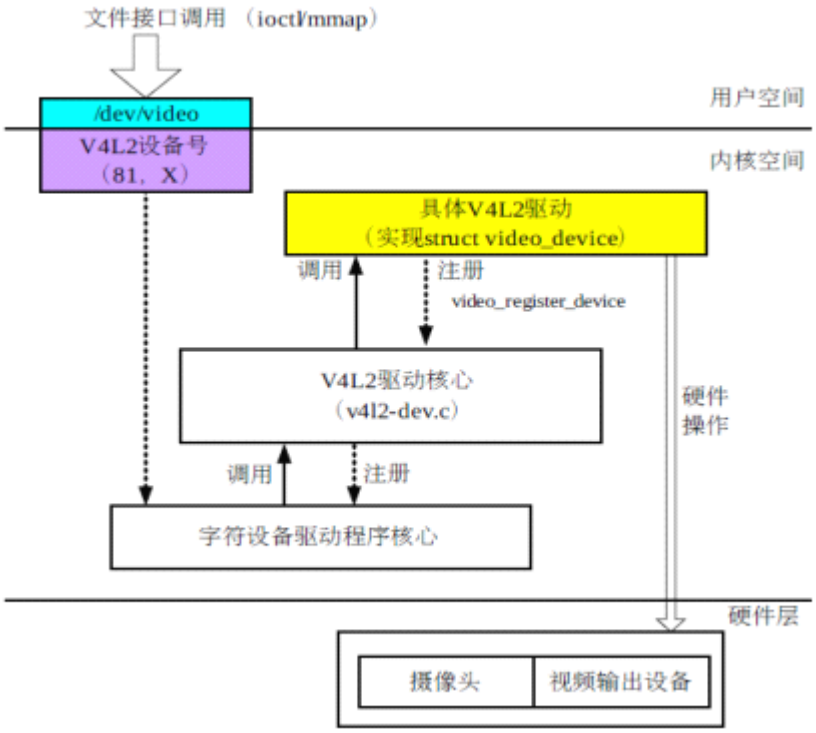

图 3.3 V412 摄像头-视频驱动

4、OSS 音频驱动

(Open Sound System)音频驱动:开放声音系统,节点/dev/mixer;/dev/sndstat;/dev/dsp。

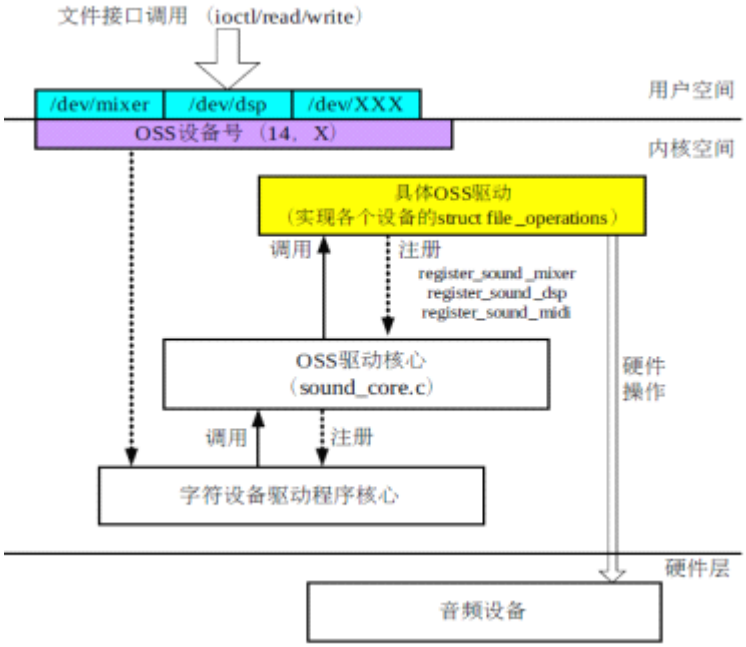

图 3.4 OSS 音频驱动

5、ALSA 音频驱动

ALSA 音频驱动高级 Linux 声音体系，节点/dev/snd/controlCX；/dev/snd/pcmXXXc/dev/snd/pcmXXXp；/dev/snd/pcmXXXp；/dev/snd/seq；/dev/snd/timer。

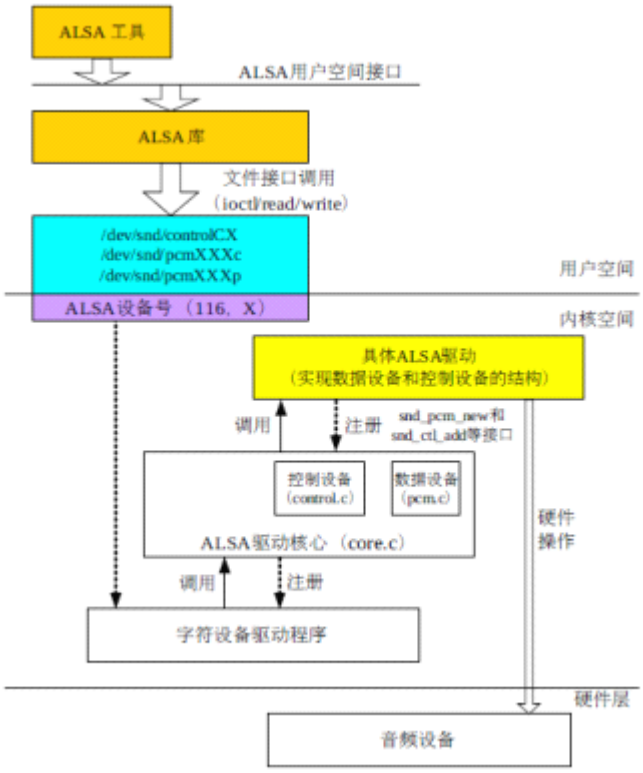

图 3.5 ALSA 音频驱动

6、MTD 驱动

Flash 驱动通常使用 MTD(memory technology device)内存技术设备。

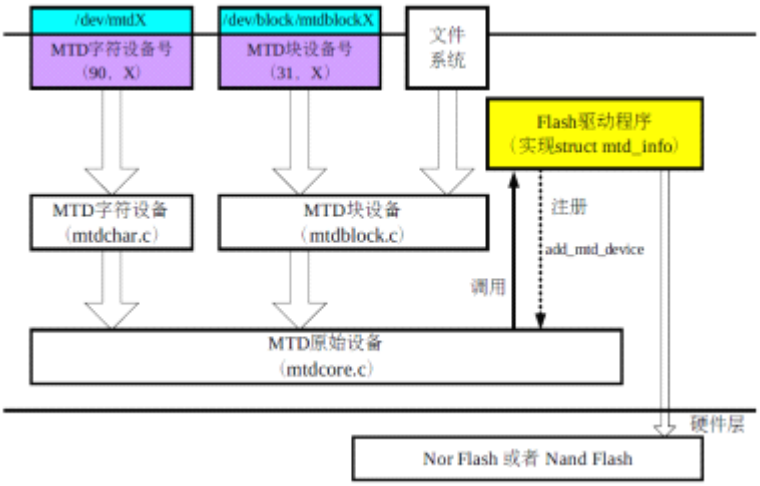

图 3.6 MTD 驱动

7、蓝牙驱动

在 Linux 中，蓝牙设备驱动是网络设备，使用网络接口，蓝牙设备的网络协议:协议族 AF_BLUETOOTH,蓝牙驱动程序一般都通过标准的 HCI 控制实现，根据硬件接口和初始化流程的不同，又存在一些差别，初始化动作一般是一些晶振频率，波特等基础设置。

比如 CSR 的芯片一般通过 BCSP 协议初始化配置，在激活标准 HCI 控制流程，Linux 下 bluez 可以使用 HCI 与芯片建立通信(一般是 hciattach+hciconfig)便可以利用其上的标准协议(SCO,L2CAP 等)与蓝牙通信使其正常工作。

8、WLAN 设备驱动

网络设备，使用网络接口，Wlan 在用户空间使用标准的 socket 接口。

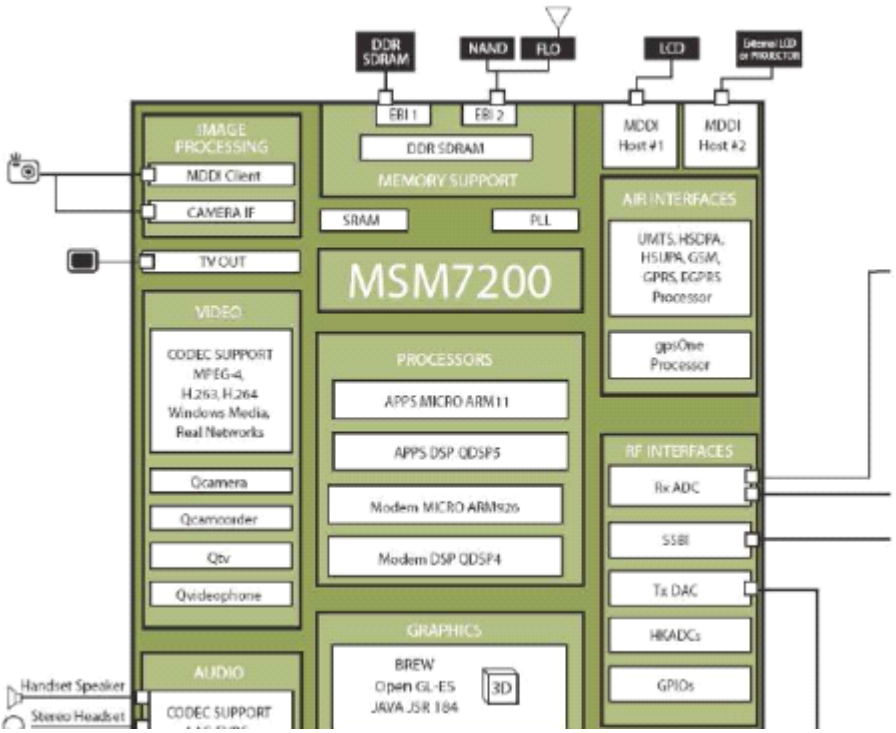

图 3.7 Wlan 在用户空间使用标准的 socket 接口

Google 基于 Gold-Fish 的手机采用 QualComm 公司的高性能处理芯片 MSM7201A，该芯片以 ARM 11 作为 Application 应用处理器，以 ARM926T 作为 BaseBand 主处理器(主要承载 GSM/GPRS/EDGE/3G 等协议栈处理)，支持 Java 硬加速(ARM 本身也自带 Java 硬加速)，包含 GPS Processor，支持 2D/3D 图形加速(每秒可处理 4 百万个多边形)另外还支持最大 800 万像素的 Sensor、支持 MPEG 4/H.263/Real Media 等多种 Codec。

【Android 底层实例】

4.1 Android Led控制实验

4.1.1 代码讲解

1、编写 HAL 层代码

一般来说 HAL moudle 需要涉及的是三个关键结构体：

struct hw_module_t;

struct hw_module_methods_t;

struct hw_device_t;

下面结合代码说明这3个结构的用法

文件：/hardware/modules/include/weiyan/led.h

```
// //HAL 规定不能直接使用 hw_module_t 结构
//因此需要做这么一个继承。
struct led_module_t {
   struct hw_module_t common;
};

struct led_control_device_t {
      //自定义的一个针对 Led 控制的结构
      //包含 hw_device_t 和支持的 API 操作
   struct hw_device_t common;
   /* supporting control APIs go here */
   int (*set_on)(struct led_control_device_t*dev,int32_t led);
   int (*set_off)(struct led_control_device_t *dev,int32_t led);
};

/*********************************************************:

struct led_control_context_t {
    struct led_control_device_t device;
};

//定义一个 MODULE_ID,
//HAL 层可以根据这个 ID 找到我们这个 HAL stub

#define LED_HARDWARE_MODULE_ID "led"
```

文件：led.c

```
int led_on(struct led_conteol_device_t *dev,int32_t led)
{
    int fd;
```

```
    char buff[3] = "";
    int size = 0;
    if((fd = open(DEVICE_NAME,O_RDWR))==-1){
            LOGI("open leds fail");
            return 0;
        }
    //memset(buff,'1',5);
    memset(buff,0,3);
    buff[0]=1;//1表示开 led 灯
    buff[1]=(char)led;//控制哪只 led(0,1,2,3)灯
    size=write(fd,buff,sizeof(buff));
    LOGI("LED Stub:set%d on",led);
    close(fd);

    return 0;
} ? end led_on ?
```

```
int led_off(struct led_control_device_t *dev,int32_t led)
{
    int fd;
    char buff[5] = "";
    int sizt = 0;
    if((fd = open(DEVICE_NAME,O_RDWR))==-1){
                LOGI("open leds fail");
                return 0;
            }
    memset(buff,0,3);
    buff[0]=0;//表示关 led 灯
    buff[1]=(char)led;//控制哪只 led(0,1,2,3)灯
    size=write(fd,buff,sizeof(buff))
    LOGI("LED Stub:set%d off",led);
    Close(fd);

    return 0;
}
```

```
static int led_device_open(const struct hw_module_t* module,const
char* name,
        struct hw_device_t** device)
{
        struct led_control_device_t* dev;

        dev = (struct led_control_device_t*)malloc(sizeof(*dev));
```

```
        memset(dev,0,sizeof(*dev));

        dev->common.tag = HARDWARE_DEVICE_TAG;
        dev->common.version = 0;
        dev->common.module = module;
        dev->common.close = led_device_close;

        dev->set on = led_on;//实例话支持的操作
        dev->set off = led_off;
        //将实例化后的 led_control_device_t 地址返回给 jni 层
        //这样 jni 层就可以直接调用 led_on、led_off、led_device_close 方法
        *device = &dev->common;

success
    return 0;
} ? end led_device_open ?
```

```
//向系统注册一个 ID 为 LED_HARDWARE_MODULE_ID 的 stub
//注意这里的 HAL_MODULE_INFO_SYM 不能修改
const struct led_module_t HAL_MODULE_INFO_SYM = {
  common:{
    tag:HARDWARE_MODULE_TAG,
    version_major:1,
    version_minor:0,
    id:LED_HARDWARE_MODULE_ID,
    name:"Sample LED Styb",
    author:"The Mokoid Open Source Project",
    methods:&led_module_methods,
  }
  /*supporting APIs go here */
};
```

2、JNI 层文件：/frameworks/base/service/com_mokoid_server_LedService.cpp

```
stuct led_control_device_t *sLedDevice = Null;

static jboolean weityan_setOn(JNIEnv* env, jobject thiz, jint led){

  LOGI("LedService JNI:weityan_setOn() is invoked.");

  if(sLedDevice == NULL){
     LOGI("LedService JNI:sLedDevice was not fetched correctly.");
     return -1;
  }else {
```

```
    return sLedDevice->set_on(sLedDevice, led);//调用 HAL 层的方法
  }
}

static jboolean weiyan_setOff(JNIEnv* env, jobject thiz, jint led){

  LOGI("LedService JNI:weiyan_setOff() is invoked.");

  if(sLedDevice == NULL){
     LOGI("LedService JNI:sLedDevice was not fetched correctly.");
     return -1;
  }else {
    return sLedDevice->set_off(sLedDevice, led);//调用 HAL 层的方法
  }
}
```

```
static inline int led_control_open(const struct hw_module_t* module,
        struct led_control_device_t** device){
   return module->methods->open(module,
         LED_HARDWARE_MODULE_ID, (struct hw_device_t**)device);
      //这个过程非常重要
      //JNI 通过 LED_HARDWARE_MODULE_ID 找到对应的 Stub
}

static jboolean
Weiyan_init(JNIEnv *env, jclass clazz)
{
  led_module_t* module;
  //根据 LED_HARDWARE_MODULE_ID 找到应对的 hw_module_t
  if(hw_get_module(LED_HEARDWARE_MODULE_ID, (const hw_module_t**)&module) == 0){
     LOGI("LedService JNI:LED Stub found.");
     if(led_control_open(&module->common, &sLedDevice) == 0){
       LOGI("LedService JNI:Got Stub operations.");
        return 0;
     }
  }
  LOGE("LedService JNI: Get Stub operations failed.");
  return -1;
}
```

```
/* JNINativeMethod 是 JNI 层注册的方法
 *Framework 层可以使用这些方法
 *_init, _set_on, _set_off 是 Framework 层调用的方法
```

```
 *()Z 无参数返回值为 bool 型
 *(I)Z 整型参数返回值为 bool 型
 */

static const JNIN ative Method gMethods = {
    {"_init",          "()Z",
                (void*)weiyan_init},//framework 层调用 _init 时促发
    {"_set_on",         "(I)Z",
                  (void*)weityan_setOn },
    {"_set_off",         "(I)Z",
                  (void*)weiyan_setOff },
};
```

```
static int registerMethods(JNIEnv* env){
  static const char* const kClassName =
    "com/weiyan/server/LedService";//必须与 Framework 层的 service 类名相同
  jclass clazz;

  /* look up the class */
  clazz = env->FindClass(kClassName);
  if(clazz == Null){
     LOGE("Cant't find class %s\n",kClassName);
     return - 1;
  }

  /* register all the methods */
  if(env->RegisterNativrs(clazz,gMethods,
       sizeof(gMethods)/ sizrof(gMethods[0]))! = JNI_OK)
  {
     LOGE("Failed registe ring methods for %s\n",kClassName);
     return - 1;
  }

  /* fill out the rest of the ID cache */
  return 0;
} ?  end registerMethods ?
```

```
/*
 *Framework 层加载 JNI 库时调用
 */

jint JNI_OnLoad(JavaVM* vm,void* reserved){
  JNIEnv* env = NULL;
```

```
    jint result = - 1;

    if(vm->GetEnv((void**) &env,JNI_VESION_1_4)! = JNI_OK){
        LOGE("ERROR:GetEnv failed\n");
        goto ↓bail;
    }
    assert(env! = NULL);
    //注册 JNINativeMethod 方法
    if (registerMethods(env)! = 0){
         LOGE("ERROR:PlatformLibrary native registration failed\n");
         goto ↓bail;
    }

    /* success - - return valid version number */
    result = JNI_VERSION_1_4;

bail:
    return result;
} ?  end JNI_OnLoad ?
```

3、Framework 层的 service，文件：/frameworks/base/service/java/com/weiyan/server

```
public final class LedService extends ILedService.Stub {

    static{
        System.load("/system/lib/libmokoid_runtime.so");//加载 jni 动态库
    }

    public LedService(){
        Log.i("LedService","Go to get LED Stub...");
        _init();
    }

    /*
     * Mokoid LED native methods.
     */
    public boolean setOn(int led){
        Log.i("MokoidPlatform","LED On");
      return _set_on(led);
    }

    public boolean setOff(int led){
        Log.i("MokoidPlatform","LED Off");
      return _set_off(led);
```

```
    }

    private static native boolean _init();//声明 jni 可以使用的方法
    private static native boolean _set_on(int led);
    private static native boolean _set_off(int led);
```

4、APP 测试程序 （属于 APP 层）

APP 层的两种调用模式

(1) Android 的 app 可以直接通过 service 调用.so 格式的 jni

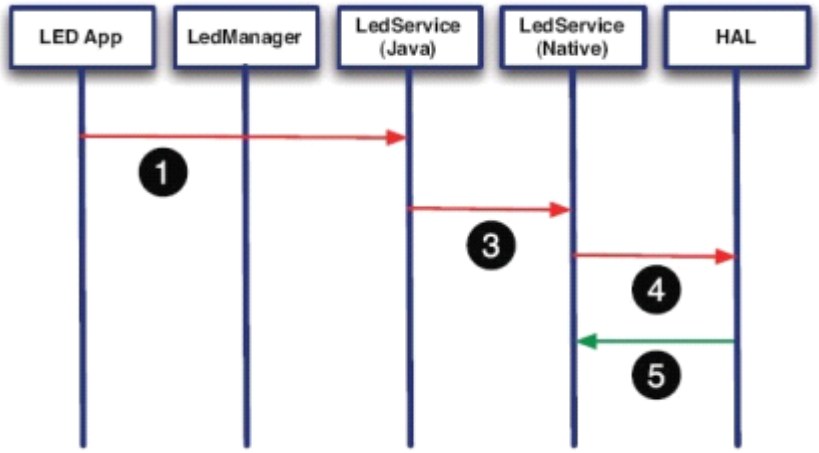

图4.1

(2) 经过 Manager 调用 service

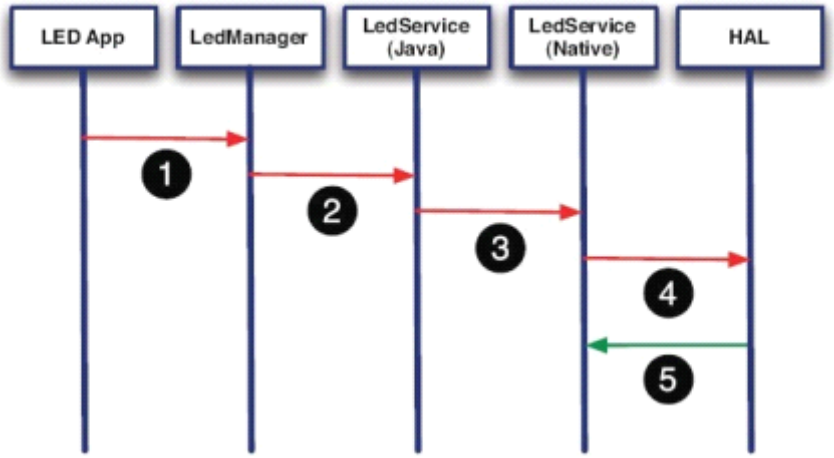

图4.2

5、Manager（属于 Framework 层）

```
public class LedManager
{
    private static final String TAG = "LedManager";
    private ILedService mLedService;

    Public LedManager() {
```

```
        //利用 ServiceManager 获取 LedService,从而调用它
        //提供的方法，这要求 LedService 必须已经增加
        //service 进程中完成
        mLedService = ILedService.Stub.as Interface(
                        Service Manager.getService("led"));

        if (mLedService! = null) {
            Log.i(TAG,"The LedManager object is ready.");
        }
    }

    public boolean LedOn(int n) {
        boolean result = false;

        try {
          resrle = mLedService.setOn(n);
        } catch(Remote Exception e) {
          Log.e(TAG,"Remote Exception in LedManager.LedOn:",e);
        }
        return result;
      }

      public boolean LedOff(int n) {
        boolean result = false;

        try {
          result = mLedService.setOff(n);
        } catch(Remote Exception e) {
          Log.e(TAG,"Remote Exception in LedManager.LedOff:",e);
```

因为 LedService 和 LedManager 在不同的进程，所以要考虑到进程通讯的问题。Manager 通过增加一个 aidl 文件来描述通讯接口文件：

weiyan/frameworks/base/core/java/weiyan/hardware/ILedService.aidl

```
package mokoid.hardware;

interface ILedService
{
    boolean setOn(int led);
    boolean setOff(int led);
    }
```

系统的 aidl 工具会将 ILedService.aidl 生成 ILedService.java 文件，实现 IledService。

6、SystemServer （属于 APP 层）文件：
/apps/LedTest/src/com/weiyan/LedTest/LedSystemServer.java

```
pubic class LedSystemServer extends Service {

  @Override
  public IBinder onBind(Intent intent) {
    return null;
  }

  public void onStart(Intent intent,int startId){
    Log.i("LedSystemServer","Start LedService...");

    /* Please also see SystemServer.java for your interests. */
    LedService ls = new LedService();

    try {
      //将 Ledservice 添加到 ServiceManager
      ServiceManager.addService("led",ls);
    }catch(Runtime Exception e) {
      Log.e("LedSystemServer","Start LedService failed.");
    }
  }
} ? eng LedSystemServer ?
```

4.1.2 APP 测试程序

文件：weiyan/apps/LedTest/src/com/weiyan/LedTest/LedTest.java

加载方法：

1、把光盘提供的 weiyan.tar.gz 解压到/opt/ android_froyo_smdk

```
$ cd /opt/ android_froyo_smdk
$ tar -jxvf weiyan.tar.bz2
```

2、修改 build/core/config.mk 文件防止编译找不到 led.h 头文件。

```
$cd /opt/ android_fsroyo_smdk
$gedit build/core/config.mk
找到 SRC_HEADERS := \
    $(TOPDIR)system/core/include \
  在后面加入
    $(TOPDIR)weiyan/hardware/modules/include
```

```
SRC_HEADERS :=\
```

```
        $(TOPDIR)system/core\include \
        $(TOPDIR)hardware/libhardware/include \
        $(TOPDIR)hardware/libhardware_legacy/inclyde \
        $(TOPDIR)hardware/ril/include \
        $(TOPDIR)dalvik/libnativehelper/include \
        $(TOPDIR)frameworks/base/include \
        $(TOPDIR)frameworks/base/opengl/include \
        $(TOPDIR)external/skia/include \
        $(TOPDIR)weiyan/hardware/modules/include
```

3、编译工程

```
$ source /opt/android_froyo_smdk/build/envsetup.sh
$ export TARGET_PRODUCT=sec_smdkv210
$ mmm  /opt/android_froyo_smdk/weiyan
```

编译成功后会如下路径生成 apk 文件,库文件,jar 包等

/opt/android_froyo_smdk/out/target/product/smdkv210/system/app/LedClient.apk
/opt/android_froyo_smdk/out/target/product/smdkv210/system/app/LedTest.apk
/opt/android_froyo_smdk/out/target/product/smdkv210/system/framework/ledctl.jar
/opt/android_froyo_smdk/out/target/product/smdkv210/system/lib/hw/led.smdkv210.so
/opt/android_froyo_smdk/out/target/product/smdkv210/system/lib/libled.so
/opt/android_froyo_smdk/out/target/product/smdkv210/system/lib/libmokoid_runtime.so

把 LedClient.apk, LedTest.apk 放到 android 的 system/app 目录，把 ledctl.jar 放到 system/framework 目录,把 led.smdkv210.so 放到 system/lib/hw 目录，把 ibled.so, libmokoid_runtime.so 放到 system/lib 目录下。

4、为了 android 桌面能显示我们的 LedClient.apk, LedTest.apk 程序，把 weiyan/framework--s/base/service/com.weiyan.server.xml 放到 android 的 system/etc/permissions 目录下。

5、加载 led 驱动模块。

把 leds.ko 复制到 android 的 system 目录下，执行

```
#insmod leds.ko
#chmod 666 /dev/leds
```

6、运行 LedClient.apk, LedTest.apk

4.2 基于 PXA310上的 Android 手机的驱动开发

4.2.1 电源管理驱动：

Android 电源驱动会在/sys 目录下创建子目录。android_power 并生成相应的属性文件，通过 attribute 的 show/store 来访问和设置 power state/request。

```
# Is /sys/android_power
state
reuest_state
acquire_full_wake_lock
acuqire_partial_wake_look
Release_wake_lock
```

```
# cat state
0-1-1
# cat request_state
wake
# android_power: sleep (0->2) at 157660949707

# cat request_state
standby
#
```

4.2.2 Android HAL 层对电源控制的操作：hardware\libhardware\power\power.c

```
const char * const OLD_PATHS[]=
{

  "/sys/android_power/release_wake_lock",
  "/sys/android_power/request_state"
};
```

```
# cat request_state
standby
# echo wake > request_state
android_power: wakeup (2->0) at 270024017733
# android_power_suspend: done
cat request_state
wake
# echo standby > request_state
android_power: sleep (0->2) at 285930358886
# cat request_state
standby
```

4.2.3 USB 驱动：

Google 没有使用原来的那套 gadget 驱动架构（file_storage.c），而是参考 file_storage.c 实现了一个新的模型——composite 模型：

```
composite.c          //实现 android 下 USB 管理的框架模型
android.c            //实现具体的 USB 功能管理
f_mass_storage.c     //实现优盘功能
```

f_adb.c　　//实现 adb 功能

1）该框架下，有三个设备：composite 设备，优盘设备，adb 设备。
2）枚举时，首先枚举 composite 设备，再枚举优盘设备，最后枚举 adb 设备。
3）composite 设备被枚举时：
a）在获取 DEVICE 描述符时，将 VID,PID 上报给 host。
b）host 第一次请求 CONFIG 描述符时，composite 设备告诉 host，它有一个 CONFIG 两个 interface（即两个功能），以及告诉 host 自己使用的端点 OUT 的地址。
C）host 会根据 interface 的个数决定枚举的次数（这对应着优盘枚举和 adb 枚举）

4.2.4　键盘驱动：

1、keyboard 驱动要通过 input_register_device()调用注册成标准的输入设备/dev/inpu--t/event0驱动上报的按键值要和 Android 系统里面的/system/usr/keylayout/*.kl 文件里面的记录一致，否则会导致 android 系统不能正确识别按键消息。

key 232	DPAD_CENTER	WAKE_DROPPED
key 108	DPAD_DOWN	WAKE_DROPPED
key 103	DPAD_UP	WAKE_DROPPED
key 102	HOME	WAKE
key 105	DPAD_LEFT	WAKE_DROPPED
key 106	DPAD_RIGHT	WAKE_DROPPED

2、Linux 输入系统的任何一次事件通知包涵如下三个信息：

事件类型＋事件码＋事件值（键值）

=> 这三个信息的作用如下，并举例说明：

(1)对键盘输入设备来说：
a.事件类型 EV_KEY，表明是键盘送上来的数据。
b.事件码 KEY_CAMERA 表明是键盘上的 camera 键被操作了。
c.事件值表示该 camera 键是按下还是松开了。

(2)对触摸屏输入设备来说：
a.事件类型 EV_ABS,表明是触摸屏送上来的绝对坐标值数据。
b.事件码 ABS_X 表明是触摸屏当前点的 x 坐标。
c.事件值表示该当前点的 x 坐标的具体绝对值。

3、Android 启动时会检测输入设备：

```
I/EventHub( 690):New device:path=/dev/input/event2 name=ADS784x Touchscreen id=0x10000
                 (of 0x1)index=1 fd=44 classes=0x2

E/HAL  ( 690):load:module=/system/lib/hw/sensors.default.so error=Cannot find library

D/SensorManager( 690):found sensor:null,handle=0

I/EventHub( 690):New device:path=/dev/input/event1 name=gpio-keys id=0x10001(of 0x2)
                 index=2 fd=46 classes=0x0

I/SystemServer( 690):Starting Bluetooth Ssrvice.

I/EventHub( 690):New device:path=/dev/input/event0 name=omap_twl4030keypad id=0x10002
                 (of 0x3)index=3 fd=47 classes=0x1
```

```
//A:
I/EventHub( 690):New keyboard:publicID=65538 device->id=65538
        devname='omap_twl4030keypad' propName='hw.keyboards.65338.devname'
        keylayout='/system/usr/keylayout/qwerty.kl'
```

4.3 Android 内核驱动——Alarm

4.3.1 基本原理

Alarm 闹钟是 android 系统中在标准 RTC 驱动上开发的一个新的驱动，提供了一个定时器用于把设备从睡眠状态唤醒，当然因为它是依赖 RTC 驱动的，所以它同时还可以为系统提供一个掉电下还能运行的实时时钟。

当系统断电时，主板上的 rtc 芯片将继续维持系统的时间，这样保证再次开机后系统的时间不会错误。当系统开始时，内核从 RTC 中读取时间来初始化系统时间，关机时便又将系统时间写回到 rtc 中，关机阶段将有主板上另外的电池来供应 rtc 计时。Android 中的 Alarm 在设备处于睡眠模式时仍保持活跃，它可以设置来唤醒设备。

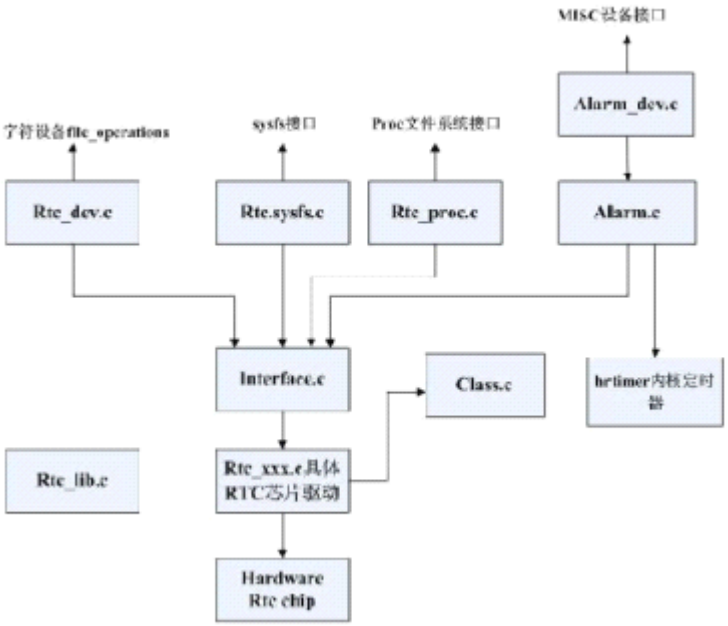

图4.3

图4.3为 android 系统中 alarm 和 rtc 驱动的框架。Alarm 依赖于 rtc 驱动框架，但它不是一个 rtc 驱动，主要还是实现定时闹钟的功能。相关源代码在 kernel/drivers/rtc/alarm.c 和 drivers/rtc/alarm_dev.c。

其中 alarm.c 文件实现的是所有 alarm 设备的通用性操作，它创建了一个设备 class，而 alarm_dev.c 则创建具体的 alarm 设备，注册到该设备 class 中。alarm.c 还实现了与 interface.c 的接口，即建立了与具体 rtc 驱动和 rtc 芯片的联系。alarm_dev.c 在 alarm.c 基础包装了一层，主要是实现了标准的 miscdevice 接口，提供给应用层调用。

可以这样概括：alarm.c 实现的是机制和框架，alarm_dev.c 则是实现符合这个框架的设备驱动，alarm_dev.c 相当于在底层硬件 rtc 闹钟功能的基础上虚拟了多个软件闹钟。

4.3.2 关键数据结构

1、alarm 定义在 include/linux/android_alarm.h 中。

```
struct alarm {
     struct rb_node           node;
     enum android_alarm_type  type;
     ktime_t softexpires;      //最早的到期时间
     ktime_t expires;          //绝对到期时间
     void (*function)(struct alarm *); //当到期时系统回调该函数
};
```

2、alarm_queue

```
struct alarm_queue {
     struct rb_root    alarms; //红黑树的根
     struct rb_node   *first; //指向第一个 alarm device,即最早到时的
     struct  hrtimer   timer; //内核定时器,android 利用它来确定 alarm 过期
                              时间
     ktime_t           delta; //是一个计算 elasped realtime 的修正值
     bool              stopped;
     ktime_t           stopped_time;
};
```

这个结构体用于将前面的 struct alarm 表示的设备组织成红黑树。它是基于内核定时器来实现 alarm 的到期闹铃的。

4.3.3 关键代码分析

1、alarm_dev.c 该文件依赖于 alarm.c 提供的框架，实现了与应用层交互的功能，具体说就是暴露出 miscdevice 的设备接口。Alarm_dev.c 定义了几个全局变量：

(1) 每种类型一个 alarm 设备，android 目前创建了 5 个 alarm 设备。

```
static struct alarm alarms[ANDROID_ALARM_TYPE_COUNT];
```

(2) wake lock 锁，当加锁时，阻止系统进 suspend 状态。

```
static struct wake_lock alarm_wake_lock;
```

(3) 标志位，alarm 设备是否被打开。

```
static int alarm_opened;
```

(4) 标志位，alarm 设备是否就绪。所谓就绪是指该 alarm 设备的闹铃时间到达，但原本等待在该 alarm 设备上的进程还未唤醒，一旦唤醒，该标志清零。

```
static uint32_t alarm_pending;
```

(5)标志位，表示 alarm 设备是否 enabled，表示该设备设置了闹铃时间（并且闹铃时间还未到），一旦闹铃时间到了，该标志清零。

```
static uint32_t alarm_enabled;
```

(6)标志位，表示原先等待该 alarm 的进程被唤醒了（它们等待的 alarm 到时了）。

```
static uint32_t wait_pending;
```

(7)该文件提供的主要函数有：

A. 模块初始化和 exit 函数：alarm_dev_init 和 alarm_dev_exit

B. 模块 miscdevice 标准接口函数：alarm_open、alarm_release 和 alarm_ioctl

C. alarm 定时时间到时候的回调函数：alarm_triggered

(8)alarm_dev_init 初始化函数调用 misc_register 注册一个 miscdevice。

```
static int __init alarm_dev_init(void){
    int err;
    int i;

    err = misc_register(&alarm_device);
    if (err)
        return err;

    for (i = 0; i < ANDROID ALARM TYPE COUNT; i++)
        alarm_init(&alarms[i], i, alarm_triggered);
    wake lock init(&alarm wake lock, WAKE LOCK SUSPEND, "alarm");

    return 0;
}
```

(9)该设备称为 alarm_device，定义如下：

```
static struct miscdevice alarm_device = {
    .minor = MISC_DYNAMIC_MINOR,
    .name = "alarm",
    .fops = &alarm_fops,
};
```

(10)对应的 file operations 为 alarm_fops，定义为：

```
static const struct file operations alarm_fops = {
    .owner = THIS_MODULE,
    .unlocked_ioctl = alarm ioctl,
    .open = alarm_open,
    .release = alarm release,
};
```

(11)然后为每个 alarm device 调用 alarm_init 初始化，这个函数代码在 alarm.c 中，如下：

```
void alarm_init(struct alarm *alarm,
                enum android alarm_type type,
                void (*function)(struct alarm *)){
    RB_CLEAR_NODE(&alarm->node);
    alarm->type = type;
    alarm->function = function;
    pr_alarm(FLOW, "created alarm, type %d, func %pF\n", type, function);
}
```

(12)就是初始化 alarm 结构体，设置其回调函数为 alarm_triggered。最后调用 wake_lock_init 初始化 alarm_wake_lock，它是 suspend 型的。alarm_triggered 是回调函数，当定时闹铃的时间到了，alarm_timer_triggered 函数会调用该函数（详细请看 alarm.c 的 alarm_timer_triggered 函数）。

```
static void alarm triggered(struct alarm *alarm){
    unsigned long flags;
    uint32 t alarm type mask = 1U << alarm->type;

    pr_alarm(INT, "alarm_triggered type %d\n", alarm->type);
    spin lock irqsave(&alarm slock, flags);
    if (alarm_enabled & alarm_type_mask) {
        wake lock timeout(&alarm wake lock, 5 * HZ);
        alarm_enabled &= ~alarm_type_mask;
        alarm pending |= alarm type mask;
        wake up(&alarm wait queue);
    }
    spin_unlock_irqrestore(&alarm_slock, flags);
}
```

(13)这个函数里调用 wake_lock_timeout 对全局 alarm_wake_lock(超时锁，超时时间是 5 秒)加锁，禁止对应的 alarm 设备。唤醒所有等待在该 alarm 设备上的进程。这时，如果 AP 层呼叫 ioctl(fd, ANDROID_ALARM_WAIT)，会返回表示等到 alarm 的返回值（这个会在 AlarmManagerSevice.java 中细述）。

alarm_ioctl 定义了以下命令:

ANDROID_ALARM_CLEAR	清除 alarm，即 deactivate 这个 alarm
ANDROID_ALARM_SET_OLD	设置 alarm 闹铃时间
ANDROID_ALARM_SET	同上
ANDROID_ALARM_SET_AND_WAIT_OLD	设置 alarm 闹铃时间并等待这个 alarm
ANDROID_ALARM_SET_AND_WAIT	同上
ANDROID_ALARM_WAIT	等待 alarm
ANDROID_ALARM_SET_RTC	设置 RTC 时间
ANDROID_ALARM_GET_TIME	读取 alarm 时间，根据 alarm 类型又分四种情况

ANDROID_ALARM_RTC_WAKEUP ANDROID_ALARM_RTC	读取系统当前时间，即从1970-1-1-00：00：00至今过去多少秒，精确到us
ANDROID_ALARM_ELAPSED_REALTIME_WAKEUP ANDROID_ALARM_ELAPSED_REALTIME	读取 elapsed time
ANDROID_ALARM_SYSTEMTIME ANDROID_ALARM_TYPE_COUNT	读取 system time，代码注释说是所谓的 monotonic clock，就是ANDROID_ALARM_RTC 的时间做了个修正

2、**alarm.c**　该文件完成主要功能有：

a.创建一个 alarm class，所有 alarm 设备都属于这个类；

b.注册了 platform driver，提供 suspend 和 resume 支持；

c.实现了一系列函数，包括 alarm_init，alarm_start_range，alarm_cancel，alarm_timer_triggered 函数等。

(1)Alarm.c 的初始化函数 alarm_driver_init 如下：

```
static int  init alarm driver init(void){
    int err;
    int i;

    for (i = 0; i < ANDROID_ALARM_SYSTEMTIME; i++) {
        hrtimer init(&alarms[i].timer, CLOCK REALTIME, HRTIMER MODE ABS);
        alarms[i].timer.function = alarm_timer_triggered;
    }
    hrtimer_init(&alarms[ANDROID_ALARM_SYSTEMTIME].timer,
            CLOCK MONOTONIC, HRTIMER MODE ABS);
    alarms[ANDROID ALARM SYSTEMTIME].timer.function = alarm timer triggered;

    err = platform driver register(&alarm driver);
    if (err < 0)
        goto err1;
    wake_lock_init(&alarm_rtc_wake_lock, WAKE_LOCK_SUSPEND, "alarm_rtc");
    rtc_alarm_interface.class = rtc_class;
    err = class interface register(&rtc alarm interface);
    if (err < 0)
       goto err2;

    return 0;

err2:
    wake lock destroy(&alarm rtc wake lock);
    platform_driver_unregister(&alarm_driver);
err1:
```

```
    return err;
}
```

(2)该函数初始化5个 alarm device 相关联的 hrtimer 定时器，设置 hrtimer 定时器的回调函数为 alarm_timer_triggered 函数，再注册一个 plateform driver 和 class interface。

如果设置了闹铃时间，则内核通过 hrtimer 定时器来跟踪是否到时间，到时后会触发调用 hrtimer 的处理函数 alarm_timer_triggered。alarm_timer_triggered 的 code 如下：

```
static enum hrtimer restart alarm timer triggered(struct hrtimer *timer){
    struct alarm_queue *base;
    struct alarm *alarm;
    unsigned long flags;
    ktime_t now;

    spin lock irqsave(&alarm slock, flags);

    base = container_of(timer, struct alarm_queue, timer);
    now = base->stopped ? base->stopped time : hrtimer cb get time(timer);
    now = ktime_sub(now, base->delta);

    pr_alarm(INT, "alarm_timer_triggered type %d at %lld\n",
        base - alarms, ktime_to_ns(now));

    while (base->first) {
        alarm = container of(base->first, struct alarm, node);
        if (alarm->softexpires.tv64 > now.tv64) {
            pr_alarm(FLOW, "don't call alarm, %pF, %lld (s %lld)\n",
                alarm->function, ktime_to_ns(alarm->expires),
                ktime to ns(alarm->softexpires));
            break;
        }
        base->first = rb_next(&alarm->node);
        rb_erase(&alarm->node, &base->alarms);
        RB_CLEAR_NODE(&alarm->node);
        pr_alarm(CALL, "call alarm, type %d, func %pF, %lld (s %lld)\n",
            alarm->type, alarm->function,
            ktime to ns(alarm->expires),
            ktime to ns(alarm->softexpires));
        spin_unlock_irqrestore(&alarm_slock, flags);
        alarm->function(alarm);
        spin_lock_irqsave(&alarm_slock, flags);
    }
    if (!base->first)
        pr_alarm(FLOW, "no more alarms of type %d\n", base - alarms);
```

```
    update_timer_locked(base, true);
    spin_unlock_irqrestore(&alarm_slock, flags);
    return HRTIMER_NORESTART;
}
```

(3)它会轮询红黑树中的所有 alarm 节点，符合条件的节点会执行 alarm.function(alarm)，指向 alarm_dev.c 的 alarm_triggered 函数。因为我们在执行 alarm_dev.c 的 alarm_init 时，把每个 alarm 节点的 function 设置成了 alarm_triggered。

请注意这两个函数的区别，alarm_triggered 和 alarm_timer_triggered。前者是 rtc 芯片的 alarm 中断的回调函数，后者是 android alarm_queue->timer 到时的回调函数。

(4)上面说到，alarm_driver_init 注册了一个类接口。

```
class_interface_register(&rtc_alarm_interface)
```

(5)rtc_alarm_interface 的 code 如下：

```
static struct class_interface rtc_alarm_interface = {
    .add_dev = &rtc_alarm_add_device,
    .remove_dev = &rtc_alarm_remove_device,
};
```

(6)在 rtc_alarm_add_device 中，注册了一个 rtc 中断 rtc_irq_register(rtc, &alarm_rtc_task)

```
static int rtc_alarm_add_device(struct device *dev,
                                struct class_interface *class_intf){
    int err;
    struct rtc_device *rtc = to_rtc_device(dev);

    mutex_lock(&alarm_setrtc_mutex);

    if (alarm_rtc_dev) {
        err = -EBUSY;
        goto err1;
    }

    alarm_platform_dev = platform_device_register_simple("alarm", -1, NULL, 0);
    if (IS_ERR(alarm_platform_dev)) {
        err = PTR_ERR(alarm_platform_dev);
        goto err2;
    }
    err = rtc_irq_register(rtc, &alarm_rtc_task);
    if (err)
        goto err3;
    alarm_rtc_dev = rtc;
```

```
    pr_alarm(INIT_STATUS, "using rtc device, %s, for alarms", rtc->name);
    mutex_unlock(&alarm_setrtc_mutex);

    return 0;

err3:
    platform_device_unregister(alarm_platform_dev);
err2:
err1:
    mutex_unlock(&alarm_setrtc_mutex);
    return err;
}
```

(7)中断的回调函数为alarm_triggered_func：

```
static struct rtc_task alarm_rtc_task = {
    .func = alarm_triggered_func
};

static void alarm_triggered_func(void *p){
    struct rtc_device *rtc = alarm_rtc_dev;
    if (!(rtc->irq_data & RTC_AF))
        return;
    pr_alarm(INT, "rtc alarm triggered\n");
    wake_lock_timeout(&alarm_rtc_wake_lock, 1 * HZ);
}
```

(8)当硬件rtc chip的alarm中断发生时，系统会调用alarm_triggered_func函数。alarm_triggered_func的功能很简单，wake_lock_timeout锁住alarm_rtc_wake_lock 1秒。因为这时，alarm会进入alarm_resume，lock住alarm_rtc_wake_lock以防止alarm在此时进入suspend。

(9)AlarmManager.java，该文件提供的接口主要有：

a. 设置闹钟。

```
public void set(int type, long triggerAtTime, PendingIntent operation);
```

b. 设置周期闹钟。

```
public void setRepeating(int type, long triggerAtTime,
long interval,PendingIntent operation);
```

c. 取消闹钟。

```
public void cancel(PendingIntent operation);
```

上面3个函数分别会呼叫到AlarmManagerSevice.java以下三个函数：

```
public void set(int type, long triggerAtTime, PendingIntent operation);
```

```
public void setRepeating(int type, long triggerAtTim
                         long interval,PendingIntent operation);
public void remove(PendingIntent operation);
```

AlarmManagerSevice.java通过JNI机制可以呼叫com_android_server_AlarmManagerService.cpp透出的几个接口。

(10)AlarmManagerSevice.java有关接口的code如下：

```
private native int init();
private native void close(int fd);
private native void set(int fd, int type, long seconds, long nanoseconds);
private native int waitForAlarm(int fd);
private native int setKernelTimezone(int fd, int minuteswest);
```

(11)com_android_server_AlarmManagerService.cpp有关接口的对应code如下：

```
static JNINativeMethod sMethods[] = {
  /* name, signature, funcPtr */
{"init", "()I", (void*)android_server_AlarmManagerService_init},
{"close", "(I)V", (void*)android_server_AlarmManagerService_close},
{"set", "(IIJJ)V", (void*)android_server_AlarmManagerService_set},
{"waitForAlarm", "(I)I",
            (void*)android_server_AlarmManagerService_waitForAlarm},
{"setKernelTimezone", "(II)I",
       (void*)android_server_AlarmManagerService_setKernelTimezone},
};
```

(12)当AP呼叫AlarmManager.java的set或setRepeating函数时，最终会呼叫com_android_server_AlarmManagerService.cpp的。

```
static void android_server_AlarmManagerService_set (JNIEnv* env, jobject obj,
                        jint fd, jint type, jlong seconds, jlong nanoseconds)
```

(13)在此函数中，会执行

```
ioctl(fd, ANDROID_ALARM_SET(type), &ts);
```

然后会呼叫到alarm-dev.c中alarm_ioctl中，接着alarm-dev.c会往它的红黑树中增加一个alarm节点。

(14)在AlarmManagerService开始的时候，会启动一个AlarmThread。在这个AlarmThread中有一个while循环去执行waitForAlarm这个动作，这个函数最终通过JNI机制呼叫到com_android_server_AlarmManagerService.cpp的。

```
static jint android_server_AlarmManagerService_waitForAlarm(JNIEnv*
                                          env,jobject obj, jint fd){
#if HAVE_ANDROID_OS
```

```
    int result = 0;

    do
    {
        result = ioctl(fd, ANDROID_ALARM_WAIT);
    } while (result < 0 && errno == EINTR);

    if (result < 0) {
        LOGE("Unable to wait on alarm: %s\n", strerror(errno));
        return 0;
    }

    return result;
#endif
}
```

(15)从 code 中可以看到，实际上它是在不断地执行 ioctl（fd，ANDROID_ALARM_WAIT)，上面说到，当闹钟到期时，alarm.c 中的 alarm_timer_triggered 函数会调用 alarm_triggered，这时，AP 层在呼叫 ioctl（fd，ANDROID_ALARM_WAIT)时，会返回表示等到 alarm 的返回值。

从 code 中可以看到，实际上它是在不断地执行 ioctl（fd，ANDROID_ALARM_WAIT)，上面说到，当闹钟到期时，alarm.c 中的 alarm_timer_triggered 函数会调用 alarm_triggered，这时，AP 层在呼叫 ioctl（fd，ANDROID_ALARM_WAIT)时，会返回表示等到 alarm 的返回值。

(16)AlarmThread 的 code 如下：

```
private class AlarmThread extends Thread {
    public AlarmThread() {
        super("AlarmManager");
    }

    public void run() {
        while (true)
        {
            int result = waitForAlarm(mDescriptor);

            ArrayList<Alarm> triggerList = new ArrayList<Alarm>();

            if ((result & TIME_CHANGED_MASK) != 0) {
                remove(mTimeTickSender);
                mClockReceiver.scheduleTimeTickEvent();
                Intent intent = new Intent(Intent.ACTION_TIME_CHANGED);
                intent.addFlags(Intent.FLAG_RECEIVER_REPLACE_PENDING);
                mContext.sendBroadcast(intent);
            }
```

```
            synchronized (mLock) {
                final long nowRTC = System.currentTimeMillis();
                final long nowELAPSED = SystemClock.elapsedRealtime();
                if (localLOGV) Slog.v(
                    TAG, "Checking for alarms... rtc=" + nowRTC
                    + ", elapsed=" + nowELAPSED);

                if ((result & RTC_WAKEUP_MASK) != 0)
                    triggerAlarmsLocked(mRtcWakeupAlarms, triggerList, nowRTC);

                if ((result & RTC_MASK) != 0)
                    triggerAlarmsLocked(mRtcAlarms, triggerList, nowRTC);

               if ((result & ELAPSED_REALTIME_WAKEUP_MASK) != 0)
                    triggerAlarmsLocked(mElapsedRealtimeWakeupAlarms,
                                        triggerList, nowELAPSED);

               // now trigger the alarms
               Iterator<Alarm> it = triggerList.iterator();
               while (it.hasNext()) {
                   Alarm alarm = it.next();
                   try {
                       if (localLOGV) Slog.v(TAG, "sending alarm " + alarm);
                       alarm.operation.send(mContext, 0,
                                   mBackgroundIntent.putExtra(
                                       Intent.EXTRA ALARM COUNT, alarm.count),
                                   mResultReceiver, mHandler);
                       // we have an active broadcast so stay awake.
                       if (mBroadcastRefCount == 0) {
                           mWakeLock.acquire();
                       }
                       mBroadcastRefCount++;

                       BroadcastStats bs = getStatsLocked(alarm.operation);
                       if (bs.nesting == 0) {
                           bs.startTime = nowELAPSED;
                       } else {
                           bs.nesting++;
                       }
                       if (alarm.type == AlarmManager.ELAPSED REALTIME WAKEUP
                           || alarm.type == AlarmManager.RTC_WAKEUP) {
                           bs.numWakeup++;
```

```
                            ActivityManagerNative.noteWakeupAlarm(
                                                    alarm.operation);
                        }
                    } catch (PendingIntent.CanceledException e) {
                        if (alarm.repeatInterval > 0) {
                            remove(alarm.operation);
                        }
                    } catch (RuntimeException e) {
                        Slog.w(TAG, "Failure sending alarm.", e);
                    }
                }
            }
        }
    }
}
```

4.3.4 接口

Android 中，Alarm 的操作通过 AlarmManager 来处理，AlarmManager 系统服务的具体实现在:frameworks/base/services/java/com/android/server/AlarmManagerServic.java 文件中。应用程序中可以通过 getSystemService 获得其系统服务，如下所示：

```
AlarmManager alarms = (AlarmManager)getSystemService(Context.ALARM_SERVICE);
```

为了创建一个新的 Alarm，使用 set 方法并指定一个 Alarm 类型、触发时间和在 Alarm 触发时要调用的 Intent。如果你设定的 Alarm 发生在过去，那么它将立即触发。

这里有4种 Alarm 类型。你的选择将决定你在 set 方法中传递的时间值代表什么，是特定的时间或者是时间流逝：

1、**RTC_WAKEUP**：在指定的时刻（设置 Alarm 的时候），唤醒设备来触发 Intent。
2、**RTC**：在一个显式的时间触发 Intent，但不唤醒设备。
3、**ELAPSED_REALTIME**：从设备启动后，如果流逝的时间达到总时间，那么触发 Intent，但不唤醒设备。流逝的时间包括设备睡眠的任何时间。注意一点的是，时间流逝的计算点是自从它最后一次启动算起。
4、**ELAPSED_REALTIME_WAKEUP**：从设备启动后，达到流逝的总时间后，如果需要将唤醒设备并触发 Intent。

4.3.5 Alarm 的实例

1、 建立一个 AlarmReceiver 继承入 BroadcastReceiver，并在 AndroidManifest.xml 声明。

```
Public static class AlarmReceiver extends BroadcastReceiver {
        @Override
        Public void onReceive (Context context, Intent intent) {
        Toast.makeText(context, “时间到”, Toast.LENGTH_LONG).show();
        }
}
```

2、建立 Intent 和 PendingIntent，来调用目标组件。

```
Intent it = new Intent(this, AlarmReceiver.class);
PendingIntent pi = PendingIntent.getBroadcast(this, 0, intent, 0);
```

3、设置闹钟。

a.获取闹钟管理的实例：

```
AlarmManager am = (AlarmManager)getSystemService(Context.ALARM_SERVICE);
```

b.设置单次闹钟：

```
am.set(AlarmManager.RTC_WAKEUP, System.currentTimeMillis() + (5*1000), pi);
```

c.设置周期闹钟：

```
am.setRepeating(AlarmManager.RTC_WAKEUP, System.currentTimeMillis() + (10*1000), (24*60*60*1000), pi);
```

【Android 驱动源码】（技术达人：高铜 选稿）

5.1 CameraSerivce 服务的注册流程。

在 init.rc 中能看到如下语句：

```
service media /system/bin/mediaserver
  user media
  group system audio camera graphics inet net_bt net_bt_admin
```

这个服务的入口是 Main_mediaservice.c 中的 main()函数。而且是在 servicemanager 服务之后才启动的。

5.1.1 CameraSerivce 服务的注册入口

```
@Main_mediaservice.c)
int main(int argc, char** argv){
    sp<ProcessState> proc(ProcessState::self());
    sp<IServiceManager> sm = defaultServiceManager();
                        //取得 Service_Manager 的代理句柄
    AudioFlinger::instantiate();              //Audio 服务
    MediaPlayerService::instantiate();        //mediaPlayer 服务
    CameraService::instantiate();             //Camera 服务
    ProcessState::self()->startThreadPool();  //为进程开启缓冲池
    IPCThreadState::self()->joinThreadPool(); //将进程加入到缓冲池
}
```

这个函数也比较简单，就是 几个子服务的初始化。这里我们只关心 CameraService 的初始化。最后这个函数将 启动自己的线程，等待其他应用来请求服务。

5.1.2 Camera 服务的初始化

```
@CameraService.cpp
void CameraService::instantiate() {
    defaultServiceManager()->addService(
          String16("media.camera"), new CameraService());
}
```

这个函数非常简单，就是取得服务管理器的代理句柄，并调用他们 addServie 服务函数，addService 的参数要注意，一是"media.camera"字符串，在获取的时候，就要输入这个字符串。在 getService 的时候将看到这一样的字符串。

另外第二个参数传入的是 CameraService 类的一个对象。在我们获取这个服务的时候，将获得 CameraService 类的一个代理对象。

5.1.3 获取服务管理器的代理句柄

```
@IServiceManager.cpp
sp<IServiceManager> defaultServiceManager()
{
    if (gDefaultServiceManager != NULL) return gDefaultServiceManager;
    {
        if (gDefaultServiceManager == NULL)
        {
            gDefaultServiceManager = interface_cast<IServiceManager>(
                    ProcessState::self()->getContextObject(NULL));
        }
    }
    return gDefaultServiceManager;
}
```

这个函数第一次被调用前 gDefaultServiceManager 为 NULL，因此系统要调用 ProcessState::self()获取一个 ProcessState 实例。该 ProcessState 会打开/dev/binder 驱动供 IPCThreadState 使用，从而确保 supportsProcesses()返回 true。

```
@ProcessState.cpp
sp<IBinder> ProcessState::getContextObject(const sp<IBinder>& caller)
{     // caller 这个参数被忽略。
if (supportsProcesses()) {
    return getStrongProxyForHandle(0);
                            // caller 这个参数被忽略，直接传入参数 0
 } else {
        return getContextObject(String16("default"), caller);
        }
 }
@ProcessState.cpp
bool ProcessState::supportsProcesses() const
{
    return mDriverFD >= 0; //表示如果有 IBinder 驱动，则返回为 true.
}

@ProcessState.cpp
sp<IBinder> ProcessState::getStrongProxyForHandle(int32_t handle)
{
sp<IBinder> result;
handle_entry* e = lookupHandleLocked(handle);
if (e != NULL) {
    IBinder* b = e->binder;
    if (b == NULL || !e->refs->attemptIncWeak(this)) {
            b = new BpBinder(handle);
```

```
            e->binder = b;
        if (b) e->refs = b->getWeakRefs();
        result = b;
        } else {
        result.force_set(b);
        e->refs->decWeak(this);
            }
    }
return result; //通过 getContextObject 返回一个 BpBinder 实例
}
```

在第一次调用时，b 为 NULL，因此会先创建一个 BpBinder 实例。从以上调用关系可知 ProcessState::self()->getContextObject(NULL)最终将返回一个 BpBinder 实例,这个函数的返回值将 作为 interface_cast<IServiceManager>(obj)的参数。

5.1.4 BpBinder 实例转换为 IServiceManager

interface_cast 定义如下：

```
@IInterface.h
template<typename INTERFACE>
inline sp<INTERFACE> interface_cast(const sp<IBinder>& obj)
{
return INTERFACE::asInterface(obj);
}
```

查找 INTERFACE::asInterface(obj)定义如下

```
@IInterface.h
#defineIMPLEMENT_META_INTERFACE(INTERFACE, NAME)     \
const String16 I##INTERFACE::descriptor(NAME);                    \
const String16 &I##INTERFACE::getInterfaceDescriptor() const    \
return I##INTERFACE::descriptor;                                  \
}                                                    \
sp<I##INTERFACE> I##INTERFACE::asInterface(const sp<IBinder>& obj)    \
{                                                    \
    sp<I##INTERFACE> intr;                                        \
    if(obj!=NULL){                                    \
    intr= static_cast<I##INTERFACE*>( \
            obj->queryLocalInterface(                                    \

                    I##INTERFACE::descriptor).get());         \
                    if (intr==NULL)                        \
                    intr = new Bp##INTERFACE(obj);          \
                    }                                     \
    }                           \
```

```
    return intr;                    \
}               \
I##INTERFACE::I##INTERFACE(){}    \
I##INTERFACE::~I##INTERFACE(){}   \
```

将这个宏 interface_cast<IServiceManager>(obj)扩展后最终得到的是：

```
sp<IServiceManager> IServiceManager::asInterface(const sp<IBinder>& obj)
{
    sp<IServiceManager> intr;
    if (obj != NULL) {
        intr = static_cast<IServiceManager*>(
                obj->queryLocalInterface(
                        IServiceManager::descriptor).get());
                        if (intr == NULL) {
                            intr = new BpServiceManager(obj);
                        }
    }
    return intr;
}
```

即返回一个 BpServiceManager 对象,这里 obj 就 是前面创建的 BpBInder 对象。

5.1.5 BpServiceManager 对象与 IServiceManager 的关系

@IServiceManager.cpp

从 class BpServiceManager 定义就可知道，BpServiceManager 是 IServiceManager 的基类,也就是 IServiceManager 的一个实现类。

```
class BpServiceManager : public BpInterface<IServiceManager>
{
public:
    BpServiceManager(const sp<IBinder>& impl)
        : BpInterface<IServiceManager>(impl)
    {
    }
```

5.1.2 Camera Service 注册具体过程。

获得 BpServiceManager 对象后，instantiate()函数会调用 IServiceManager::addService (String16("media.camera"), new CameraService())。

我们看到传递给 IServiceManager::addService 的参数是一个 CameraService 实例化地址，而 CameraService 继成于 BnCameraService，BnCameraService 继成于 BnInterface，BnInterface 继成于 BBinder，BBinder 继成于 IBinder。

1、IServiceManager::addService 将毁调用到 BpServiceManager::addService。因为 IServiceManager::addService 是纯虚函数。

```
@IServiceManager.cpp
virtual status_t addService(const String16& name, const sp<IBinder>&
                            service)
{
    Parcel data, reply;
    data.writeInterfaceToken(IServiceManager::getInterfaceDescriptor(
                            ));
    data.writeString16(name);
    data.writeStrongBinder(service);
    status_t err = remote()->transact(ADD_SERVICE_TRANSACTION, data,
                                      &reply);
    return err == NO_ERROR ? reply.readInt32() : err;
}
```

2、首先注意 data.writeStrongBinder(service)

```
@Parcel.cpp
status_t Parcel::writeStrongBinder(const sp<IBinder>& val
{
    return flatten_binder(ProcessState::self(), val, this);
}
status_t flatten_binder(const sp<ProcessState>& proc,
    const sp<IBinder>& binder, Parcel* out)
{
   flat_binder_object obj;

    obj.flags = 0x7f | FLAT_BINDER_FLAG_ACCEPTS_FDS;
    if (binder != NULL) {
        IBinder *local = binder->localBinder();
        if (!local) {
            BpBinder *proxy = binder->remoteBinder();
            if (proxy == NULL) {
                LOGE("null proxy");
            }
            const int32_t handle = proxy ? proxy->handle() : 0;
            obj.type = BINDER_TYPE_HANDLE;
            obj.handle = handle;
            obj.cookie = NULL;
        } else {//我们知道传进的是一个 CameraService 对象，所以会走到这个分支来
            obj.type = BINDER_TYPE_BINDER;
```

```
            //在 binder_transaction()函数中会检查这个类型， 并生成这个 binder
对应的 handle，系统有了 handle 后，通过 readStrongBinder()à unflatten_binder()
就可以生成对应的 BpBinder 对象。
            obj.binder = local->getWeakRefs();
            obj.cookie = local;
        }
    } else {
        obj.type = BINDER_TYPE_BINDER;
        obj.binder = NULL;
        obj.cookie = NULL;
    }
    return finish_flatten_binder(binder, obj, out);
}
```

3、其次注意这 里的 remote()将返回我们前面得 BpBinder 对象，我们还记得 intr = new BpServiceManager(obj);

```
@IServiceManager.cpp
    而 BpServiceManager(const sp<IBinder>& impl)
        : BpInterface<IServiceManager>(impl)//基类 BpInterface
    {
    }
@IInterface.h
template<typename INTERFACE>
inline BpInterface<INTERFACE>::BpInterface(const sp<IBinder>& remote)
    : BpRefBase(remote)  //基类 BpRefBase
{
}
@ Binder.cpp
BpRefBase::BpRefBase(const sp<IBinder>& o)
    : mRemote(o.get()), mRefs(NULL), mState(0)
{
    extendObjectLifetime(OBJECT_LIFETIME_WEAK);
    if (mRemote) {
        mRemote->incStrong(this);  // Removed on first IncStrong().
        mRefs = mRemote->createWeak(this);  // Held for our entire lifetime.
    }
}
@include/binder/Binder.h
class BpRefBase : public virtual RefBase
{
protected:
    BpRefBase(const sp<IBinder>& o);
    virtual                 ~BpRefBase();
```

```
    virtual void            onFirstRef();
    virtual void            onLastStrongRef(const void* id);
    virtual bool            onIncStrongAttempted(uint32_t flags, const
                                                 void* id);
    inline  IBinder*        remote()                { return mRemote; }
  ...。
```

```
@BpBinder.cpp
status_t BpBinder::transact(
    uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags)
 {
          status_t status = IPCThreadState::self()->transact(
            mHandle, code, data, reply, flags);
  }
```

4、mHandle 为0，BpBinder 往下调用 IPCThreadState:transact 函数将数据发给与 mHandle 相关联的 Service Manager Process。

```
status_t IPCThreadState::transact(int32_t handle,
                                  uint32_t code, const Parcel& data,
                                  Parcel* reply, uint32_t flags)
{
   ....
```

```
err = writeTransactionData(BC_TRANSACTION, flags, handle, code, data,
                           NULL);
    if (reply) {
            err = waitForResponse(reply);
        } else {
            Parcel fakeReply;
            err = waitForResponse(&fakeReply);
        }
       ....
   }
```

5、IPCThreadState::transact 首先调用 writeTransactionData 函数为 binder 内核驱动构造一个 transaction 结构。

```
@IPCThreadState.cpp
status_t IPCThreadState::writeTransactionData(int32_t cmd,
uint32_t binderFlags, int32_t handle, uint32_t code, const Parcel&
data, status_t* statusBuffer)
{
  binder_transaction_data tr;
  tr.target.handle = handle;
```

```
  tr.code = code;
  tr.flags = binderFlags;

  const status_t err = data.errorCheck();
  if (err == NO_ERROR) {
    tr.data_size = data.ipcDataSize();
    tr.data.ptr.buffer = data.ipcData();
    tr.offsets_size = data.ipcObjectsCount()*sizeof(size_t);
    tr.data.ptr.offsets = data.ipcObjects();
  }
 …
  mOut.writeInt32(cmd);
  mOut.write(&tr, sizeof(tr));
}
```

6、waitForResponse 将调用 talkWithDriver 与对 Binder kernel 进行读写操作。当 Binder kernel 接收到数据后，service_mananger 线程的 ThreadPool 就会启动。

```
@IPCThreadState.cpp
status_t IPCThreadState::waitForResponse(Parcel *reply, status_t *acquireResult)
{
      while (1) {
        if ((err=talkWithDriver()) < NO_ERROR) break;

.....
}

@IPCThreadState.cpp
status_t IPCThreadState::talkWithDriver(bool doReceive)
{
   ....
        if (ioctl(mProcess->mDriverFD, BINDER_WRITE_READ, &bwr) >= 0)
            err = NO_ERROR;
    ....
}
```

7、系统分析到这里，事务数据已经发送到了 binder 驱动，下面将讨论 binder 驱动的相关处理过程。

```
@ drivers/misc/binder.c
static int binder_open(struct inode *nodp, struct file *filp)
{
  struct binder_proc *proc;
  if (binder_debug_mask & BINDER_DEBUG_OPEN_CLOSE)
```

```
    proc = kzalloc(sizeof(*proc), GFP_KERNEL);
    get_task_struct(current);
  proc->tsk = current;
  INIT_LIST_HEAD(&proc->todo);
  init_waitqueue_head(&proc->wait);
  proc->default_priority = task_nice(current);
  mutex_lock(&binder_lock);
  binder_stats.obj_created[BINDER_STAT_PROC]++;
  hlist_add_head(&proc->proc_node, &binder_procs);
  proc->pid = current->group_leader->pid;
  INIT_LIST_HEAD(&proc->delivered_death);
  filp->private_data = proc;
  mutex_unlock(&binder_lock);
  if (binder_proc_dir_entry_proc) {
    char strbuf[11];
    snprintf(strbuf, sizeof(strbuf), "%u", proc->pid);
    create_proc_read_entry(strbuf, S_IRUGO, binder_proc_dir_entry_proc,
                           binder_read_proc_proc, proc);
  }
  return 0;
}
```

8、当任何进程打开/dev/binder 驱动时，会分配相应的 binder_proc 结构（给进程）。因此当进行 ioctl 调用时，binder 系统通过 binder_proc 结构知道应该和那个进程交 换数据了

Client 与 Binder Driver 交互过程:

```
@ drivers/misc/binder.c
static long binder_ioctl(struct file *filp, unsigned int cmd, unsigned
                         long arg)
{
thread = binder_get_thread(proc);  //根据 当前 caller 进程 消息获取该进程线程
                                     池数据结构
......
  switch (cmd) {
  case BINDER_WRITE_READ: {
    struct binder_write_read bwr;
    if (size != sizeof(struct binder_write_read)) {
      ret = -EINVAL;
      goto err;
    }
    if (copy_from_user(&bwr, ubuf, sizeof(bwr))) {
      ret = -EFAULT;
      goto err;
```

```
    }
    if (bwr.write_size > 0) {
      ret = binder_thread_write(proc, thread, (void __user
            *)bwr.write_buffer, bwr.write_size, &bwr.write_consumed);
      if (ret < 0) {
        bwr.read_consumed = 0;
        if (copy_to_user(ubuf, &bwr, sizeof(bwr)))
          ret = -EFAULT;
      goto err;
      }
    }
    if (bwr.read_size > 0) {
      ret = binder_thread_read(proc, thread, (void __user
            *)bwr.read_buffer, bwr.read_size, &bwr.read_consumed,
            filp->f_flags & O_NONBLOCK);
      if (!list_empty(&proc->todo))
        wake_up_interruptible(&proc->wait);// 恢复挂起的 caller 进程
      if (ret < 0) {
        if (copy_to_user(ubuf, &bwr, sizeof(bwr)))
          ret = -EFAULT;
        goto err;
      }
    }
    if (copy_to_user(ubuf, &bwr, sizeof(bwr))) {
      ret = -EFAULT;
      goto err;
    }
    break;
  }
  ......
}
```

9、驱动首先处理写，然后读。让我们先看看 binder_thread_write()函数。该函数的核心是一个从写缓冲中解析命令并执行该命令。

```
@ drivers/misc/binder.c
Int binder_thread_write(struct binder_proc *proc, struct binder_thread
                  *thread,void __user *buffer, int size, signed long
                  *consumed)
{
  uint32_t cmd;
  void __user *ptr = buffer + *consumed;
  void __user *end = buffer + size;
  while (ptr < end && thread->return_error == BR_OK) {
```

```
    if (get_user(cmd, (uint32_t __user *)ptr))
      return -EFAULT;
    ptr += sizeof(uint32_t);
    if (_IOC_NR(cmd) < ARRAY_SIZE(binder_stats.bc)) {
      binder_stats.bc[_IOC_NR(cmd)]++;
      proc->stats.bc[_IOC_NR(cmd)]++;
      thread->stats.bc[_IOC_NR(cmd)]++;
    }
    switch (cmd) {
    case BC_INCREFS:
       …
    case BC_TRANSACTION://IPCThreadState 通过 writeTransactionData()设置
                        该 cmd
      case BC_REPLY: {
      struct binder_transaction_data tr;
      if (copy_from_user(&tr, ptr, sizeof(tr)))
     return -EFAULT;
    ptr += sizeof(tr);
    binder_transaction(proc, thread, &tr, cmd == BC_REPLY);
    break;
    default:
      …
    }
    *consumed = ptr - buffer;
  }
  return 0;
}
```

10、在 binder_transaction ()中，首先设置 target_node，target_proc 和 target_thread，并将请求放入链表，并唤 醒 binder_thread_read()中的等待线程。

```
@ drivers/misc/binder.c
static void binder_transaction(struct binder_proc *proc, struct
             binder_thread *thread, struct binder_transaction_data
             *tr, int reply)
      target_node = binder_context_mgr_node;
e->to_node = target_node->debug_id;
      target_proc = target_node->proc;
   …
struct binder_transaction *tmp;
  tmp = thread->transaction_stack;
  while (tmp) {
    if (tmp->from && tmp->from->proc == target_proc)
      target_thread = tmp->from;
```

```
    tmp = tmp->from_parent;
  }
…
switch (fp->type) {
            case BINDER_TYPE_BINDER:// writeStrongBinder()
                                    flatten_binder() 会设置这个 type
            case BINDER_TYPE_WEAK_BINDER: {
            struct binder_ref *ref;
                    struct binder_node *node = binder_get_node(proc,
                                               fp->binder);
                    if (node == NULL) {
                            node = binder_new_node(proc, fp->binder,
                                                   fp->cookie);
            …
                    }

                    ref = binder_get_ref_for_node(target_proc, node);

                    if (fp->type == BINDER_TYPE_BINDER)
                            fp->type = BINDER_TYPE_HANDLE;
                    else
                            fp->type = BINDER_TYPE_WEAK_HANDLE;
                    fp->handle = ref->desc;//生成 handle,以后通过他创建
                                           BpBinder

            } break;
......
  t->work.type = BINDER_WORK_TRANSACTION;
  list_add_tail(&t->work.entry, target_list);
  tcomplete->type = BINDER_WORK_TRANSACTION_COMPLETE;
  list_add_tail(&tcomplete->entry, &thread->todo);
  if (target_wait)
    wake_up_interruptible(target_wait);//唤醒服务器中的线程
  return;
  ......
}
```

11、这样数据就写入 binder Driver 中，service_manager 的读将获得数据从而 binder_thread_read()将数据读给 service_manager。

```
@ drivers/misc/binder.c
static int binder_thread_read(struct binder_proc *proc, struct
binder_thread *thread,
void __user *buffer, int size, signed long *consumed, int non_block)
```

```
{
 ......
```

12、下面的语句将 client 的写缓冲区的数据拷贝到 service_manager 的读缓冲区。

```
tr.data_size = t->buffer->data_size;
tr.offsets_size = t->buffer->offsets_size;
tr.data.ptr.buffer = (void *)((void *)t->buffer->data +
                              proc->user_buffer_offset);
tr.data.ptr.offsets = tr.data.ptr.buffer +
            ALIGN(t->buffer->data_size, sizeof(void *));
if (put_user(cmd, (uint32_t __user *)ptr))
  return -EFAULT;
ptr += sizeof(uint32_t);
if (copy_to_user(ptr, &tr, sizeof(tr)))
  return -EFAULT;
ptr += sizeof(tr);
…
}
```

13、到现在为止，service_manager 已经得到了一个从 client 发送过来的 BR_TRANSACTION 类型的数据包，service_manager 将通过 binder_parse()来分析这些数据包并作相应处理. 最 后通过 binder_send_reply()返回处理结果。

binder_loop()函数

```
@Service_manager.c
void binder_loop(struct binder_state *bs, binder_handler func)
{
   …
    binder_write(bs, readbuf, sizeof(unsigned));
for (;;) {
     bwr.read_size = sizeof(readbuf);
     bwr.read_consumed = 0;
     bwr.read_buffer = (unsigned) readbuf;
     res = ioctl(bs->fd, BINDER_WRITE_READ, &bwr);
     //如果没有要处理的请求进程将挂起
     if (res < 0) {
         LOGE("binder_loop: ioctl failed (%s)\n", strerror(errno));
         break;
     }
     res = binder_parse(bs, 0, readbuf, bwr.read_consumed, func);
     //这里 func 就是
     //svcmgr_handler
      …
```

```
    }
}
```

```
@ frameworks/base/cmds/servicemanager/binder.c
int binder_parse(struct binder_state *bs, struct binder_io *bio,
                 uint32_t *ptr, uint32_t size, binder_handler func)
{
  ......
  case BR_TRANSACTION: {
    struct binder_txn *txn = (void *) ptr;
    binder_dump_txn(txn);
    if (func) {
      unsigned rdata[256/4];
      struct binder_io msg;
      struct binder_io reply;
      int res;
      bio_init(&reply, rdata, sizeof(rdata), 4);
      bio_init_from_txn(&msg, txn);
      res = func(bs, txn, &msg, &reply);  // 即 svcmgr_handler()
      binder_send_reply(bs, &reply, txn->data, res);
    }
    ptr += sizeof(*txn) / sizeof(uint32_t);
    break;
  }
  ......
}
```

14、binder_parse 会调用 svcmgr_handler（也就是参数 func），按照 BpServerManager 相反的步骤处理 BR_TRANSACTION 数据包。这里 binder_txn 结构实际上与 binder_transaction_data 结构是一样的。在本文的例 子中，事务码（transaction code）为 SVC_MGR_ADD_SERVICE。

```
@/frameworks/base/cmds/servicemanager/binder.c
int svcmgr_handler(struct binder_state *bs,
                   struct binder_txn *txn,
                   struct binder_io *msg,
                   struct binder_io *reply)
{
  …
  s = bio_get_string16(msg, &len);
  …
  switch(txn->code) {
  case SVC_MGR_GET_SERVICE:
  case SVC_MGR_CHECK_SERVICE:
    s = bio_get_string16(msg, &len);
```

```
    ptr = do_find_service(bs, s, len);
     bio_put_ref(reply, ptr);
    return 0;
  case SVC_MGR_ADD_SERVICE:
    s = bio_get_string16(msg, &len);
    ptr = bio_get_ref(msg);
    if (do_add_service(bs, s, len, ptr, txn->sender_euid))
      return -1;
    break;

  default:
    return -1;
  }
  bio_put_uint32(reply, 0);
  return 0;
}
```

15、于是 service_manager 通过 bio_get_ref()获取到这个服务的相关信息。并进行注册处理。

```
@frameworks/base/cmds/servicemanager/binder.c
void *bio_get_ref(struct binder_io *bio)
{
  struct binder_object *obj;
  obj = _bio_get_obj(bio);
  if (obj->type == BINDER_TYPE_HANDLE)
    return obj->pointer;
  return 0;
}
```

svcmgr_handler()返回后将通过 binder_send_reply()将反馈结果给 client 的 Binder driver, 最后返回到 IPCThreadState::waitForResponse(),并退回到 IPCThreadState::transact(),再退回到 BpBinder::transact(),再退回到 BpServiceManager ::addService(),CameraService::instantiate(), 最后退回到 main()@main_mediaserver.cpp,从而完成 CameraService 的注册。

5.2 ramdisk driver 驱动实现的源码。

以下是源码内容：

```
/*
 * ramdisk.c - Multiple RAM disk driver - gzip-loading version - v. 0.8 beta.
 *  Cosmetic changes in #ifdef MODULE, code movement, etc.
 *  When the RAM disk module is removed, free the protected buffers
 *  Default RAM disk size changed to 2.88 MB
```

```
 *
 */

#include <linux/config.h>
#include <linux/string.h>
#include <linux/slab.h>
#include <asm/atomic.h>
#include <linux/bio.h>
#include <linux/module.h>
#include <linux/moduleparam.h>
#include <linux/init.h>
#include <linux/devfs_fs_kernel.h>
#include <linux/pagemap.h>
#include <linux/blkdev.h>
#include <linux/genhd.h>
#include <linux/buffer_head.h>     /* for invalidate_bdev() */
#include <linux/backing-dev.h>
#include <linux/blkpg.h>
#include <linux/writeback.h>

#include <asm/uaccess.h>

/* Various static variables go here.  Most are used only in the RAM disk code.
 */

static struct gendisk *rd_disks[CONFIG_BLK_DEV_RAM_COUNT];
static struct block_device *rd_bdev[CONFIG_BLK_DEV_RAM_COUNT];/* Protected
                                                              device data */
static struct request_queue *rd_queue[CONFIG_BLK_DEV_RAM_COUNT];

/*
 * Parameters for the boot-loading of the RAM disk.  These are set by
 * init/main.c (from arguments to the kernel command line) or from the
 * architecture-specific setup routine (from the stored boot sector
 * information).
 */
int rd_size = CONFIG_BLK_DEV_RAM_SIZE;    /* Size of the RAM disks */
/*
 * It would be very desirable to have a soft-blocksize (that in the case
 * of the ramdisk driver is also the hardblocksize ;) of PAGE_SIZE because
 * doing that we'll achieve a far better MM footprint. Using a rd_blocksize
  of
 * BLOCK_SIZE in the worst case we'll make PAGE_SIZE/BLOCK_SIZE buffer-pages
```

```
 * unfreeable. With a rd_blocksize of PAGE_SIZE instead we are sure that only
 * 1 page will be protected. Depending on the size of the ramdisk you
 * may want to change the ramdisk blocksize to achieve a better or worse MM
 * behaviour. The default is still BLOCK_SIZE (needed by rd_load_image that
 * supposes the filesystem in the image uses a BLOCK_SIZE blocksize).
 */
static int rd_blocksize = BLOCK_SIZE;     /* blocksize of the RAM disks */

/*
 * Copyright (C) 2000 Linus Torvalds.
 *             2000 Transmeta Corp.
 * aops copied from ramfs.
 */

/*
 * If a ramdisk page has buffers, some may be uptodate and some may be not.
 * To bring the page uptodate we zero out the non-uptodate buffers.  The
 * page must be locked.
 */
static void make_page_uptodate(struct page *page)
{
    if (page_has_buffers(page)) {
        struct buffer_head *bh = page_buffers(page);
        struct buffer_head *head = bh;

        do {
            if (!buffer_uptodate(bh)) {
                memset(bh->b_data, 0, bh->b_size);
                /*
                 * akpm: I'm totally undecided about this.  The
                 * buffer has just been magically brought "up to
                 * date", but nobody should want to be reading
                 * it anyway, because it hasn't been used for
                 * anything yet.  It is still in a "not read
                 * from disk yet" state.
                 *
                 * But non-uptodate buffers against an uptodate
                 * page are against the rules.  So do it anyway.
                 */
                set_buffer_uptodate(bh);
            }
        } while ((bh = bh->b_this_page) != head);
    } else {
```

```
        memset(page_address(page), 0, PAGE_CACHE_SIZE);
    }
    flush_dcache_page(page);
    SetPageUptodate(page);
}

static int ramdisk_readpage(struct file *file, struct page *page)
{
    if (!PageUptodate(page))
        make_page_uptodate(page);
    unlock_page(page);
    return 0;
}

static int ramdisk_prepare_write(struct file *file, struct page *page,
                                 unsigned offset, unsigned to)
{
    if (!PageUptodate(page))
        make_page_uptodate(page);
    return 0;
}

static int ramdisk_commit_write(struct file *file, struct page *page,
                                 unsigned offset, unsigned to)
{
    set_page_dirty(page);
    return 0;
}

/*
 * ->writepage to the the blockdev's mapping has to redirty the page so that
    the
 * VM doesn't go and steal it.  We return AOP_WRITEPAGE_ACTIVATE so that the
    VM
 * won't try to (pointlessly) write the page again for a while.
 *
 * Really, these pages should not be on the LRU at all.
 */
static int ramdisk_writepage(struct page *page, struct writeback_control
                             *wbc)
{
    if (!PageUptodate(page))
        make_page_uptodate(page);
```

```
    SetPageDirty(page);
    if (wbc->for_reclaim)
        return AOP_WRITEPAGE_ACTIVATE;
    unlock_page(page);
    return 0;
}

/*
 * This is a little speedup thing: short-circuit attempts to write back the
 * ramdisk blockdev inode to its non-existent backing store.
 */
static int ramdisk_writepages(struct address_space *mapping,
                              struct writeback_control *wbc)
{
    return 0;
}

/*
 * ramdisk blockdev pages have their own ->set_page_dirty() because we don't
 * want them to contribute to dirty memory accounting.
 */
static int ramdisk_set_page_dirty(struct page *page)
{
    SetPageDirty(page);
    return 0;
}

static struct address_space_operations ramdisk_aops = {
    .readpage = ramdisk_readpage,
    .prepare_write  = ramdisk_prepare_write,
    .commit_write   = ramdisk_commit_write,
    .writepage   = ramdisk_writepage,
    .set_page_dirty = ramdisk_set_page_dirty,
    .writepages  = ramdisk_writepages,
};

    static int rd_blkdev_pagecache_IO(int rw, struct bio_vec *vec, sector_t
                                      sector,struct address_space *mapping)
{
    pgoff_t index = sector >> (PAGE_CACHE_SHIFT - 9);
    unsigned int vec_offset = vec->bv_offset;
    int offset = (sector << 9) & ~PAGE_CACHE_MASK;
    int size = vec->bv_len;
```

```
    int err = 0;

    do {
        int count;
        struct page *page;
        char *src;
        char *dst;

        count = PAGE_CACHE_SIZE - offset;
        if (count > size)
            count = size;
        size -= count;

        page = grab_cache_page(mapping, index);
        if (!page) {
            err = -ENOMEM;
            goto out;
        }

        if (!PageUptodate(page))
            make_page_uptodate(page);

        index++;

        if (rw == READ) {
            src = kmap_atomic(page, KM_USER0) + offset;
            dst = kmap_atomic(vec->bv_page, KM_USER1) + vec_offset;
        } else {
            src = kmap_atomic(vec->bv_page, KM_USER0) + vec_offset;
            dst = kmap_atomic(page, KM_USER1) + offset;
        }
        offset = 0;
        vec_offset += count;

        memcpy(dst, src, count);

        kunmap_atomic(src, KM_USER0);
        kunmap_atomic(dst, KM_USER1);

        if (rw == READ)
            flush_dcache_page(vec->bv_page);
        else
            set_page_dirty(page);
```

```
        unlock_page(page);
        put_page(page);
    } while (size);

 out:
    return err;
}

/*
 *  Basically, my strategy here is to set up a buffer-head which can't be
 *  deleted, and make that my Ramdisk.  If the request is outside of the
 *  allocated size, we must get rid of it...
 *
 * 19-JAN-1998  Richard Gooch <rgooch@atnf.csiro.au>  Added devfs support
 *
 */
static int rd_make_request(request_queue_t *q, struct bio *bio)
{
    struct block_device *bdev = bio->bi_bdev;
    struct address_space * mapping = bdev->bd_inode->i_mapping;
    sector_t sector = bio->bi_sector;
    unsigned long len = bio->bi_size >> 9;
    int rw = bio_data_dir(bio);
    struct bio_vec *bvec;
    int ret = 0, i;

    if (sector + len > get_capacity(bdev->bd_disk))
        goto fail;

    if (rw==READA)
        rw=READ;

    bio_for_each_segment(bvec, bio, i) {
        ret |= rd_blkdev_pagecache_IO(rw, bvec, sector, mapping);
        sector += bvec->bv_len >> 9;
    }
    if (ret)
        goto fail;

    bio_endio(bio, bio->bi_size, 0);
    return 0;
fail:
    bio_io_error(bio, bio->bi_size);
```

```
    return 0;
}

static int rd_ioctl(struct inode *inode, struct file *file,
                    unsigned int cmd, unsigned long arg)
{
    int error;
    struct block_device *bdev = inode->i_bdev;
    if (cmd != BLKFLSBUF)
        return -ENOTTY;

    /*
     * special: we want to release the ramdisk memory, it's not like with
     * the other blockdevices where this ioctl only flushes away the buffer
     * cache
     */
    error = -EBUSY;
    down(&bdev->bd_sem);
    if (bdev->bd_openers <= 2) {
        truncate_inode_pages(bdev->bd_inode->i_mapping, 0);
        error = 0;
    }
    up(&bdev->bd_sem);
    return error;
}

/*
 * This is the backing_dev_info for the blockdev inode itself.  It doesn't
    need
 * writeback and it does not contribute to dirty memory accounting.
 */
static struct backing_dev_info rd_backing_dev_info = {
    .ra_pages = 0,   /* No readahead */
    .capabilities   = BDI_CAP_NO_ACCT_DIRTY | BDI_CAP_NO_WRITEBACK |
                      BDI_CAP_MAP_COPY,
    .unplug_io_fn   = default_unplug_io_fn,
};

/*
 * This is the backing_dev_info for the files which live atop the ramdisk
 * "device".  These files do need writeback and they do contribute to dirty
 * memory accounting.
 */
```

```
static struct backing_dev_info rd_file_backing_dev_info = {
    .ra_pages = 0,   /* No readahead */
    .capabilities   = BDI_CAP_MAP_COPY,/* Does contribute to dirty memory */
    .unplug_io_fn   = default_unplug_io_fn,
};

static int rd_open(struct inode *inode, struct file *filp)
{
    unsigned unit = iminor(inode);

    if (rd_bdev[unit] == NULL) {
        struct block_device *bdev = inode->i_bdev;
        struct address_space *mapping;
        unsigned bsize;
        gfp_t gfp_mask;

        inode = igrab(bdev->bd_inode);
        rd_bdev[unit] = bdev;
        bdev->bd_openers++;
        bsize = bdev_hardsect_size(bdev);
        bdev->bd_block_size = bsize;
        inode->i_blkbits = blksize_bits(bsize);
        inode->i_size = get_capacity(bdev->bd_disk)<<9;

        mapping = inode->i_mapping;
        mapping->a_ops = &ramdisk_aops;
        mapping->backing_dev_info = &rd_backing_dev_info;
        bdev->bd_inode_backing_dev_info = &rd_file_backing_dev_info;

        /*
         * Deep badness.  rd_blkdev_pagecache_IO() needs to allocate
         * pagecache pages within a request_fn.  We cannot recur back
         * into the filesytem which is mounted atop the ramdisk, because
         * that would deadlock on fs locks.  And we really don't want
         * to reenter rd_blkdev_pagecache_IO when we're already within
         * that function.
         *
         * So we turn off __GFP_FS and __GFP_IO.
         *
         * And to give this thing a hope of working, turn on __GFP_HIGH.
         * Hopefully, there's enough regular memory allocation going on
         * for the page allocator emergency pools to keep the ramdisk
         * driver happy.
```

```
         */
        gfp_mask = mapping_gfp_mask(mapping);
        gfp_mask &= ~(__GFP_FS|__GFP_IO);
        gfp_mask |= __GFP_HIGH;
        mapping_set_gfp_mask(mapping, gfp_mask);
    }

    return 0;
}

static struct block_device_operations rd_bd_op = {
    .owner =  THIS_MODULE,
    .open =      rd_open,
    .ioctl =  rd_ioctl,
};

/*
 * Before freeing the module, invalidate all of the protected buffers!
 */
static void __exit rd_cleanup(void)
{
    int i;

    for (i = 0; i < CONFIG_BLK_DEV_RAM_COUNT; i++) {
        struct block_device *bdev = rd_bdev[i];
        rd_bdev[i] = NULL;
        if (bdev) {
            invalidate_bdev(bdev, 1);
            blkdev_put(bdev);
        }
        del_gendisk(rd_disks[i]);
        put_disk(rd_disks[i]);
        blk_cleanup_queue(rd_queue[i]);
    }
    devfs_remove("rd");
    unregister_blkdev(RAMDISK_MAJOR, "ramdisk");
}

/*
 * This is the registration and initialization section of the RAM disk driver
 */
static int __init rd_init(void)
{
```

```
    int i;
    int err = -ENOMEM;

    if (rd_blocksize > PAGE_SIZE || rd_blocksize < 512 ||
           (rd_blocksize & (rd_blocksize-1))) {
       printk("RAMDISK: wrong blocksize %d, reverting to defaults\n",
             rd_blocksize);
       rd_blocksize = BLOCK_SIZE;
    }

    for (i = 0; i < CONFIG_BLK_DEV_RAM_COUNT; i++) {
       rd_disks[i] = alloc_disk(1);
       if (!rd_disks[i])
          goto out;
    }

    if (register_blkdev(RAMDISK_MAJOR, "ramdisk")) {
       err = -EIO;
       goto out;
    }

    devfs_mk_dir("rd");

    for (i = 0; i < CONFIG_BLK_DEV_RAM_COUNT; i++) {
       struct gendisk *disk = rd_disks[i];

       rd_queue[i] = blk_alloc_queue(GFP_KERNEL);
       if (!rd_queue[i])
          goto out_queue;

       blk_queue_make_request(rd_queue[i], &rd_make_request);
       blk_queue_hardsect_size(rd_queue[i], rd_blocksize);

       /* rd_size is given in kB */
       disk->major = RAMDISK_MAJOR;
       disk->first_minor = i;
       disk->fops = &rd_bd_op;
       disk->queue = rd_queue[i];
       disk->flags |= GENHD_FL_SUPPRESS_PARTITION_INFO;
       sprintf(disk->disk_name, "ram%d", i);
       sprintf(disk->devfs_name, "rd/%d", i);
       set_capacity(disk, rd_size * 2);
       add_disk(rd_disks[i]);
```

```
    }

    /* rd_size is given in kB */
    printk("RAMDISK driver initialized: "
        "%d RAM disks of %dK size %d blocksize\n",
        CONFIG_BLK_DEV_RAM_COUNT, rd_size, rd_blocksize);

    return 0;
out_queue:
    unregister_blkdev(RAMDISK_MAJOR, "ramdisk");
out:
    while (i--) {
        put_disk(rd_disks[i]);
        blk_cleanup_queue(rd_queue[i]);
    }
    return err;
}

module_init(rd_init);
module_exit(rd_cleanup);

/* options - nonmodular */
#ifndef MODULE
static int __init ramdisk_size(char *str)
{
    rd_size = simple_strtol(str,NULL,0);
    return 1;
}
static int __init ramdisk_size2(char *str)   /* kludge */
{
    return ramdisk_size(str);
}
static int __init ramdisk_blocksize(char *str)
{
    rd_blocksize = simple_strtol(str,NULL,0);
    return 1;
}
__setup("ramdisk=", ramdisk_size);
__setup("ramdisk_size=", ramdisk_size2);
__setup("ramdisk_blocksize=", ramdisk_blocksize);
#endif

/* options - modular */
```

```
module_param(rd_size, int, 0);
MODULE_PARM_DESC(rd_size, "Size of each RAM disk in kbytes.");
module_param(rd_blocksize, int, 0);
MODULE_PARM_DESC(rd_blocksize, "Blocksize of each RAM disk in bytes.");
MODULE_ALIAS_BLOCKDEV_MAJOR(RAMDISK_MAJOR);
MODULE_LICENSE("GPL");
```

【其他】

6.1 提交 BUG

如果你发现文档中有不妥的地方，请发邮件至 eoeandroid@eoemobile.com 进行反馈，我们会定期更新、发布更新后的版本。

6.2 关于 eoeAndroid

eoeAndroid 是国内成立最早，规模最大的 Android 开发者社区，拥有海量的 Android 学习资料。分享、互助的氛围，让 Android 开发者迅速成长，逐步从懵懂到了解，从入门到开发出属于自己的应用，eoeAndroid 为广大 Android 开发者奠定坚实的技术基础。从初级到高级，从环境搭建到底层架构，eoeAndroid 社区为开发者精挑细选了丰富的学习资料和实例代码。让 Android 开发者在社区中迅速的成长，并在此基础上开发出更多优秀的 Android 应用。

6.3 【北京站】eoe 携手支付宝移动应用开发者沙龙，免费报名参加！

支付宝联合 eoe，将全国各大城市举办开发者沙龙，探讨移动互联网市场的未来。届时我们将邀请到来自支付宝、eoe、淘宝等专业领域的技术专家和热心网友，共同分析技术趋势并现场回答朋友们感兴趣的问题。

2011年4月24日，“移动应用发展新机遇，2011支付宝携手 eoe 移动开发者沙龙大会”北京站，现在开始报名。我们欢迎各位同学踊跃报名，免费参加，到场的每位开发者都将有机会得到智能手机、电脑包，还有个性精美 U 盘等着你拿！

详情点击：http://www.eoeandroid.com/thread-69217-1-1.html

6.4 4月9日，eoe Android 移动互联高峰论坛在深圳举行

如今，我们不得不惊叹 Android 的发展速度，而究其背后，是移动互联网的发展壮景。2011年4月9日下午，由 eoe 主办的，在深圳会展中心召开的 eoe Android 移动互联高峰论坛，正是以 Android 为主题而进行一场关于移动互联网讨论的大会。移动互联网各个环节的参与者都在会上提出了自己的观点。

Google、支付宝、淘宝等代表受邀出席了此次会议并发表演讲。众多国内优秀应用开发团队在本次高峰论坛上展示了自己的应用，这是 eoe 针对国内开发者做出的一项福利工作，旨在通过这个环节，帮助合作伙伴推广自己的应用。帮助开发者，服务好合作伙伴一直的 eoe 的使命。

详情点击：http://www.eoeandroid.com/thread-68634-1-1.html

北京易联致远无限技术有限公司

责任编辑：莫言默语

美术支持：阿彦　　**技术支持：**gaotong86

中国最大的 Android 开发者社区　：www.eoeandroid.com

中国本土的 Android 软件下载平台：www.eoemarket.com