华清远见深圳中心

项目设计说明书

华清远见深圳中心 0911 期 学员 李永亮（125781030@qq.com） 2010-4-12

项目名称 Android1.5 移植

项目编号 V1.0

文件编号

提　　交 李永亮

拟　　制 2010 年 4 月 12 日

审　　核 范一航

批　　准 易松华

Android 移植

【摘要】

本项目的主要目的是将 Android1.5 移植到 Omap3530 的开发板上，在修改和添加相应代码后，使 Android1.5 系统能够在 Omap3530 上实现：接打电话、收发短信、GPRS 上网、Wifi 上网、音视频的播放，Camera 预览及 GPS 导航等功能。

通过移植达到深入了解 Android 的目的。

【关键字】

Android、中间件，OSS、ALSA、video4linux、video4linux2、bmp、peg 解码、HAL

第一章 绪 论

项目介绍

这个项目的在 Google 的 Android1.5(cupcake)系统移植到 Omap3530 开发板上，利用相关技术使其能够正常运行，并且能够接打电话、收发短信、GPRS 上网、Wifi 上网、音视频的播放，Camera 预览及 GPS 导航等功能。

使用的软件平台：**cupcake1.5r2**, **ubuntu9.10**, **Linux2.6.29 内核**等。

使用的硬件：**Omap3530，中星微摄像头，GSM 模块、wifi 模块，GPS 模块，耳机**等；

1 Android

Android 是一个较新的系统技术，因此很多人还不是很了解这是一个什么概念，因此我们得先了解一下 Android，了解什么是 Android 和 Android 应用在什么方面。

1.1 什么是 Android

Android 是基于 Linux 内核的软件平台和操作系统，是 Google 在 2007 年 11 月 5 日公布的手机系统平台，早期由 Google 开发，后由开放手机联盟（Open Handset Alliance）开发。它采用了软件堆层（software stack，又名以软件叠层）的架构，主要分为三部分。低层以 Linux 内核工作为基础，只提供基本功能；其他的应用软件则由各公司自行开发，以 Java 作为编写程序的一部分。另外，为了推广此技术，Google 和其它几十个手机公司建立了开放手机联盟。Android 在未公开之前常被传闻为 Google 电话或 gPhone

Android 是运行于 Linux kernel 之上，但并不是 GNU Linux。因为在一般 GNU/Linux 里的功能，Android 大都没有支持。

1.2 Android 的应用

Android 最初的定位是手机的软件平台和操作系统，但是近来很多商家已经推出 Android 系统的上网本及应用 Android 系统其它嵌入式产品。例如：明基、HTC、ASUS 等都将或已经发布了 Android 的上网本。

2 DevKit8000

DevKit8000 采用德州仪器（TI）OMAP3530 处理器作为主 CPU。OMAP3530 处理器集成了 600MHz 的 ARM Cortex™-A8 内核及 412MHz 的具有高级数字信号处理算法的 DSP 核，并提供了丰富的外设接口。DevKit8000 评估主板扩展出了网口、S-VIDEO 接口、音频

输入输出接口、USB OTG、USB HOST、SD/MMC 接口、串口、SPI 接口、IIC 接口、JTAG 接口、Camera 接口、TFT 屏接口、触摸屏接口、键盘接口及 HDMI 接口

3. Android 中间件

操作系统与应用程序的沟通桥梁，并用分为两层：函数层（Library）和虚拟机（Virtual Machine）。

Bionic 是 Android 改良 libc 的版本。Android 同时包含了 Webkit，所谓的 Webkit 就是 Apple Safari 浏览器背后的引擎。Surface flinger 是就 2D 或 3D 的内容显示到屏幕上。Android 使用工具链(Toolchain)为 Google 自制的 Bionic Libc。

Android 采用 OpenCORE 作为基础多媒体框架。OpenCORE 可分 7 大块：PVPlayer、PVAuthor、Codec、PacketVideo Multimedia Framework(PVMF)、Operating System Compatibility Library(OSCL)、Common、OpenMAX。

Android 使用 skia 为内核图形引擎，搭配 OpenGL/ES。skia 与 Linux Cairo 功能相当，但相较于 Linux Cairo, skia 功能还只是阳春型的。2005 年 Skia 公司被 Google 收购，2007 年初，Skia GL 源码被公开，目前 Skia 也是 Google Chrome 的图形引擎。

Android 的多媒体数据库采用 SQLite 数据库系统。数据库又分为共用数据库及私用数据库。用户可通过 ContentResolver 类（Column）取得共用数据库。

Android 的中间层多以 Java 实现，并且采用特殊的 Dalvik 虚拟机（Dalvik Virtual Machine）。Dalvik 虚拟机是一种“寄存器型态”（Register Based）的 Java 虚拟机，变量皆存放于寄存器中，虚拟机的指令相对减少。

Dalvik 虚拟机可以有多个 instance, 每个 Android 应用程序都用一个自属的 Dalvik 虚拟机来运行，让系统在运行程序时可达到优化。Dalvik 虚拟机并非运行 Java Bytecode，而是运行一种称为.dex 格式的文件

4. Android 硬件抽像层

Android HAL 是为了将 Android framework 与 Linux kernel 隔开，降低对 Linux kernel 的依赖，以达成 kernel independent。Android HAL 目前以 HAL stub 的形式存在，本身是.so 档，是一种 proxy 的概念。Android runtime 向 HAL 取得 stub 的 operations，再以 callback 的方式操作函数。

5. Linux 音频驱动

5.1 数字音频设备

音频编解码器是数字音频系统的核心，衡量它的主要指标如下：

1．采样频率：采样频率是每秒钟的采样次数，理论上采样频率越高，转换精度越高，目前主流的采样频率是 48kHz。

2．量化精度：是指对采样数据分析的精度，量化精度越高，声音就越逼真。

5.2 音频设备的硬件接口

1. PCM 接口：最简单的音频接口是 PCM(脉冲编码调制)接口，PCM 接口很容易实现，原则上能够支持任何数据方案和任何采样率，但需要每个音频通道获得一个独立的数据队列。
2. IIS 接口：IIS 更适合于立体声系统。
3. AC97 接口：AC97 不只是一种数据格式，用于音频编码的内部架构规格，它还具有控制功能。与具有分离控制接口的 IIS 方案相比，AC97 明显减少了整体管脚数。

5.3 Linux OSS 音频设备驱动

1. OSS 驱动的组成

   OSS 标准中有两个最基本的音频设备：mixer 和 dsp。

   需要指出的是，声卡采样频率是由内核中的驱动程序所决定的，而不取决于应用程序从声卡读取数据的速度。

   向 DSP 设备写入数据时，数字信号会经过 D/A 转换器编程模拟信号，然后产生声音。
2. mixer 接口

   注册一个混音器：register_sound_mixer()。
3. dsp 接口

   注册一个 dsp 设备：register_sound_dsp()。

   在数据从缓冲区复制到音频控制器的过程中，通常会使用 DMA，DMA 对声卡而言非常重要。

   在 OSS 驱动中，建立存放音频数据的环形缓冲区通常是值得推荐的方法。

5.4 Linux ALSA 音频设备驱动

1. ALSA 组成：

虽然 OSS 已经非常成熟，但它毕竟是一个没有完全开放源代码的商业产品，而 ALSA 恰好填补了这一空白。ALSA 的主要特点：支持多种声卡设备、模块化的内核驱动程序、支持 SMP 和多线程、提供应用开发函数库(alsa-lib)以简化应用程序开发、支持 OSS API，兼容 OSS 应用程序

2. card 和组件管理

对于每个声卡而言，必须创建一个 card 实例。card 是声卡的“总部”，它管理这个声卡上的所有设备。

3. PCM 设备

每个声卡最多可以有四个 PCM 实例，一个 PCM 实例对应一个设备文件。PCM 实例由 PCM 播放和录音流组成，而每个 PCM 流又由一个或多个 PCM 子流组成。

4. 控制接口

控制接口对于许多开关(switch)和调节器(slider)而言应用相当广泛，它能从用户空间被存取。control 的最主要的用途是 mixer，所有的 mixer 元素基于 control 内核 API 实现。

5. ALSA 用户空间编程

ALSA 驱动的声卡在用户空间不宜直接使用文件接口，而应使用 alsa-lib。

6. Platform 总线

从 Linux 2.6 起引入了一套新的驱动管理和注册机制:Platform_device 和 Platform_driver。

Linux 中大部分的设备驱动，都可以使用这套机制，设备用 Platform_device 表示，驱动用 Platform_driver 进行注册。

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

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

通过 Platform 机制开发发底层驱动的大致流程为： 定义 platform_device ，注册 platform_device ，定义 platform_driver 注册 platform_driver。

7. Bmp 文件

BMP 是一种与硬件设备无关的图像文件格式，使用非常广。它采用位映射存储格式，除了图像深度可选以外，不采用其他任何压缩，因此，BblP 文件所占用的空间很大。BMP 文件的图像深度可选 lbit、4bit、8bit 及 24bit。BMP 文件存储数据时，图像的扫描方式是按从左到右、从下到上的顺序。

由于 BMP 文件格式是 Windows 环境中交换与图有关的数据的一种标准，因此在 Windows 环境中运行的图形图像软件都支持 BMP 图像格式。

典型的 BMP 图像文件由三部分组成：位图文件头数据结构，它包含 BMP 图像文件的类型、显示内容等信息；位图信息数据结构，它包含有 BMP 图像的宽、高、压缩方法，以及定义颜色等信息。

一、图像文件头

1）1：图像文件头。424Dh="BM"，表示是 Windows 支持的 BMP 格式。

2）2-3：整个文件大小。4690 0000，为 00009046h=36934。

3）4-5：保留，必须设置为 0。

4）6-7：从文件开始到位图数据之间的偏移量。4600 0000，为 00000046h=70，上面的文件头就是 35 字=70 字节。

5）8-9：位图图信息头长度。

6）10-11：位图宽度，以像素为单位。8000 0000，为 00000080h=128。

7）12-13：位图高度，以像素为单位。9000 0000，为 00000090h=144。

8）14：位图的位面数，该值总是 1。0100，为 0001h=1。

二、位图信息头

9）15：每个像素的位数。有 1（单色），4（16 色），8（256 色），16（64K 色，高彩色），24（16M 色，真彩色），32（4096M 色，增强型真彩色）。1000 为 0010h=16。

10）16-17：压缩说明：有 0（不压缩），1（RLE 8，8 位 RLE 压缩），2（RLE 4，4 位 RLE 压缩，3（Bitfields，位域存放）。RLE 简单地说是采用像素数+像素值的方式进行压缩。T408 采用的是位域存放方式，用两个字节表示一个像素，位域分配为 r5b6g5。图中 0300 0000 为 00000003h=3。

11）18-19：用字节数表示的位图数据的大小，该数必须是 4 的倍数，数值上等于位图宽度×位图高度×每个像素位数。0090 0000 为 00009000h=80 × 90 × 2h=36864。

12）20-21：用象素/米表示的水平分辨率。A00F 0000 为 0000 0FA0h=4000。

13）22-23：用象素/米表示的垂直分辨率。A00F 0000 为 0000 0FA0h=4000。

14）24-25：位图使用的颜色索引数。设为 0 的话，则说明使用所有调色板项。

15）26-27：对图象显示有重要影响的颜色索引的数目。如果是 0，表示都重要。

三、彩色板 （非必有）

16）28-35：彩色板规范。对于调色板中的每个表项，用下述方法来描述 RGB 的值：

1 字节用于蓝色分量

1 字节用于绿色分量

1 字节用于红色分量

1 字节用于填充符(设置为 0)

对于 24-位真彩色图像就不使用彩色板，因为位图中的 RGB 值就代表了每个象素的颜色。

如，彩色板为 00F8 0000 E007 0000 1F00 0000 0000 0000，其中：

00FB 0000 为 FB00h=1111100000000000（二进制），是红色分量的掩码。

E007 0000 为 07E0h=0000011111100000（二进制），是绿色分量的掩码。

1F00 0000 为 001Fh=0000000000011111（二进制），是红色分量的掩码。

0000 0000 总设置为 0。

将掩码跟像素值进行“与”运算再进行移位操作就可以得到各色分量值。看看掩码，就可以明白事实上在每个像素值的两个字节 16 位中，按从高到低取 5、6、5 位分别就是 r、g、b 分量值。取出分量值后把 r、g、b 值分别乘以 8、4、8 就可以补齐第个分量为一个字节，再把这三个字节按 rgb 组合，放入存储器（同样要反序），就可以转换为 24 位标准 BMP 格式了。

四、图像数据阵列

17)17-. . .：每两个字节表示一个像素。阵列中的第一个字节表示位图左下角的象素，而最后一个字节表示位图右上角的象素。

第二章 主要框架说明

1. Linux 音频驱动

一个典型的数字音频系统的电路组成如图 2.1 所示：嵌入式微控制器 DSP 中集成了 PCM、IIS、或 AC97 音频接口，通过这些接口连接外部的音频编解码器即可实现声音 AD 和 DA 转换，图中的功放完成模拟信号的放大功能

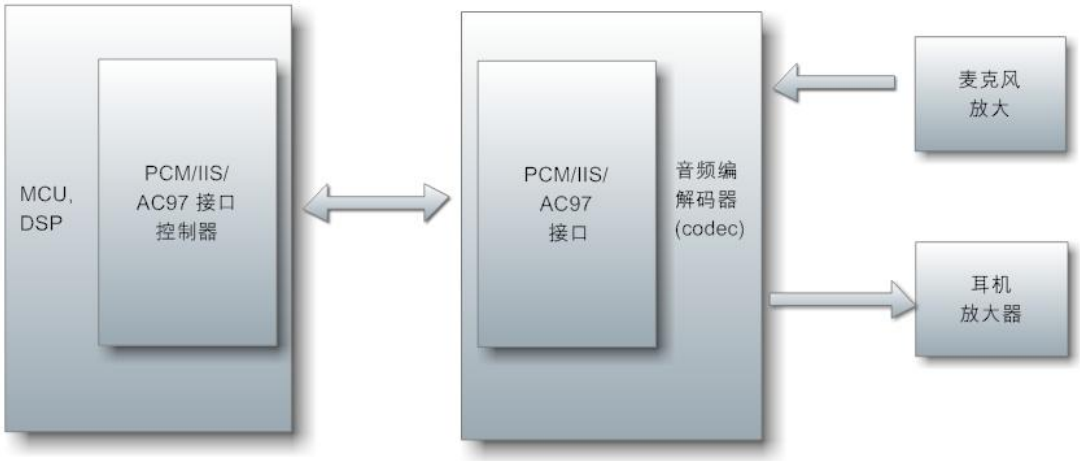

图 2.1 典型的数字音频系统电路

1.1 OSS

OSS 标准中有两个最基本的音频设备：mixer（混音器）和 dsp （数字信号处理器）。

在声卡硬件电路中，mixer 是一个重要的组成部分，它的作用是将多个信号组合或者叠加在一起。/dev/mixer 设备文件是应用程序对 mixer 进行操作的软件接口。

DSP 也称编解码器，实现录音和放音（播放），其对应的设备文件是/dev/dsp 或/dev/sound/dsp。OSS 声卡驱动程序提供的/dev/dsp/是用于数字采样和数字录音设备文件，向该设备写数据即意味着激活声卡上的 D/A 转换器进行播放，而向该设备读数据则意思激活声卡上的 A/D 转换器进行录音。

1. dsp 编程

dsp 接口的操作一般包括如下几个步骤：

(1) 打开设备文件/dev/dsp。

(2) 如果有需要，设置缓冲区大小。

(3) 设置声道数量。

(4) 设置采样格式和采样频率。

(5) 读写/dev/dsp 实现播放或录音

2. mixer 编程

声卡上的混音器由多个混音通道组成，它们可以通过驱动程序提供的设备文件/dev/mixer 进行编程。

对声卡的输入增益和输出增益进行调节是混音器的一个主要作用，目前大部分声卡采用的是 8 位或者 16 位的增益控制器，声卡驱动程序会将它们转换成百分比的形式，也就是说无论是输入增益还是输出增益，其取值范围都是从 0~100。

根据 dsp 编程和 mixer 编程，我们可以画出 Linux OSS 驱动的结构简图，如图 2.2 所示。

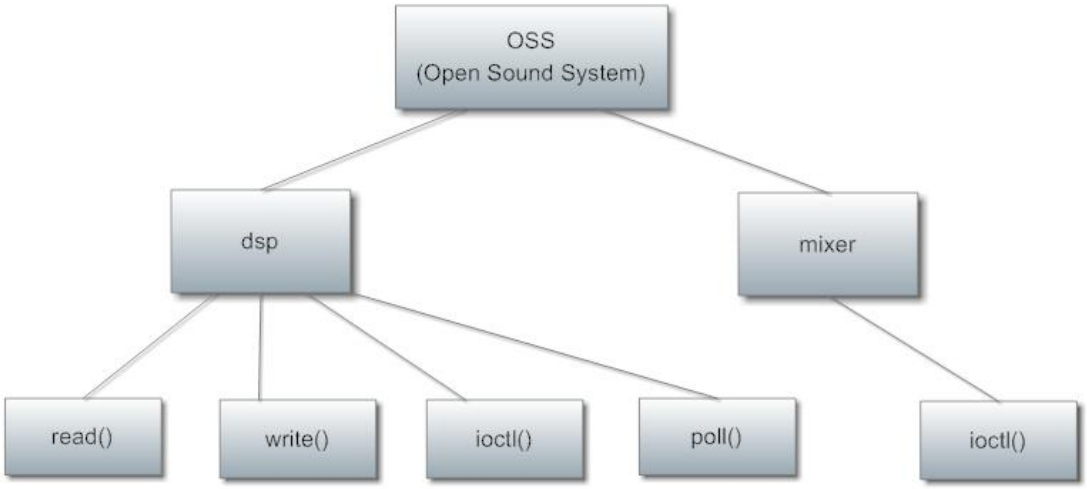

图 2.2 Linux OSS 驱动结构

1.2 ALSA

虽然 OSS 已经非常成熟，但它毕竟是一个没有完全开放源代码的商业产品，ALSA(Advanced Linux Sound Architecture) 恰好填补了这一空白。ALSA 的主要特点：支持多种声卡设备、模块化的内核驱动程序、支持 SMP 和多线程、提供应用开发函数库(alsa-lib)以简化应用程序开发、支持 OSS API，兼容 OSS 应用程序

图 2.3 ALSA 驱动体系

由图 2.3 所示：ALSA 声卡驱动体系中的 alsa-driver 与 alsa-lib 是整个 alsa 音频驱动的中间层，alsa-libs 指向用户空间的函数库，提供给应用程序使用，应用程序应包含头文件 asoundlib.h,并使用共享库 libasound.so; Alsa-driver 指向内核驱动程序，包括硬件相关的代码和一些公共代码，非常庞大，代码总量达数十万行; 上层 alsa app 只需要集中注意力到应用逻辑,下层的 alsa device driver 也只需要关注如何实现 alsa driver 要求的接口.

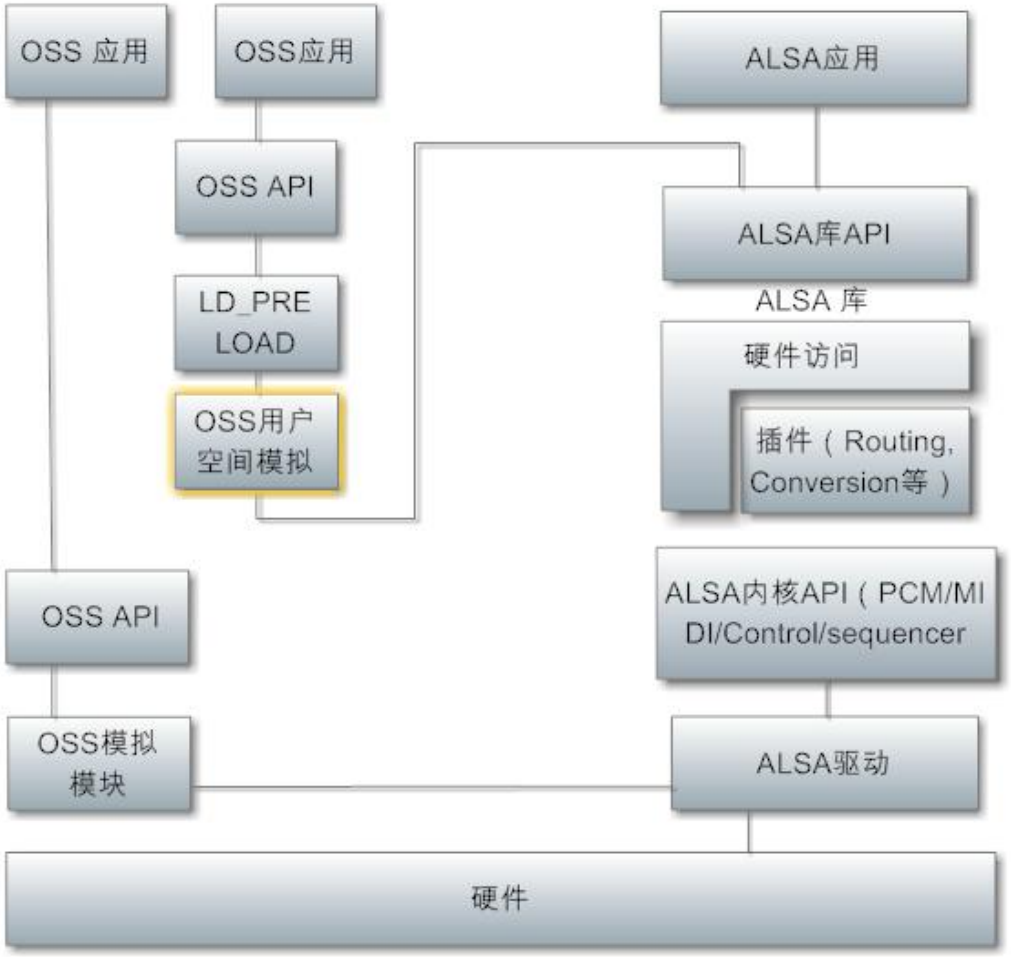

图 2.4 ALSA 体系结构

和 OSS 类似，ALSA 的接口也以文件的方式被提供，不同的是这些接口被提供给 alsa-lib 使用，而不是直接给应用程序使用的，应用程序最好使用 alsa-lib，或者更高级的接口，比如 jack 提供的接口。

图 2.4 所示为 ALSA 声卡驱动与用户空间体系结构的简图，从中可以看出 ALSA 内核驱动与用户空间库及 OSS 之间的关系。

2. Camera 驱动

2.1 V4L

1.什么是 video4linux
Video4linux（简称 V4L),是 linux 中关于视频设备的内核驱动。在 Linux 中，视频设备是设备文件，可以像访问普通 文件一样对其进行读写，摄像头在/dev/video0 下。
2.Video4linux 下视频编程的流程
（1）打开视频设备：
（2） 读取设备信息
（3）更改设备当前设置（没必要的话可以不做）
（4）进行视频采集，两种方法:
a.内存映射
b.直接从设备读取
（5）对采集的视频进行处理
（6）关闭视频设备。

为程序定义的数据结构

```
•typedef struct v4l_struct
•{
•    int fd;
•    struct video_capability capability;
•    struct video_channel channel[4];
•    struct video_picture picture;
•    struct video_window window;
•    struct video_capture capture;
•    struct video_buffer buffer;
•    struct video_mmap mmap;
•    struct video_mbuf mbuf;
•    unsigned char *map;
•    int frame;
•    int framestat[2];
•}vd;
```

3.Video4linux 支持的数据结构及其用途
（1） video_capability
•包含设备的基本信息（设备名称、支持的最大最小分辨率、信号源信息等）
•包含的分量：
•name[32] //设备名称
•maxwidth ，maxheight，minwidth，minheight
•Channels //信号源个数
•type //是否能 capture，彩色还是黑白，是否能裁剪等等。值如 VID_TYPE_CAPTURE

等

•

（2）video_picture

•设备采集的图象的各种属性

•brightness 0~65535

•hue

•colour

•contrast

•whiteness

•depth // 24

•palette //VIDEO_PALETTE_RGB24

（3）video_channel 关于各个信号源的属性

Channel //信号源的编号

name

tuners

Type VIDEO_TYPE_TV | IDEO_TYPE_CAMERA

Norm 制式

（4）video_window //包含关于 capture area 的信息

xx windows 中的坐标.

y x windows 中的坐标.

width The width of the image capture.

height The height of the image capture.

chromakey A host order RGB32 value for the chroma key.

flags Additional capture flags.

clips A list of clipping rectangles. (Set only)

clipcount The number of clipping rectangles. (Set only)

（5）video_mbuf 利用 mmap 进行映射的帧的信息

size //每帧大小

Frames //最多支持的帧数

Offsets //每帧相对基址的偏移

（6）video_buffer 最底层对 buffer 的描述

void *baseBase physical address of the buffer

int heightHeight of the frame buffer

int widthWidth of the frame buffer

int depthDepth of the frame buffer

int bytesperlineNumber of bytes of memory between the start of two adjacent lines

实际显示的部分一般比它描述的部分小

（7）video_mmap //用于 mmap

4.关键步骤介绍

（1）打开视频：

Open（”/dev/video0”，vdàfd);
关闭视频设备用 close（”/dev/video0”，vdàfd);
（2）读 video_capability 中信息
ioctl(vd->fd, VIDIOCGCAP, &(vd->capability))
成功后可读取 vd->capability 各分量 eg.
（3）读 video_picture 中信息
ioctl(vd->fd, VIDIOCGPICT, &(vd->picture))；
（4）改变 video_picture 中分量的值 （可以不做的）
先为分量赋新值，再调用 VIDIOCSPICT
Eg.

```
•vd->picture.colour = 65535;
•if(ioctl(vd->fd, VIDIOCSPICT, &(vd->picture)) < 0)
•{
•perror("VIDIOCSPICT");
•return -1;
•}
```

（5）初始化 channel （可以不做的）

```
•必须先做得到 vd->capability 中的信息
•for (i = 0; i < vd->capability.channels; i++)
•    {
•        vd->channel[i].channel = i;
•        if (ioctl(vd->fd, VIDIOCGCHAN, &(vd->channel[i])) < 0)
•  {
•            perror("v4l_get_channel:");
•            return -1;
•            }
•    }
```

重点：截取图象的两种方法
一、用 mmap（内存映射）方式截取视频
•mmap( )系统调用使得进程之间通过映射同一个普通文件实现共享内存。普通文件被映射到进程地址空间后，进程可以向访问普通内存一样对文件进行访问，不必再调用 read()，write（）等操作。
•两个不同进程 A、B 共享内存的意思是，同一块物理内存被映射到进程 A、B 各自的进程地址空间。进程 A 可以即时看到进程 B 对共享内存中数据的更新，反之亦然
•采用共享内存通信的一个显而易见的好处是效率高，因为进程可以直接读写内存，而不需要任何数据的拷贝
（1）设置 picture 的属性
（2） 初始化 video_mbuf，以得到所映射的 buffer 的信息
ioctl(vd->fd, VIDIOCGMBUF, &(vd->mbuf))
（3）可以修改 video_mmap 和帧状态的当前设置

```
•    Eg.      vd->mmap.format = VIDEO_PALETTE_RGB24
•               vd->framestat[0] = vd->framestat[1] = 0;   vd->frame = 0;
```

（4）将 mmap 与 video_mbuf 绑定

•void* mmap (void * addr , size_t len , int prot , int flags , int fd , off_t offset)
•len //映射到调用进程地址空间的字节数，它从被映射文件开头 offset 个字节开始算起
•Prot //指定共享内存的访问权限 PROT_READ（可读）, PROT_WRITE （可写）, PROT_EXEC （可执行）
•flags // MAP_SHARED MAP_PRIVATE 中必选一个 // MAP_ FIXED 不推荐使用 addr // 共内存享的起始地址，一般设 0，表示由系统分配
•Mmap( ) 返回值是系统实际分配的起始地址
•if((vd->map = (unsigned char*)mmap(0, vd->mbuf.size, PROT_READ|PROT_WRITE, MAP_SHARED, vd->fd, 0)) < 0)
•{
•perror("v4l_mmap mmap:");
•return -1;
•}
（5）Mmap 方式下真正做视频截取的 VIDIOCMCAPTURE
ioctl(vd->fd, VIDIOCMCAPTURE, &(vd->mmap)) ;
•若调用成功，开始一帧的截取，是非阻塞的，
•是否截取完毕留给 VIDIOCSYNC 来判断
（6）调用 VIDIOCSYNC 等待一帧截取结束
•if(ioctl(vd->fd, VIDIOCSYNC, &frame) < 0)
•{
•perror("v4l_sync:VIDIOCSYNC");
•return -1;
•}
若成功，表明一帧截取已完成。可以开始做下一次 VIDIOCMCAPTURE
•frame 是当前截取的帧的序号。

****关于双缓冲：
•video_bmuf bmuf.frames = 2;
•一帧被处理时可以采集另一帧
•int frame; //当前采集的是哪一帧
•int framestat[2]; //帧的状态 没开始采集|等待采集结束
•帧的地址由 vd->map + vd->mbuf.offsets[vd->frame]得到
•采集工作结束后调用 munmap 取消绑定
•munmap(vd->map, vd->mbuf.size)

二、视频截取的第二种方法：直接读设备
关于缓冲大小，图象等的属性须由使用者事先设置
•调用 read（）；
•int read (要访问的文件描述符；指向要读写的信息的指针；应该读写的字符数);
•返回值为实际读写的字符数
•int len ;

2.2 V4L2

一.什么是 video4linux2

Video4linux2（简称 V4L2), V4L2 较 V4L 有较大的改动，并已成为 2.6 的标准接口。V4L2 采用流水线的方式，操作更简单直观，基本遵循打开视频设备、设置格式、处理数据、关闭设备，更多的具体操作通过 ioctl 函数来实现。

般操作流程（视频设备）：

1. 打开设备文件。 int fd=open(”/dev/video0″,O_RDWR);
2. 取得设备的 capability，看看设备具有什么功能，比如是否具有视频输入,或者音频输入输出等。VIDIOC_QUERYCAP,struct v4l2_capability
3. 选择视频输入，一个视频设备可以有多个视频输入。VIDIOC_S_INPUT,struct v4l2_input
4. 设置视频的制式和帧格式，制式包括 PAL，NTSC，帧的格式个包括宽度和高度等。VIDIOC_S_STD,VIDIOC_S_FMT,struct v4l2_std_id,struct v4l2_format
5. 向驱动申请帧缓冲，一般不超过 5 个。struct v4l2_requestbuffers
6. 将申请到的帧缓冲映射到用户空间，这样就可以直接操作采集到的帧了，而不必去复制。mmap
7. 将申请到的帧缓冲全部入队列，以便存放采集到的数据.VIDIOC_QBUF,struct v4l2_buffer
8. 开始视频的采集。VIDIOC_STREAMON
9. 出队列以取得已采集数据的帧缓冲，取得原始采集数据。VIDIOC_DQBUF
10. 将缓冲重新入队列尾,这样可以循环采集。VIDIOC_QBUF
11. 停止视频的采集。VIDIOC_STREAMOFF
12. 关闭视频设备。close(fd);

三、常用的结构体(参见/usr/include/linux/videodev2.h)：

struct v4l2_requestbuffers reqbufs;//向驱动申请帧缓冲的请求，里面包含申请的个数
struct v4l2_capability cap;//这个设备的功能，比如是否是视频输入设备
struct v4l2_input input; //视频输入
struct v4l2_standard std;//视频的制式，比如 PAL，NTSC
struct v4l2_format fmt;//帧的格式，比如宽度，高度等
struct v4l2_buffer buf;//代表驱动中的一帧
v4l2_std_id stdid;//视频制式，例如：V4L2_STD_PAL_B
struct v4l2_queryctrl query;//查询的控制
struct v4l2_control control;//具体控制的值

下面具体说明开发流程

打开视频设备

在 V4L2 中，视频设备被看做一个文件。使用 open 函数打开这个设备：

```
1.  // 用非阻塞模式打开摄像头设备
2.  int cameraFd;
```

```
3.    cameraFd = open("/dev/video0", O_RDWR | O_NONBLOCK, 0);
4.    // 如果用阻塞模式打开摄像头设备，上述代码变为：
5.    //cameraFd = open("/dev/video0", O_RDWR, 0);
```

关于阻塞模式和非阻塞模式

应用程序能够使用阻塞模式或非阻塞模式打开视频设备，如果使用非阻塞模式调用视频设备，即使尚未捕获到信息，驱动依旧会把缓存（DQBUFF）里的东西返回给应用程序。

设定属性及采集方式

打开视频设备后，可以设置该视频设备的属性，例如裁剪、缩放等。这一步是可选的。在 Linux 编程中，一般使用 ioctl 函数来对设备的 I/O 通道进行管理：

```
extern int ioctl (int __fd, unsigned long int __request, ...) __THROW;
```

__fd：设备的 ID，例如刚才用 open 函数打开视频通道后返回的 cameraFd；

__request：具体的命令标志符。

在进行 V4L2 开发中，一般会用到以下的命令标志符：

1. VIDIOC_REQBUFS：分配内存
2. VIDIOC_QUERYBUF：把 VIDIOC_REQBUFS 中分配的数据缓存转换成物理地址
3. VIDIOC_QUERYCAP：查询驱动功能
4. VIDIOC_ENUM_FMT：获取当前驱动支持的视频格式
5. VIDIOC_S_FMT：设置当前驱动的频捕获格式
6. VIDIOC_G_FMT：读取当前驱动的频捕获格式
7. VIDIOC_TRY_FMT：验证当前驱动的显示格式
8. VIDIOC_CROPCAP：查询驱动的修剪能力
9. VIDIOC_S_CROP：设置视频信号的边框
10. VIDIOC_G_CROP：读取视频信号的边框
11. VIDIOC_QBUF：把数据从缓存中读取出来
12. VIDIOC_DQBUF：把数据放回缓存队列
13. VIDIOC_STREAMON：开始视频显示函数
14. VIDIOC_STREAMOFF：结束视频显示函数
15. VIDIOC_QUERYSTD：检查当前视频设备支持的标准，例如 PAL 或 NTSC。

这些 IO 调用，有些是必须的，有些是可选择的。

检查当前视频设备支持的标准

在亚洲，一般使用 PAL（720X576）制式的摄像头，而欧洲一般使用 NTSC（720X480），使用 VIDIOC_QUERYSTD 来检测：

```
1.    v4l2_std_id std;
2.    do {
3.    ret = ioctl(fd, VIDIOC_QUERYSTD, &std);
4.    } while (ret == -1 && errno == EAGAIN);
5.    switch (std) {
6.    case V4L2_STD_NTSC:
```

```
7.   //……
8.   case V4L2_STD_PAL:
9.   //……
10.  }
```

设置视频捕获格式

当检测完视频设备支持的标准后，还需要设定视频捕获格式：

```
1.   struct v4l2_format    fmt;
2.   memset ( &fmt, 0, sizeof(fmt) );
3.   fmt.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
4.   fmt.fmt.pix.width = 720;
5.   fmt.fmt.pix.height = 576;
6.   fmt.fmt.pix.pixelformat = V4L2_PIX_FMT_YUYV;
7.   fmt.fmt.pix.field = V4L2_FIELD_INTERLACED;
8.   if (ioctl(fd, VIDIOC_S_FMT, &fmt) == -1) {
9.   return -1;
10.  }
```

v4l2_format 结构体定义如下：

```
1.   struct v4l2_format
2.   {
3.   enum v4l2_buf_type type;//数据流类型，必须永远是
     V4L2_BUF_TYPE_VIDEO_CAPTURE
4.   union
5.   {
6.   struct v4l2_pix_format    pix;
7.   struct v4l2_window        win;
8.   struct v4l2_vbi_format    vbi;
9.   __u8    raw_data[200];
10.  } fmt;
11.  };
12.
13.  struct v4l2_pix_format
14.  {
15.  __u32                  width;   // 宽，必须是 16 的倍数
16.  __u32                  height;  // 高，必须是 16 的倍数
17.  __u32                  pixelformat;//视频数据存储类型，如 YUV422,RGB
18.  enum v4l2_field        field;
19.  __u32                  bytesperline;
20.  __u32                  sizeimage;
21.  enum v4l2_colorspace   colorspace;
22.  __u32                  priv;
23.  };
```

分配内存

接下来可以为视频捕获分配内存：

```
1.   struct v4l2_requestbuffers  req;
2.   if (ioctl(fd, VIDIOC_REQBUFS, &req) == -1) {
3.   return -1;
4.   }
```

v4l2_requestbuffers 定义如下：

```
1.   struct v4l2_requestbuffers
2.   {
3.   __u32 count;  //缓存数量，也就是说在缓存队列里保持多少张照片
4.   enum v4l2_buf_type type;// 数据流类型，必须永远是
   V4L2_BUF_TYPE_VIDEO_CAPTURE
5.   enum v4l2_memory   memory; // V4L2_MEMORY_MMAP 或 V4L2_MEMORY_USERPT
   R
6.   __u32              reserved[2];
7.   };
```

获取并记录缓存的物理空间

使用 VIDIOC_REQBUFS，我们获取了 req.count 个缓存，下一步通过调用 VIDIOC_QUERYBUF 命令来获取这些缓存的地址，然后使用 mmap 函数转换成应用程序中的绝对地址，最后把这段缓存放入缓存队列：

```
1.   typedef struct VideoBuffer {
2.   void *start;
3.   size_t  length;
4.   } VideoBuffer;
5.
6.
7.   VideoBuffer*          buffers = calloc( req.count, sizeof(*buffers)
   );
8.   struct v4l2_buffer    buf;
9.
10.  for (numBufs = 0; numBufs < req.count; numBufs++) {
11.  memset( &buf, 0, sizeof(buf) );
12.  buf.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
13.  buf.memory = V4L2_MEMORY_MMAP;
14.  buf.index = numBufs;
15.  // 读取缓存
16.
17.  if (ioctl(fd, VIDIOC_QUERYBUF, &buf) == -1) {
18.  return -1;
19.  }
20.
21.  buffers[numBufs].length = buf.length;
22.
23.  // 转换成相对地址
24.  buffers[numBufs].start = mmap(NULL, buf.length,
```

```
25.  PROT_READ | PROT_WRITE,
26.  MAP_SHARED,
27.  fd, buf.m.offset);
28.
29.  if (buffers[numBufs].start == MAP_FAILED) {
30.  return -1;
31.  }
32.
33.  // 放入缓存队列
34.  if (ioctl(fd, VIDIOC_QBUF, &buf) == -1) {
35.  return -1;
36.  }
37.  }
```

关于视频采集方式

操作系统一般把系统使用的内存划分成用户空间和内核空间，分别由应用程序管理和操作系统管理。应用程序可以直接访问内存的地址，而内核空间存放的是 供内核访问的代码和数据，用户不能直接访问。v4l2 捕获的数据，最初是存放在内核空间的，这意味着用户不能直接访问该段内存，必须通过某些手段来转换地 址。

一共有三种视频采集方式：使用 read、write 方式；内存映射方式和用户指针模式。

read、write 方式:在用户空间和内核空间不断拷贝数据，占用了大量用户内存空间，效率不高。

内存映射方式：把设备里的内存映射到应用程序中的内存控件，直接处理设备内存，这是一种有效的方式。上面的 mmap 函数就是使用这种方式。

用户指针模式：内存片段由应用程序自己分配。这点需要在 v4l2_requestbuffers 里将 memory 字段设置成 V4L2_MEMORY_USERPTR。

处理采集数据

V4L2 有一个数据缓存，存放 req.count 数量的缓存数据。数据缓存采用 FIFO 的方式，当应用程序调用缓存数据时，缓存队列将最先采集到的 视频数据缓存送出，并重新采集一张视频数据。这个过程需要用到两个 ioctl 命令,VIDIOC_DQBUF 和 VIDIOC_QBUF:

```
1.  struct v4l2_buffer buf;
2.  memset(&buf,0,sizeof(buf));
3.  buf.type=V4L2_BUF_TYPE_VIDEO_CAPTURE;
4.  buf.memory=V4L2_MEMORY_MMAP;
5.  buf.index=0;
6.
7.  //读取缓存
8.  if (ioctl(cameraFd, VIDIOC_DQBUF, &buf) == -1)
9.  {
10.  return -1;
11.  }
12.
```

```
13.  //……视频处理算法
14.  //重新放入缓存队列
15.  if (ioctl(cameraFd, VIDIOC_QBUF, &buf) == -1) {
16.  return -1;
17.  }
```

关闭视频设备

使用 close 函数关闭一个视频设备

```
close(cameraFd)
```

还需要使用 munmap 方法。

3. Android Audio System

Audio System 在 Android 中负责音频方面的数据流传输和控制功能，也负责音频设备的管理。这个部分作为 Android 的 Audio 系统的输入/输出层次，一般负责播放 PCM 声音输出和从外部获取 PCM 声音，以及管理声音设备和设置。

Audio 系统主要分成如下几个层次：

1) media 库提供的 Audio 系统本地部分接口；
2) AudioFlinger 作为 Audio 系统的中间层；
3) Audio 的硬件抽象层提供底层支持；
4) Audio 接口通过 JNI 和 Java 框架提供给上层。

Audio 系统的各个层次接口主要提供了两方面功能：放音（Track）和录音（Recorder）。

Android 的 Audio 系统结构如图 2.5 所示。

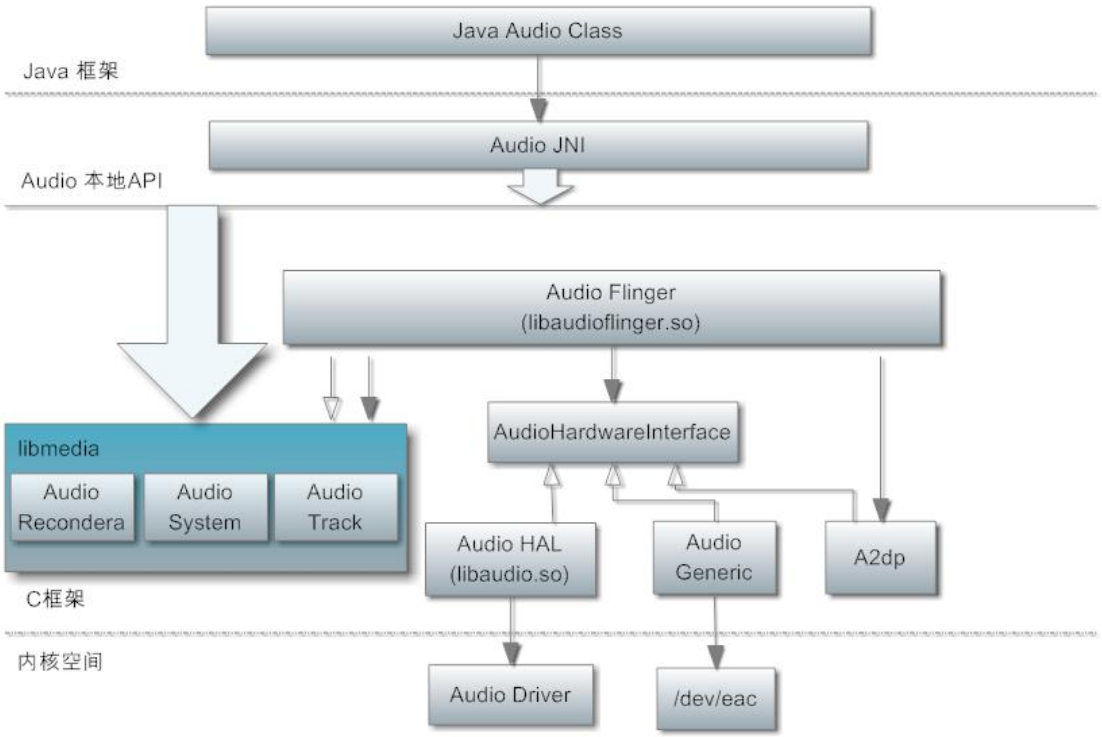

图 2.5 Android 的 Audio 系统结构

Audio 系统的各层次情况如下所示。

Audio 本地框架类是 libmedia.so 的一个部分，这些 Audio 接口对上层提供接口，由下层的本地代码去实现。

AudioFlinger 继承 libmeida 中的接口，提供实现库 libaudiofilnger.so。这部分内容没有自己的对外头文件，上层调用的只是 libmedia 本部分的接口，但实际调用的内容是 libaudioflinger.so。

Audio 使用 JNI 和 Java 对上层提供接口，JNI 部分通过调用 libmedia 库提供的接口来实现。

Audio 的硬件抽象层提供到硬件的接口，供 AudioFlinger 调用。Audio 的硬件抽象层实际上是各个平台开发过程中需要主要关注和独立完成的部分。

Android 的 Audio 系统不涉及编解码环节，只是负责上层系统和底层 Audio 硬件的交互，一般以 PCM 作为输入/输出格式。

在 Android 的 Audio 系统中，无论上层还是下层，都使用一个管理类和输出输入两个类来表示整个 Audio 系统，输出输入两个类负责数据通道。在各个层次之间具有对应关系，如表 2-1 所示所示。

表 2-1 Android 各个层次的对应关系

	Audio 管理环节	Audio 输出	Audio 输入
Java 层	android.media.AudioSystem	android.medie.AudioTrack	android.media.AudioRecorder
本地框架层	AudioSystem	AudioTrack	AudioRecorder
AudioFlinger	IAudioFlinger	IAudioTrack	IAudioRecorder
硬件抽象层	AudioHardwareInterface	AudioStreamOut	AudioStreamIn

4. Android Video output System: Overlay System

Overlay 系统在 Android 中作为一个独立的部分，其含义是在主显示区域之外的显示区域，使用 OverLay（叠加）方式，放置于主显示区域之上。

Android 的 Overlay 系统是只具有底层的系统，它在 Java 层没有接口，因此也没有 JNI 的部分。主要涉及的部分有：

a. 在 libui 中提供的 Overlay 框架部分；

b. SurfaceFlinger 部分提供 Overlay 中间层；

c. Overlay 硬件抽象层。

Android 的 Overlay 系统结构如图 2.6 所示。

在 Android 中，Overlay 系统的开发工作与其他系统略有不同，共他系统一般只需要构建移植层（硬件抽象层）即可；Overlay 系统不仅需要构建移植层，还需要在其他地方主动使用 Overlay 系统。主要使用 Overlay 系统的地方有两个：视频播放器的实现部分的输出环节和 Camera 的硬件抽象层。这两个环节分别用于解码视频数据和摄像头取景器预览数据的输出。

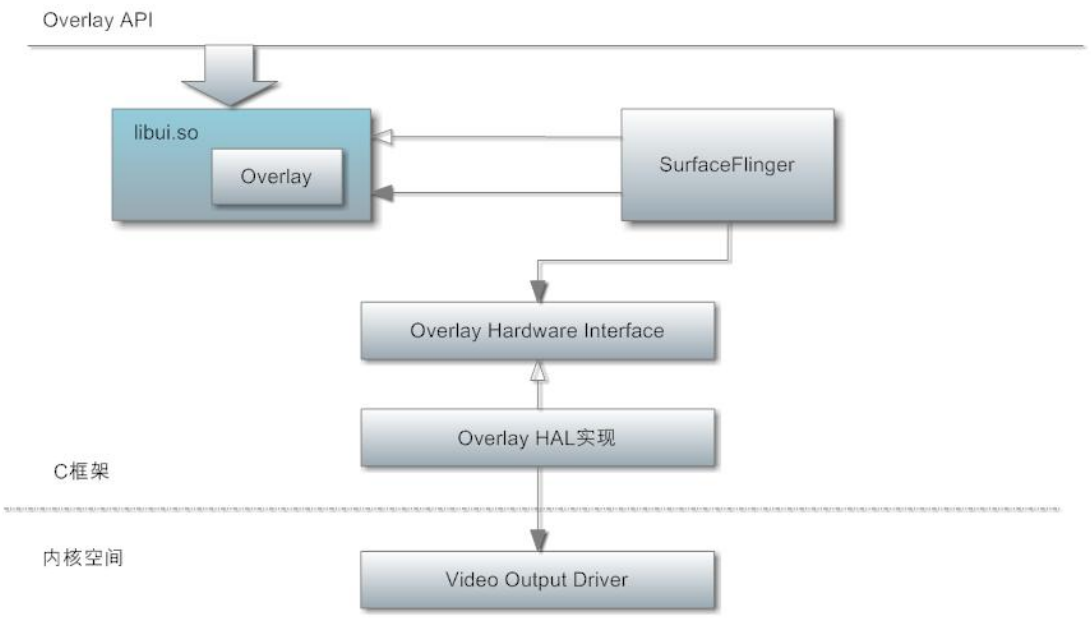

图 2.6 Android 的 Overlay 系统结构

5. Android Video ouput System: Camera System

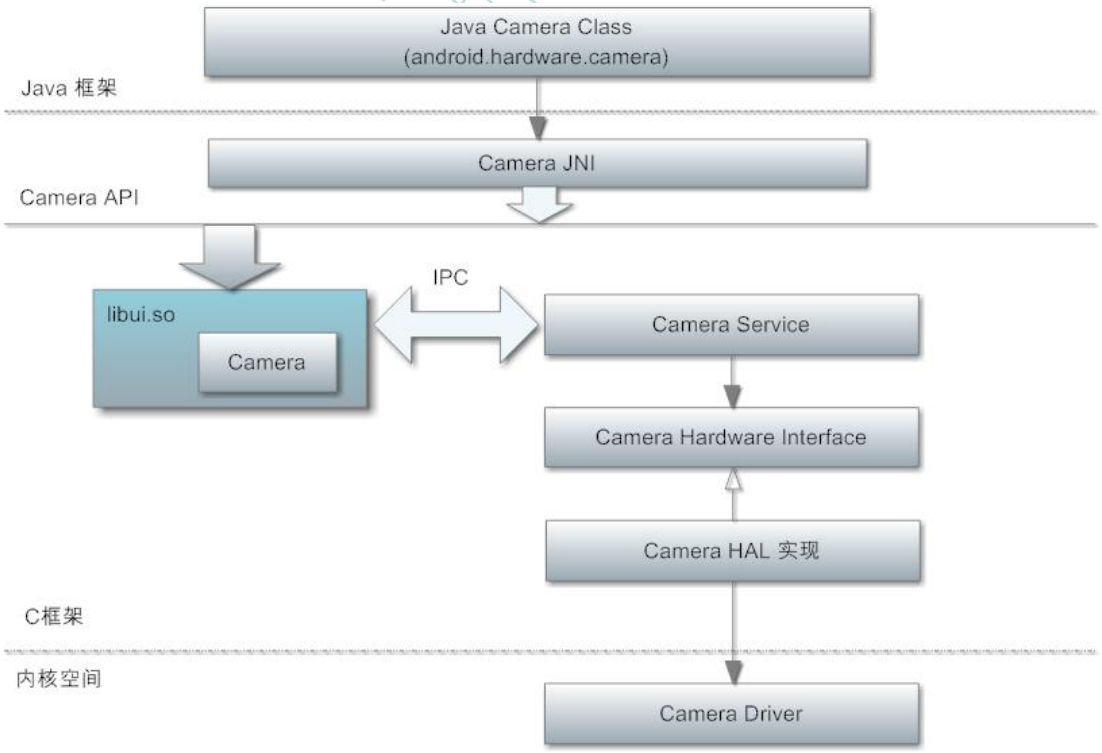

图 2.6 Android 的 Camera 系统结构

Android 的 Camera 系统涉及的功能主要是视频输入部分，在照相机、摄像机、视频电话中都会有所使用。

Camera 系统结构包含了本地代码和 Java 代码两个层次，其中 Camera 本地代码分成有个部分

a. 在 libui 中提供的 Camera 框架部分；

b. CameraSeries 提供中间层支持；

c. Camera 硬件抽象层。

Camera 本地代码提供的 API 在 libui 中，提供了取景器预览、视频录制、拍摄相片 3 个方面的功能。其中 API 提供给两个部分调用：一方面通过封装 Camera 的 JNI 接口到 Java，实现 Android 照相机应用的 Java 实现部分；另一方面， Camera 可以给本地代码的其他程序使用，作为视频系统输入环节使用，例如在摄像机和视频电话中使用。

Android 的 Camera 系统结构如图 2.6 所示

Camera 系统处理的宏观逻辑是：上层通过调用控制接口来来控件下层，并设置回调函数；下层通过回调函数向上层传递各种类型的数据（预览数据、视频数据、照片数据等）。

从代码分布的角度看，CameraService 调用 Camera 的硬件抽象层，二者运行在同一个进程中，libui 实现 Camera.h 中定义的 Camera 类，Camera 系统 的调用者通过调用 libui 来完成，它运行在另一个进程，通过 IPC 和 Camera 服务进行通信。

在 Camera 实际的调用过程中，调用方式为 Camera 的客户端 →Icamera →Camera 本地的实现。

第三章 功能的实现

1. Android AudioSystem 硬件抽象层的实现

1. 在网上下载移植代码(用 GIT 下载） //这里我要说明一下，网上对于下载的移植代码可能跟我的不同，这主要你要看一下，AudioSystem 这个类里时否定义了 DEVICE_OUT_EARPIECE， 如果没有定义就用我这套移植代码，如果定义了，你就用其它一套移植代码吧，

a. platform_external_alsa-lib:

git clone git://gitorious.org/android-on-freerunner/platform_external_alsa-lib.git

将其复制到 external 目录下，并重命名为 alsa-lib

b. platform_hardware_alsa_sound

git clone git://gitorious.org/android-on-freerunner/platform_hardware_alsa_sound.git

将其复制到 hardware 目录下，并重命名为 libaudio-alsa

c. platform_external_alsa-utils (可选）

git clone git://gitorious.org/android-on-freerunner/platform_external_alsa-utils.git

将其复制到 external 目录下，并重命名为 alsa-utils

另附定义了 DEVICE_OUT_EARPIECE 的代码下载

git clone git://android.git.kernel.org/platform/external/alsa-lib.git
git clone git://android.git.kernel.org/platform/external/alsa-utils.git
git clone git://android.git.kernel.org/platform/hardware/alsa_sound.git

2.修改 system/core/init/device.c 加上一段代码以创建/dev/snd:

```
1.  ......
2.          } else if(!strncmp(uevent->subsystem, "mtd", 3)) {
3.              base = "/dev/mtd/";
4.              mkdir(base, 0755);
5.          } else if(!strncmp(uevent->subsystem, "sound", 5)) {
6.              base = "/dev/snd/";
7.              mkdir(base, 0755);
```

3. 修改 system/core/init/devices.c,增加设备节点及权限:

```
1.  static struct perms_ devperms[] = {
2.  ...
3.      { "/dev/snd/",          0664,   AID_SYSTEM,     AID_AUDIO,      1 },
4.  ...
5.
```

4. 修改：build/target/board/generic/BoardConfig.mk

```
1.  1 # config.mk
2.    2 #
3.    3 # Product-specific compile-time definitions.
4.    4 #
5.    5
6.    6 # The generic product target doesn't have any hardware-specific pieces.
7.    7 TARGET_NO_BOOTLOADER := true
8.    8 TARGET_NO_KERNEL := true
9.    9 TARGET_NO_RADIOIMAGE := true
10. 10 #HAVE_HTC_AUDIO_DRIVER := true
11. 11 BOARD_USES_ALSA_AUDIO := true
12. 12 BUILD_WITH_ALSA_UTILS := true
13. 13 #BOARD_USES_GENERIC_AUDIO := true
14. 14 BOARD_USES_GENERIC_AUDIO := false
```

5. 修改 hardware/libaudio-alsa/Android.mk

```
1.  1 # hardware/libaudio-alsa/Android.mk
2.    2 #
3.    3 # Copyright 2008 Wind River Systems
```

```
4.    4 #
5.    5
6.    6 ifeq ($(strip $(BOARD_USES_ALSA_AUDIO)),true)
7.    7
8.    8   LOCAL_PATH := $(call my-dir)
9.    9
10.  10   include $(CLEAR_VARS)
11.  11
12.  12   LOCAL_ARM_MODE := arm
13.  13   LOCAL_CFLAGS := -D_POSIX_SOURCE
14.  14  # LOCAL_WHOLE_STATIC_LIBRARIES := libasound
15.  15
16.  16   LOCAL_C_INCLUDES += external/alsa-lib/include
17.  17
18.  18   LOCAL_SRC_FILES := AudioHardwareALSA.cpp
19.  19
20.  20   LOCAL_MODULE := libaudio
21.  22   LOCAL_STATIC_LIBRARIES += libaudiointerface \
22.  23   #  libasound
23.  24
24.  25   LOCAL_SHARED_LIBRARIES := \
25.  26     libcutils \
26.  27     libutils \
27.  28     libmedia \
28.  29     libhardware_legacy \
29.  30     libdl \
30.  31     libc \
31.  32     libasound
32.  33
33.  34   include $(BUILD_SHARED_LIBRARY)
34.  35
35.  36 endif
```

6. 重建编译选项：

a). build/envsetup.sh

b). choosecombo

7. make clean (这一步必需的)

8. 编译 make -j2

9. 制作文件系统.

10. 在 asound.conf 文件里需要几个特别的配置（从这里开始，以下步骤可以不做了）

```
ctl.AndroidOut {
       type hw
       card 0
 }
 ctl.AndroidIn {
    type hw
    card 0
}
pcm.AndroidPlayback {
    type hw
   card 0
   device 0
}
pcm.AndroidRecord {
   type hw
  card 0
  device 0
```

11.最后还需要修改 init.rc 文件，重新设置 Audio 驱动的设备节点的 owner 和访问属性（编译后,文件系统中的 init.rc)

chown root audio /dev/snd/controlC0

chown root audio /dev/snd/pcmC0D0c

chown root audio /dev/snd/pcmC0D0p

chown root audio /dev/snd/timer

chmod 0666 /dev/snd/controlC0

chmod 0666 audio /dev/snd/pcmC0D0c

chmod 0666 audio /dev/snd/pcmC0D0p

chmod 0666 audio /dev/snd/timer

2. Android Camera 硬件抽象层的实现（USB Camera）

1. 在 hardware/ 目录下面新建一个文件夹 camera
   mkdir hardware/camera
2. 将 frameworks/base/camera/libcameraservice/ 目录下的 CameraHardwareStub.cpp CameraHardwareStub.h 拷贝一份到 hardware/camera 中
   cp frameworks/base/camera/libcameraservice/CameraHardwareStub.* hardware/camera
3. 用替换的方法，将 CameraHardwareStub.h, CameraHardwareStub.cpp 两个文件中的 CameraHardwareStub 都替换成 CameraHardware;
4. 修改 frameworks/base/camera/libcameraservice/ 目录下的 CameraSeriver.cpp
   把 482 行的 PIXEL_FORMAT_YCbCr_420_SP 改成 PIXEL_FORMAT_RGB_565;
   解决黑白问题
5. 在 hardware/camera 目录下新建 UsbCamer.h

```
#ifndef _USBCAMERA_H
#define _USBCAMERA_H
#define NB_BUFFER 4
#define DEFAULT_FRAME_RATE 30
#define VIDEO_PALETTE_JPEG 21
#define DEFAULT_DEVICE "/dev/video0"
#define MAX_JPEG_FILE 128
#define POSIX_FNAME_MAX 256
#define BMP_FILE_HEADER_LEN 54
#define BMP_FILE_WIDTH_FIELD_OFFSET 0x12
#define BMP_FILE_HEIGHT_FIELD_OFFSET 0x16
#define VIDEO_MAXFRAME 2
#include <utils/MemoryBase.h>
#include <utils/MemoryHeapBase.h>
#include <linux/videodev.h>

namespace android {
typedef struct _Usb_Camera_struct
{
    int fd;
    struct video_capability capability;
    struct video_picture picture;
    struct video_window videowin;
    struct video_mmap mmap;
    struct video_mbuf mbuf;
    unsigned char *map; //用于指向图像数据的指针
    int frame_current;
    int frame_using[VIDEO_MAXFRAME]; //这两个变量用于双缓冲。
}UsbCamera_device;
typedef struct {
    long filesize;
```

```
37.     char reserved[2];
38.     long headersize;
39.     long infoSize;
40.     long width;
41.     long depth;
42.     short biPlanes;
43.     short bits;
44.     long biCompression;
45.     long biSizeImage;
46.     long biXPelsPerMeter;
47.     long biYPelsPerMeter;
48.     long biClrUsed;
49.     long biClrImportant;
50. } BMPHEAD;
51.
52. class UsbCamera {
53. public:
54.      UsbCamera();
55.      ~UsbCamera();
56.      int Open(char *dev);
57.      void Close ();
58.      int Init ();
59.      int getNextFrameAsRGB565(unsigned char *buf);
60.      void GrabRawFrame(char *previewBuffer);//jpeg: 640X480
61.
62.
63.  private:
64.
65.      UsbCamera_device vd;
66.      int jpeg_size;
67.
68.      int Get_capability( );
69.      int Get_picture( );
70.      int GetDepth(int format);
71.      int Get_jpeg_size( );
72.      int get_bmpfile_info(const char *rawbmpbuf,
73.                           unsigned char **bmpdata,
74.                           int *width,
75.                           int *height);
76.      void get_RGB565(char *buf, unsigned short *fb16, int xMax, int yMax);
77.      int Grab_init( );
78.      int Read_pic( );
79.      int Set_picture(int br,int hue,int col,int cont,int white);
80.      int Usb_Camera_init( );
```

```
81.      void convertframe(unsigned char *dst, size_t jpegsize) ;
82.      int jpeg2bmp(char *bmp_buf, char *jpg_buf, int jpegsize);
83.      int partition_bmp( char *filename, char *bmpbuf,int w, int h);
84.
85.       int bmp2fb16_rgb565(unsigned char *bmpdata,
86.                                   unsigned short *fb16,
87.                                   int xres,
88.                                   int yres,
89.                                   int bytes_per_line);
90.     int Device_init( );
91. };
92. }; // namespace android
93. #endif
```

6. 在 hardware/camera 目录下新建 UsbCamer.cpp 用于实现类的功能
根据 video for linux 来实现。

7. 修改 hardware/camera 目录下的 CameraHardwareStub.h, CameraHardwareStub.cpp 重命名为 CameraHardware.h, CameraHardware.cpp

8. 修改 CameraHardware.h

a. 在公有成员处增加如下代码：

```
108. #ifdef USB_CAMERA
109.     UsbCamera              *mUsbcamera;
110. #else
111.     FakeCamera          *mFakeCamera;
112. #endif
```

9. 修改 CameraHardware.cpp

a. 在 int CameraHardware::previewThread()中修改

view plaincopy to clipboardprint?

```
190. uint8_t *frame = ((uint8_t *)base) + offset;
191. #ifdef USB_CAMERA
192. mUsbcamera->getNextFrameAsRGB565(frame);
193. #else
194. fakeCamera->getNextFrameAsYuv422(frame);
195. #endif
```

b. 在 status_t CameraHardware::startPreview（）中修改

```
241. #ifdef USB_CAMERA
```

```
242.     LOGE("Start USB Camera");
243.     mUsbcamera->Init();
244. #endif
245.     mPreviewThread = new PreviewThread(this);
246.     return NO_ERROR;
```

c. void CameraHardware::stopPreview()中修改

```
261.  if (mPreviewThread != 0)  {
262.
263. #ifdef USB_CAMERA
264.             LOGE("tianfeng: close usb camera");
265.             //usb_camera_close();
266.             mUsbcamera->Close();
267.   #endif
268. }
```

10. 修改 system/core/init/devices.c,增加设备节点及权限:

```
1.  static struct perms_ devperms[] = {
2.  ...
3.      { "/dev/video0",          0666,   AID_CAMERA,     AID_CAMERA,      0 },

4.  ...
5.
```

11. 重建编译选项：

a). build/envsetup.sh

b). choosecombo

12. make –j2

13. 制作文件系统

14. 修改内核

a. 在 arm linux 的 kernel 目录下 make menuconfig。

b. 首先选择 Multimedia device->下的 Video for linux。加载 video4linux 模块，为视频采集设备提供了编程接口；

c. 然后在 usb support->目录下(*)选择 support for usb 和 usb camera 相关驱动。

d. 最后 make uImage，使用这个内核启动。

3. Android （jffs2）烧写

1. 得到 jffs2 文件的系统的制作程序 mkfs.jffs2 并把其放到 ubntun9.10 系统的 usr/bin 里面去，这个时候要注意一下权限问题。
2. 修改 Android 文件系统中 init.rc 文件。将下面那句话前面的“ #”去掉
   # mount yaffs2 mtd@userdata /data nosuid nodev
   mount yaffs2 mtd@userdata /data nosuid nodev
3. cd 到 Android 文件系统所在的目录，使用以下命令制作 jffs2 文件系统
   mkfs.jffs2 -r filesystem -o filesystem.jffs2 -e 0x4000
4. 修改内核分区表即修改 arch/arm/mach-omap2board-omap3beagle.c 文件

```
1.  static struct mtd_partition omap3beagle_nand_partitions[] = {
2.      /* All the partition sizes are listed in terms of NAND block size */
3.      {
4.          .name       = "X-Loader",
5.          .offset     = 0,
6.          .size       = 4 * NAND_BLOCK_SIZE,
7.          .mask_flags = MTD_WRITEABLE,    /* force read-only */
8.      },
9.      {
10.         .name       = "U-Boot",
11.         .offset     = MTDPART_OFS_APPEND,   /* Offset = 0x80000 */
12.         .size       = 15 * NAND_BLOCK_SIZE,
13.         .mask_flags = MTD_WRITEABLE,    /* force read-only */
14.     },
15.     {
16.         .name       = "U-Boot Env",
17.         .offset     = MTDPART_OFS_APPEND,   /* Offset = 0x260000 */
18.         .size       = 1 * NAND_BLOCK_SIZE,
19.     },
20.     {
21.         .name       = "Kernel",
22.         .offset     = MTDPART_OFS_APPEND,   /* Offset = 0x280000 */
23.         .size       = 32 * NAND_BLOCK_SIZE,
24.     },
25.     {
26.         .name       = "File System",
27.         .offset     = MTDPART_OFS_APPEND,   /* Offset = 0x680000 */
28.         .size       = 80*(8*NAND_BLOCK_SIZE),
```

```
29.     },
30.     {
31.         .name       = "userdata",
32.         .offset     = MTDPART_OFS_APPEND,    /* offset = 0x5680000 */
33.         .size       = 32*(8*NAND_BLOCK_SIZE),
34.     },
35. };
```

5. 修改内核选上支持 jffs2 格式的选项，并重新编译内核
6. 烧录内核
   b. 将内核镜像放到 tftp 服务所指向的文件夹中
   c. 在串口连接端输入以下命令

```
tftp 0x80300000 uImage
nand unlock
nand ecc sw
nand erase 280000 300000
nand write.i 80300000 280000 300000
```

7. 烧录文件系统
   a) 将 jffs2 文件系统镜像放到 tftp 服务所指向的文件夹中
   b) 在串口连接端输入以下命令；用你的文件名，代替 XX，如 filesystem.jffs2 等，还有要你文件系统的大小（16 进制）代替 YY ，如：0x33e0838

```
tftp 0x81000000 XX.jffs2
nand unlock
nand ecc sw
nand erase 680000 0x5000000
nand write.jffs2 81000000 680000 YY
nand erase 5680000 0x2000000
```

7. 设置启动参数

```
setenv bootargs mem=128M console=ttyS2,115200n8 init=/init root=31:04 rootfstype=jffs2 video=omapfb:mode:4.3inch_LCD
setenv bootcmd nand read.i 80300000 280000 250000 \; bootm 80300000
saveenv
```

结束语

经过一个月的努力，项目基本上完全，能够在 Omap3530 上正常播放音、视频，并且实现了 USB Camera 预览功能。

但是由于时间较短，项目还有很多不如人意的地方，例如：USB Camera 的预览图像有些越界的现象，还有在关闭 Camera 预览的时候可能会出现内存错误，还有内存泄漏的问题，这些都是后续需要继续改进的

致谢：

在项目的一个月多里, 我受到了华清远见. 深圳中心 教学部老师细心指导, 老师严谨的指导态度与深厚的理论知识都让我受益非浅, 从他们身上我学到了很多的东西, 无论是理论还是实践都使我的知识有很大的提高. 借此我特提出感谢.

同时我还感谢华清远见. 深圳中心的其他老师，他们为我提供相应的硬件设备，软件环境及其它实验条件，让我顺利的完成了本项目开发，同时还要感谢同小组的成员及他给我提供帮助的同学，如果没有大家的努力，就没现在的成绩

参考文献

(1) 《Android 系统原理及开发要点详解》 韩超 梁泉 著 电子工业出版社

(2) 《Linux 设备驱动开发详解》 宋宝华 著 人民邮电出版社

(3) http://blog.csdn.net/zhanghuiliang/archive/2008/10/20/3110758.aspx

(4) http://www.rosoo.net/a/linux/201001/8382.html

(5) http://hi.baidu.com/xiaoyufen/blog/item/41108ad43a73a400a18bb7b0.html