目录

[1 Rk3308 简介： 4](#_Toc11425736)

[2使用adb连接板子 4](#_Toc11425737)

[2.1 adb安装 4](#_Toc11425738)

[2.2 使用putty连接板子 4](#_Toc11425739)

[2.3 上传下载文件到开发板 4](#_Toc11425740)

[2.4 更新内核 4](#_Toc11425741)

[3编译下载内核 5](#_Toc11425742)

[3.1 编译uboot(32位) 5](#_Toc11425743)

[3.2 编译内核 5](#_Toc11425744)

[3.3 编译驱动的Makefile（32位内核） 5](#_Toc11425745)

[3.4 板子匹配的设备树 5](#_Toc11425746)

[3.5 添加lcd驱动到内核 6](#_Toc11425747)

[3.6 编译应用的编译器(32位编译器) 6](#_Toc11425748)

[3.7 编译配置buildroot 6](#_Toc11425749)

[3.8 mkfirmware.sh生成固件 7](#_Toc11425750)

[3.9 parameter的配置和含义 8](#_Toc11425751)

[3.10 userdata.img生成 8](#_Toc11425752)

[3.11 开发板和UIOT开发板设备树区分 8](#_Toc11425753)

[3.12 设置应用开机启动 8](#_Toc11425754)

[3.13 关于userdata和data里面数据断电丢失问题 9](#_Toc11425755)

[3.14 修改Mic配置 9](#_Toc11425756)

[3.15 内核升级 9](#_Toc11425757)

[3.15.1配置支持升级系统 9](#_Toc11425758)

[3.15.2 生成update.img镜像 10](#_Toc11425759)

[3.15.3 升级系统 10](#_Toc11425760)

[3.15.4 恢复出厂设置 10](#_Toc11425761)

[3.15.5 升级注意事项 10](#_Toc11425762)

[3.15.6 UIOT板子升级 10](#_Toc11425763)

[3.16 安卓tool启动配置 11](#_Toc11425764)

[3.16 动态链接库配置和生成 11](#_Toc11425765)

[3.16.1 生成动态链接库 11](#_Toc11425766)

[3.16.2 添加新的bin路径和动态链接库路径 12](#_Toc11425767)

[4 GPIO口使用 13](#_Toc11425768)

[4.1 GPIO dts配置 13](#_Toc11425769)

[4.2 将IO口导入到用户控件 13](#_Toc11425770)

[4.2.1 命令行导出io口 14](#_Toc11425771)

[4.2.2 在应用程序中操作 14](#_Toc11425772)

[5 spi 接口Lcd驱动 15](#_Toc11425773)

[5.1 设计思路 15](#_Toc11425774)

[5.2 硬件信息 15](#_Toc11425775)

[4.2.1 rk3308 SPI功能 15](#_Toc11425776)

[5.2.2配置内核支持瑞芯微spi 16](#_Toc11425777)

[5.2.3 lcd 屏接口模式配置 16](#_Toc11425778)

[5.2.4 lcd屏spi接口引脚描述 16](#_Toc11425779)

[5.2 dts配置 16](#_Toc11425780)

[5.3 驱动例程 17](#_Toc11425781)

[5.4 测试 25](#_Toc11425782)

[6 触摸屏ft6236驱动 26](#_Toc11425783)

[6.1 设计思路 26](#_Toc11425784)

[6.2 配置dts 26](#_Toc11425785)

[6.3驱动例程 27](#_Toc11425786)

[6.4 测试 27](#_Toc11425787)

[7 配置minigui支持 27](#_Toc11425788)

[8 单独移植minigui 28](#_Toc11425789)

[9移植openssl和curl 31](#_Toc11425790)

[10 显示logo 32](#_Toc11425791)

[10.1 uboot显示logo 32](#_Toc11425792)

[10.3 测试uboot的spi驱动是否可用 33](#_Toc11425793)

[10.2 内核显示logo 34](#_Toc11425794)

[11 裁剪内存 34](#_Toc11425795)

[12 PWM背光 35](#_Toc11425796)

[12.1 配置内核支持pwm背光驱动 35](#_Toc11425797)

[12.2 配置pwm背光设备节点 35](#_Toc11425798)

[12.2.3 背光驱动 36](#_Toc11425799)

[13 wifi 配置 41](#_Toc11425800)

[13.1配置设备树 41](#_Toc11425801)

[13.2 配置内核 41](#_Toc11425802)

[13.3 命令行配网 41](#_Toc11425803)

[13.3.1 buildroot选择配置文件 41](#_Toc11425804)

[13.3.2 串口终端输入 41](#_Toc11425805)

[13.4 手机配网 42](#_Toc11425806)

[13.5 iw工具使用 43](#_Toc11425807)

[13.6 wpa\_supplicant 使用 43](#_Toc11425808)

[13.5 驱动源码 44](#_Toc11425809)

[13.6 rtl8189f 功能引脚 45](#_Toc11425810)

[13.7 调试遇到问题 45](#_Toc11425811)

[13.8 wifi连接断开监测 45](#_Toc11425812)

[14 高精度定时器hrtimer 45](#_Toc11425813)

[14.1 hrtimer函数的简单介绍 45](#_Toc11425814)

[14.2 hrtimer结构 46](#_Toc11425815)

[14.3 hrtimer使用 47](#_Toc11425816)

[15 红外感应开关 47](#_Toc11425817)

[15.1 分离器件电路 47](#_Toc11425818)

[15.1.1 设计思路 47](#_Toc11425819)

[15.1.2 设备树配置 48](#_Toc11425820)

[15.2 采用集成红外芯片 48](#_Toc11425821)

[15.2.1 设计思路 48](#_Toc11425822)

[15.2.2 dts配置和电路图 48](#_Toc11425823)

[15.2.3 驱动代码 48](#_Toc11425824)

[15.2.4 应用测试代码 52](#_Toc11425825)

[15.2.5 测试 53](#_Toc11425826)

[16 ADC按键驱动 53](#_Toc11425827)

[16.1 配置内核支持ADC和adc-keys驱动 54](#_Toc11425828)

[16.2 配置设备树 54](#_Toc11425829)

[16.3 linux IIO子系统使用 56](#_Toc11425830)

[接口说明 56](#_Toc11425831)

[16.4 ADC按键驱动分析 57](#_Toc11425832)

[16.3.1 驱动中配置设备树 57](#_Toc11425833)

[16.3.2 注册adc\_keys驱动 57](#_Toc11425834)

[16.3.3 probe函数 58](#_Toc11425835)

[16.4 应用层测试程序 59](#_Toc11425836)

[16.5 自定义reset键码值添加 59](#_Toc11425837)

[17 串口驱动 59](#_Toc11425838)

[17.1 配置内核 59](#_Toc11425839)

[17.2 配置设备树 60](#_Toc11425840)

[17.3 串口配置 60](#_Toc11425841)

[17.3.1 串口名字 60](#_Toc11425842)

[17.3.2 pinctrl配置 60](#_Toc11425843)

[17.3.2 DMA使用 60](#_Toc11425844)

[17.3.3波特率配置 60](#_Toc11425845)

[17.4 linux打印串口配置 60](#_Toc11425846)

[17.5 测试板子串口和ubuntu串口是否通 61](#_Toc11425847)

[17.6 通过串口实现GDB 调试 62](#_Toc11425848)

[18 音频驱动 62](#_Toc11425849)

[18.1 功率放大器 62](#_Toc11425850)

[18.2 CODEC配置 63](#_Toc11425851)

[18.2.1 UIOT板子codec配置 63](#_Toc11425852)

[18.2.2 rk3308 codec设备树配置知识 64](#_Toc11425853)

[18.2.3 Asound.conf 配置说明 66](#_Toc11425854)

[18.3 录音播放 70](#_Toc11425855)

[18.4 CODEC 音频增益调节 71](#_Toc11425856)

[18.5 静音控制驱动 75](#_Toc11425857)

[19 新建nandflash绝对地址分区 78](#_Toc11425858)

[20 音频学习 78](#_Toc11425859)

[20.1 ALSA 学习 78](#_Toc11425860)

[20.1.1 ALSA名词解释 78](#_Toc11425861)

[20.1.1 ALSA测试声卡 82](#_Toc11425862)

[20.2 ALSA音频工具 mixer,aplay,arecord 83](#_Toc11425863)

[20.2.1声音录制命令arecord 83](#_Toc11425864)

[20.3 tinyalsa 86](#_Toc11425865)

[20.4 alsa-lib应用层接口分析 88](#_Toc11425866)

[linux iio子系统 93](#_Toc11425867)

[错误解决 97](#_Toc11425868)

# 1 Rk3308 简介：

四核64位核心处理器，整合高性能CODEC音频编解码器，VAD。支持国内外多种人工智能及物联网操作系统，可配置6个麦克风阵列及POE扩展板。可用于智能音频方案、语音识别等项目。

**处理器核心：**cortex-a35核心配置，主频1.3GHZ,

**开源开发板：**ROC-RK3308-CC 是一款集多种功能为一体的AI+IOT开源主板。

**vad:(静音抑制)：**语音活动检测(Voice Activity Detection,VAD)又称语音端点检测,语音边界检测。目的是从声音信号流里识别和消除长时间的静音期，以达到在不降低业务质量的情况下节省[话路](https://baike.baidu.com/item/%E8%AF%9D%E8%B7%AF/5922604)资源的作用，它是IP电话应用的重要组成部分。静音抑制可以节省宝贵的[带宽](https://baike.baidu.com/item/%E5%B8%A6%E5%AE%BD/266879)资源，可以有利于减少用户感觉到的端到端的时延。

# 2使用adb连接板子

## 2.1 adb安装

下载adb安装包，然后将system32里面的cmd命令行拷贝到adb文件夹里面。打开cmd工具执行adb shell登陆到板子的根目录中。如果能进入根目录证明配置成功

## 2.2 使用putty连接板子

下载putty的adb版本，放入adb文件夹里面，打开选择adb模式，Host name里面填写transport-usb,然后点击open进入到板子的根目录。

## 2.3 上传下载文件到开发板

打开adb目录下的cmd命令行工具，不要执行adb shell 登陆到板子。

adb push <本地路径> <远程路径>

adb push "E:\hello.txt" "userdata/mytest/" //板子data区可读写，注意/是linux下，\window

//下使用。

adb pull <远程路径> <本地路径>

adb push "userdata/mytest/ hello.txt " "E:\ "

## 2.4 更新内核

瑞芯微开发工具软件下载，首先按住reset 插入usb线，当软件提示发现一个ADB

设备后点击切换按钮进入loader模式，点击执行开始下载。

该软件可以单独更新某个固件。如果不知道下载地址，那么可以读取设备内部的分区表，分区表就是nandflash存放uboot，kernel，rootfs等的分区信息：

GPT part: 0, name: uboot, start:0x2000, size:0x800

GPT part: 1, name: trust, start:0x2800, size:0x800

GPT part: 2, name: misc, start:0x3000, size:0x800

GPT part: 3, name: recovery, start:0x3800, size:0x6000

GPT part: 4, name: boot, start:0x9800, size:0x3000

GPT part: 5, name: rootfs, start:0xc800, size:0x11800

GPT part: 6, name: oem, start:0x1e000, size:0xe000

GPT part: 7, name: userdata, start:0x2c000, size:0x97df

# 3编译下载内核

## 3.1 编译uboot(32位)

source buildroot/build/envsetup.sh //选择3,32位debug版本

Cd uboot/

make ARCH=arm menuconfig

./make.sh evb-aarch32-rk3308

## 3.2 编译内核

source buildroot/build/envsetup.sh //选择3,32位debug版本

cd kernel

make ARCH=arm rk3308\_linux\_aarch32\_debug\_defconfig

make ARCH=arm menuconfig

make ARCH=arm savedefconfig

cp defconfig arch/arm/configs/rk3308\_linux\_aarch32\_debug\_defconfig

make ARCH=arm rk3308-voice-module-board-v10-aarch32.img //uiot使用这个

//make ARCH=arm rk3308-voice-module-amic-mainboard-v10-aarch32.img

## 3.3 编译驱动的Makefile（32位内核）

编译驱动的终端必须是配置内核的终端，而且使用root用户。

KERN\_DIR = /home/chao/rk3308\_linux/kernel

all:

make -C $(KERN\_DIR) M=`pwd` ARCH=arm modules //指定是32位

clean:

make -C $(KERN\_DIR) M=`pwd` ARCH=arm modules clean

rm -rf modules.order

obj-m += gpio\_keys.o

## 3.4 板子匹配的设备树

配置1：

**rk3308-voice-module-amic-mainboard-v10-aarch32.**

#include "rk3308-voice-module-v10-aarch32.dtsi"

#include "rk3308-voice-module-mainboard-v10-aarch32.dtsi"

32位dts也包含arm64/boot/dts/rk3308.dtsi

**配置2：rk3308-voice-module-board-v10-aarch32**

#include "rk3308-voice-module-v10-aarch32.dtsi"

## 3.5 添加lcd驱动到内核

以我编写的spilcd驱动为例

1. driver/char/目录下新建目录spilcd
2. 把spilcd.c拷贝到spilcd目录里面
3. 创建Makefile和Kconfig两个文件//Kconfig用于图像配置，Makefile用于编译
4. Kconfig内容如下：

menu "RK3308 SPILCD"

config RK3308\_SPILCD

tristate "the driver of spilcd for rk3308"

depends on CPU\_RK3308 //表示CPU\_RK3308选中才能配置该选项

default Y

help

this is a driver for spi lcd

endmenu

1. 修改Makefile添加如下行

obj-$(CONFIG\_RK3308\_SPILCD) += spilcd.o

// CONFIG\_RK3308\_SPILCD就是前面的config RK3308\_SPILCD 中的RK3308\_SPILCD加上//CONFIG\_

1. 修改Drivers/char目录下面的Kconfig

source "drivers/char/spilcd/Kconfig"//建立和子目录的联系

endmenu

1. 修改Drivers/char目录下面的Makefile,在最后添加

obj-$(CONFIG\_RK3308\_SPILCD) += spilcd/ //建立和子目录的联系

## 3.6 编译应用的编译器(32位编译器)

/home/chao/rk3308\_linux/buildroot/output/rockchip\_rk3308\_32\_release/host/bin/arm-rockchip-linux-gnueabihf-gcc

## 3.7 编译配置buildroot

source buildroot/build/envsetup.sh

选择3,32位调试版本

cd buildroot/output/

cd rockchip\_rk3308\_32\_debug/

make menuconfig

make savedefconfig

make //编译buildroot

## 3.8 mkfirmware.sh生成固件

需要配置device/rockchip/BoardConfig.mk文件。BoardConfig.mk是mkfirmware.sh执行时的环境变量配置文件，里面包括了生成img文件的源文件路径。

可以复制BoardConfig\_32bit.mk为BoardConfig.mk然后再更改

下面是生成32位系统debug版系统镜像的配置：

#!/bin/bash

# Target arch

export RK\_ARCH=arm

# Uboot defconfig

export RK\_UBOOT\_DEFCONFIG=evb-aarch32-rk3308

# Kernel defconfig

export RK\_KERNEL\_DEFCONFIG=rk3308\_linux\_aarch32\_debug\_defconfig

# Kernel dts

export RK\_KERNEL\_DTS=rk3308-voice-module-amic-mainboard-v10-aarch32

# boot image type

export RK\_BOOT\_IMG=zboot.img

# parameter for GPT table

export RK\_PARAMETER=parameter-32bit.txt

# Buildroot config

export RK\_CFG\_BUILDROOT=rockchip\_rk3308\_32\_debug

# Recovery config

export RK\_CFG\_RECOVERY=rockchip\_rk3308\_recovery

# Pcba config

export RK\_CFG\_PCBA=rockchip\_rk3308\_pcba

# Build jobs

export RK\_JOBS=12

# target chip

export RK\_TARGET\_PRODUCT=rk3308

# Set rootfs type, including ext2 ext4 squashfs

export RK\_ROOTFS\_TYPE=squashfs

# rootfs image path

export RK\_ROOTFS\_IMG=buildroot/output/$RK\_CFG\_BUILDROOT/images/rootfs.$RK\_ROOTFS\_TYPE

# Set oem partition type, including ext2 squashfs

export RK\_OEM\_FS\_TYPE=ext2

# Set userdata partition type, including ext2, fat

export RK\_USERDATA\_FS\_TYPE=ext2

# Set flash type. support <emmc, nand, spi\_nand, spi\_nor>

export RK\_STORAGE\_TYPE=nand

#OEM config: /oem/dueros/aispeech-6mic-64bit/aispeech-2mic-64bit/aispeech-4mic-32bit/aispeech-2mic-32bit/aispeech-2mic-kongtiao-32bit/iflytekSDK/CaeDemo\_VAD/smart\_voice

export RK\_OEM\_DIR=oem//表示不打包算法程序

#userdata config

export RK\_USERDATA\_DIR=userdata\_empty //定义userdata.img源路径，此处为不生成。

MIC\_NUM=2

## 3.9 parameter的配置和含义

FIRMWARE\_VER:8.1

MACHINE\_MODEL:RK3308

MACHINE\_ID:007

MANUFACTURER: RK3308

MAGIC: 0x5041524B

ATAG: 0x00200800

MACHINE: 3308

CHECK\_MASK: 0x80

PWR\_HLD: 0,0,A,0,1

TYPE: GPT

CMDLINE:mtdparts=rk29xxnand:0x00000800@0x00002000(uboot),0x00000800@0x00002800(trust),0x00000800@0x00003000(misc),0x00006000@0x00003800(recovery),0x00003000@0x00009800(boot),0x00015000@0x0000C800(rootfs),0x0000E000@0x00021800(oem),-@0x0002F800(userdata:grow)

uuid:rootfs=614e0000-0000-4b53-8000-1d28000054a9

parameter是用来给nandflash分区的，单位是512子节。0x00000800@0x00002000(uboot)就是设置uboot分区的。起始地址在0x00002000\*512子节处，大小为0x00000800\*512B。

其中绿色部分为更改后的值，因为文件系统较大，原来预留的空间不足，修改增大了空间。

否则会导致rootfs.img下载失败。

## 3.10 userdata.img生成

Mkfirmware.sh调用 device/rockchip/common里面的mk-userdata.sh生成镜像。镜像的源文件位置device/rockchip/userdata里面。该路径是由device/rockchip/rk3308/BoardConfig.mk里面指示。

必须下载该镜像，否则会报错。

## 3.11 开发板和UIOT开发板设备树区分

rk3308-voice-module-v10-aarch32.dtsi中配置有如下宏定义则使用UIOT板子。

#define UIOT 1

## 3.12 设置应用开机启动

Oem的源文件在目录

/home/chao/rk3308\_linux/device/rockchip/rk3308/oem

修改oem/ RkLunch.sh

wpa\_supplicant -B -i wlan0 -c /data/cfg/wpa\_supplicant.conf

#./wakeWordAgent -e gpio &

sleep 5

/oem/myapp/test &

sleep 30 //启动mqtt需要wifi已经连接，需要时间较长，否则

/oem/myapp/subcallback & //mqtt应用程序

## 3.13 关于userdata和data里面数据断电丢失问题

由于文件系统机制问题，缓存在内存，并不是立即写入磁盘。因此需要执行sync命令再断电。

## 3.14 修改Mic配置

/home/chao/rk3308\_linux/buildroot/board/rockchip/rk3308/fs-overlay/etc/asound.conf

注释部分为原来的。

pcm.multi\_2\_2 {

type multi

slaves.a.pcm "hw:0,0"

slaves.a.channels 4

bindings.0.slave a

bindings.0.channel 0

bindings.1.slave a

bindings.1.channel 1

bindings.2.slave a

# bindings.2.channel 2

bindings.2.channel 3

bindings.3.slave a

# bindings.3.channel 3

bindings.3.channel 2

}

pcm.2mic\_loopback

{

type plug

# slave.pcm "multi\_2\_1"

slave.pcm "multi\_2\_2"

}

## 3.15 内核升级

### 3.15.1配置支持升级系统

rootfs 要打开 recoverySystem 的支持， configs 文件中把  
BR2\_PACKAGE\_RECOVERYSYSTEM=y 选上。  
或者配置 BR2\_PACKAGE\_UPDATE=y

[\*] Rockchip OTA update for linux 选上

或者

[\*]Rockchip recoverySystem for linux

位置在buildroot 配置

Location:

-> Target packages

-> rockchip BSP packages (BR2\_PACKAGE\_ROCKCHIP [=y])

系统根目录下执行

$ ./build.sh recovery

生成recovery.img

如果编译过程出现fakeroot.so找不到删除rockchip\_rk3308\_recovery然后再执行

上面的命令

$ ./mkfirmware.sh  
会将生成的固件拷贝至 rockdev/目录下

### 3.15.2 生成update.img镜像

把镜像放在AndroidTool\rockdev\Image文件夹里面

修改AndroidTool\rockdev\package-file文件配置升级那些文件

执行AndroidTool\rockdev下面的mkupdate脚本生成update.img

注意升级时参数必须和原来分区一致

升级文件：boot,MiniLoaderAll.bin,parameter turst uboot.

### 3.15.3 升级系统

把生成的upadte.img放到/userdata/分区里面然后执行命令：

recoverySystem ota /userdata/update.img

recoverySystem ota /oem/update.img

### 3.15.4 恢复出厂设置

我们把可以读写的配置文件保存在 userdata 分区， 出厂固件会默认一些配置参数， 用户使用一段时间后会生成或修改配置文件， 有时用户需要清除这些数据， 我们就需要恢复到出厂配置

直接运行 recoverySystem （或 update） 后面不加任何参数或者加 factory/reset 参数均可进入 recovery 后恢复出厂配置。

### 3.15.5 升级注意事项

1>打包 update.img 固件时需要注意， 升级固件不一定要全分区升级， 可修改 package-file 文件， 将不要升级的分区去掉， 这样可以减少升级包（update.img ） 的大小。

2> package-file 中 **recovery.img** 如果打包进去的话， 不会在 **recovery** 模式中升级

3> **misc** 分区不建议打包进 **update.img** 中， 即使有打包进去， 也会在升级程序中加载判断到而忽略该分区， 原因是： 即使升级了 misc 分区， 升级成功后 recovery 程序仍会清空 misc 分区中所有的命令及参数， 从而导致预想的结果达不到

4>如果将 **update.img** 升级包放置在 **flash** 中的 **userdata** 分区， 则需要保证 **package-file** 中  
不包括 **userdata.img** 被打包进去， 原因是可能会导致文件系统的损坏， 升级成功后可能使 oem或 userdata 分区 mount 不成功。 若从 SD 卡或 U 盘升级时， 可以打包 userdata.img， 从而对userdata 分区进行升级。 升级完成后会对 userdata 分区重新 resize 操作。

### 3.15.6 UIOT板子升级

由于UIOT板子nandflash空间较小，故不升级文件系统。如果有新库和新的命令行，那么放入oem里面的一个文件内，连接此文件夹。

3.15.7 修改S98\_lunch\_init删除

/home/chao/rk3308\_linux/buildroot/board/rockchip/rk3308/fs-overlay/etc/init.d

### 3.16 安卓tool启动配置

在安卓tool工具空白处右击弹出菜单，点击导出配置，命名为myconfig

打开config配置文件修改以下部分

#指定启动时加载的镜像配置文件,默认为config.cfg

DEFAULT\_IMAGE\_CONFIG=myconfig.cfg

## 3.16 动态链接库配置和生成

### 3.16.1 生成动态链接库

编写库源码hello.c，bye.c

vi hello.c

#include <stdio.h>

void hello(void)

{

printf("hello world\n");

return;

}

vi bye.c

#include <stdio.h>

void bye(void)

{

printf("bye!\n");

return ;

}

为共享库编写头文件common.h

#ifndef \_\_COMMON\_H\_\_

#define \_\_COMMON\_H\_\_

void hello(void);

void bye(void);

#endif

编译生成目标文件

arm-rockchip-linux-gnueabihf-gcc -c hello.c bye.c -fPIC -Wall

ls 查看生成的.o文件

bye.c bye.o common.h hello.c hello.o

-fPIC选项：告诉编译器生成位置无关代码

位置无关代码：生成的".o文件"文件中的代码可以被加载到任意的地址执行。编译的时候用到了相对寻址而不是绝对寻址

创建共享库common

arm-rockchip-linux-gnueabihf-gcc -shared hello.o bye.o -o libcommon.so.1

ls

bye.c bye.o common.h hello.c hello.o libcommon.so.1

-shared选项：告诉编译器生成一个共享库

生成的共享库的文件名叫"libcommon.so.1",其中".so"表示这是一个共享库，".1"表示这个库的版本是1

为共享库文件创建链接文件

ln -s libcommon.so.1 libcommon.so

共享库的使用

编写测试程序test.c

#include <stdio.h>

#include "common.h"

int main(int argc, const char \*argv[])

{

hello();

bye();

return 0;

}

[编译test.c并链接共享库libcommon.so](http://xn--test-pj6ky44d.xn--clibcommon-hy3ph0y7w9bxla126dcj0q.so/)

arm-rockchip-linux-gnueabihf-gcc test.c -o sharetest -L. -lcommon

ls

bye.c bye.o common.h hello.c hello.o libcommon.so libcommon.so.1 sharetest test.c

把生成的库放到/oem/sharelib里面。把生成的sharetest文件放到/oem/bin目录里面。

### 3.16.2 添加新的bin路径和动态链接库路径

修改

/home/chao/rk3308\_linux/buildroot/board/rockchip/rk3308/fs-overlay/etc/profile.d/目录

创建文件 profile.sh添加如下行

#!/bin/bash

export PATH=$PATH:/oem/bin

export LD\_LIBRARY\_PATH=$LD\_LIBRARY\_PATH:/oem/sharelib

profile文件有如下内容会自动执行profile.d里面的所有脚本

# Source configuration files from /etc/profile.d

for i in /etc/profile.d/\*.sh ; do

if [ -r "$i" ]; then

. $i

fi

unset i

done

在RkLunch.sh里面添加

source /etc/profile这样RkLunch.sh和其子路径都可以使用该配置

测试

在终端输入sharetest测试程序可执行证明终端可以找到库路径

/oem/shell/profile.sh 打印输出证明子shell脚本执行应用可以找到库路径

sharetest 本脚本执行应用可以正确找到库路径

# 4 GPIO口使用

## 4.1 GPIO dts配置

对引脚的描述与Firefly-RK3288有所区别，GPIO0\_B4被描述为：<&gpio0 12 GPIO\_ACTIVE\_HIGH>，这里的12来源于：8+4=12，其中8是因为GPIO0\_B4是属于GPIO0的B组，如果是A组的话则为0，如果是C组则为16，如果是D组则为24，以此递推，而4是因为B4后面的4。 GPIO\_ACTIVE\_HIGH表示高电平有效，如果想要低电平有效，可以改为：GPIO\_ACTIVE\_LOW，这个属性将被驱动所读取。

Rk3308 dts:

一般按键引脚配置：

gpio-keys {

compatible = "gpio-keys";

autorepeat;

pinctrl-names = "default";

pinctrl-0 = <&pwr\_key>;

power {

//gpio0 A组的第6个引脚

gpios = <&gpio0 RK\_PA6 GPIO\_ACTIVE\_LOW>;

linux,code = <KEY\_POWER>;

label = "GPIO Key Power";

wakeup-source;

debounce-interval = <100>;

};

};

&pinctrl {

pinctrl-names = "default";

pinctrl-0 = <&rtc\_32k>;

buttons {

pwr\_key: pwr-key {

rockchip,pins = <0 RK\_PA6 RK\_FUNC\_GPIO &pcfg\_pull\_up>;

};

};

};

};

## 4.2 将IO口导入到用户控件

**内核配置支持用户控件io：**

device drivers -> gpio support ->/sys/class/gpio.....(sysfs interface)

### 4.2.1 命令行导出io口

cd /sys/class/gpio/

echo 53 > export // pinnum就是引脚的统一编号，计算方法看4.1节

ls 可以看到多了gpio53

cd gpio53/

echo "in" > direction     -----------输入方向

echo "out" > direction    -----------输出方向

获取：（在direction 为 in的情况下）

cat value

0 ----------引脚电平为低

1 ----------引脚电平为高

控制：（在direction为out的情况下）

echo 1 > value  -------输出高电平

echo 0 > value  -------输出低电平

### 4.2.2 在应用程序中操作

{

char value;

int export\_fd, direction\_fd, gpiovalue\_fd;

export\_fd = open("/sys/class/gpio/export", O\_WRONLY);

if(-1 == export\_fd)

{

printf("[%s]:[%d] open gpio export file error\r\n", \_\_FUNCTION\_\_, \_\_LINE\_\_);

exit(1);

}

if(-1 == write(export\_fd, "7", 1))

{

printf("write file operation error\r\n");

exit(1);

}

close(export\_fd);

//sleep(2000);

direction\_fd = open("/sys/class/gpio/gpio7/direction", O\_WRONLY);

if(-1 == direction\_fd)

{

printf("[%s]:[%d] open gpio direction file error\r\n", \_\_FUNCTION\_\_, \_\_LINE\_\_);

exit(1);

}

if(-1 == write(direction\_fd, "in", sizeof("in")))

{

printf("[%s]:[%d] write operation direction error\r\n", \_\_FUNCTION\_\_, \_\_LINE\_\_);

close(direction\_fd);

exit(1);

}

close(direction\_fd);

printf("[%s]:[%d] debug information print\r\n", \_\_FUNCTION\_\_, \_\_LINE\_\_);

gpiovalue\_fd = open("/sys/class/gpio/gpio7/value", O\_RDONLY);

if(-1 == gpiovalue\_fd)

{

printf("[%s]:[%d]: open value file error\r\n", \_\_FUNCTION\_\_, \_\_LINE\_\_);

exit(1);

}

while(1)

{

if(-1 == read(gpiovalue\_fd, &value, sizeof(value)))

{

printf("[%s]:[%d] read gpiovalue is fail\r\n", \_\_FUNCTION\_\_, \_\_LINE\_\_);

close(gpiovalue\_fd);

exit(1);

}

if('0' == value)

{

printf("now, the reset key is pressing and the key value = %c !!!!\r\n", value);

lseek(gpiovalue\_fd, 0, SEEK\_SET);

sleep(1);

}

close(gpiovalue\_fd);

}

# 5 spi 接口Lcd驱动

## 5.1 设计思路

使用帧缓冲区来操作，驱动中申请缓冲区，应用层映射该缓冲区，应用层像缓冲区中写入数据，驱动每隔30ms读取缓冲区数据发送到lcd设备。定时时间间隔发送采用延时工作队列delayed\_worker。先编程实现在硬件驱动中刷新lcd,然后实现支持framebuffer的驱动，从应用层测试。

## 5.2 硬件信息

### 4.2.1 rk3308 SPI功能

摩托罗拉SPI协议

支持8位和16位。

时钟频率50MHZ

SPI 4中传输模式配置

### 5.2.2配置内核支持瑞芯微spi

Device Drivers ‐‐‐>  
[\*] SPI support ‐‐‐>  
<\*> Rockchip SPI controller driver

### 5.2.3 lcd 屏接口模式配置

接口类型总共有两种：8080 MCU接口和串口。

通过外部IM[3:0]引脚来控制。1110模式下配置为四线spi接口，SCL,SDI,D/CX,SDO,CS。 1101模式下配置为3线9位接口，SCL,SDI,SDO,CSX。 0101模式下三线9位串行接口，SCL,SDA,CSX, SDA线同时作为输入和输出口。0110四线8位模式，SCL,SDA,D/CX,CSX,SDA线同时作为输入和输出口。

原理图采用的是三线9位接口。Rk3308只能支持四线8位。

### 5.2.4 lcd屏spi接口引脚描述

CSX :片选信号，低电平有效。

WRX(D/CX) :四线系统命令数据选择。低电平表示命令，高电平表示数据。

D/CX(SCL):串行时钟

SDI/SDA:四线输入信号，三线输入输出信号。主机master输出数据端口，从设备输入端口

SDO: 四线输出信号。主机master输入数据端口，从设备输出端口

位传输顺序：最高位先传输。9位模式下，D/CX先传输。

主机通过SDA发送读命令后，芯片会在随后的时钟位发送数据到SDA。

## 5.2 dts配置

直接修改/arch/arm64/boot/dts/rk3308.dtsi

spi2: spi@ff140000 {

compatible = "rockchip,rk3308-spi", "rockchip,rk3066-spi";

reg = <0x0 0xff140000 0x0 0x1000>;

interrupts = <GIC\_SPI 17 IRQ\_TYPE\_LEVEL\_HIGH>;

#address-cells = <1>;

#size-cells = <0>;

clocks = <&cru SCLK\_SPI2>, <&cru PCLK\_SPI2>;

clock-names = "spiclk", "apb\_pclk";

max-freq = <80000000>;

dmas = <&dmac1 16>, <&dmac1 17>;

dma-names = "tx", "rx";

pinctrl-names = "default", "high\_speed";

pinctrl-0 = <&spi2\_clk &spi2\_csn0 &spi2\_miso &spi2\_mosi>;

pinctrl-1 = <&spi2\_clk\_hs &spi2\_csn0 &spi2\_miso\_hs &spi2\_mosi\_hs>;

status = "okay";

spilcd@0{

status="okay";

compatible="spilcd";

reg=<0>;

spi-max-frequency=<40000000>;

reset-gpios=<&gpio0 RK\_PC4 GPIO\_ACTIVE\_LOW>;

dc-gpios=<&gpio0 RK\_PC1 GPIO\_ACTIVE\_LOW>;

};

};

引用修改：

rk3308-voice-module-mainboard-v10-aarch32.dtsi中引用rk3308.dtsi

rk3308保持官方文件不变。只把需要更改的重新设置。这样就更改了rk3308.dtsi中的设定值。这样做的好处是，由于多个板子公用rk3308.dtsi文件，那么改变了rk3308.dtsi就改变了所有板子的配置。这不是我们想要的，我们只想更改我们自己的板子配置。

&spi2{

status = "okay";

max-freq = <48000000>;

spilcd@0{

status = "okay";

compatible = "spilcd";

reg = <0>;

spi-max-frequency = <12000000>;

#ifdef UIOT

dc-gpios = <&gpio1 RK\_PC1 GPIO\_ACTIVE\_LOW>;

reset-gpios = <&gpio1 RK\_PC2 GPIO\_ACTIVE\_LOW>;

#else

dc-gpios = <&gpio0 RK\_PC1 GPIO\_ACTIVE\_LOW>;

reset-gpios = <&gpio0 RK\_PC4 GPIO\_ACTIVE\_LOW>;

#endif

};

};

## 5.3 驱动例程

#include <linux/init.h>

#include <linux/fs.h>

#include <linux/slab.h>

#include <linux/module.h>

#include <linux/kernel.h>

#include <linux/device.h>

#include <sound/core.h>

#include <linux/spi/spi.h>

#include <asm/uaccess.h>

#include <linux/gpio.h>

#include <linux/of.h>

#include <linux/of\_gpio.h>

#include <linux/fb.h>

#include <linux/dma-mapping.h>

#include <linux/delay.h>

#include "spilcd.h"

#include "logo.h"

struct spilcd\_data{

int dc\_pin;

int reset\_pin;

int cs\_pin;

int clk\_pin;

int rx\_pin;

int tx\_pin;

};

u8 \*colorbuf;

struct spilcd{

struct spi\_device \*spidev;

struct fb\_info \*fbi;

struct spilcd\_data \*data;

struct delayed\_work delayed\_worker;

};

struct spilcd \*spilcd;

static int major;

static struct class \*class;

static struct fb\_ops spilcd\_fb\_ops = {

.owner = THIS\_MODULE,

};

//所谓的命令实际上就是寄存器地址

static void spilcd\_WriteCmd(unsigned char cmd)

{

gpio\_set\_value(spilcd->data->dc\_pin, 0);

spi\_write(spilcd->spidev, &cmd, 1);

gpio\_set\_value(spilcd->data->dc\_pin,1);

}

static void spilcd\_WriteData(unsigned char dat)

{

gpio\_set\_value(spilcd->data->dc\_pin, 1);

spi\_write(spilcd->spidev, &dat, 1);

gpio\_set\_value(spilcd->data->dc\_pin, 0);

}

static void spilcd\_init(struct spi\_device \*spi)

{

msleep(50);

spilcd\_WriteCmd(0x01); //soft reset

msleep(50);

spilcd\_WriteCmd(0x28); //display off

spilcd\_WriteCmd(0xCF); // 功耗控制器，貌似没有用,EXTC引脚高才能使用

spilcd\_WriteData(0x00);

spilcd\_WriteData(0xC1);

spilcd\_WriteData(0X30);

spilcd\_WriteCmd(0xED); //电源序列控制EXTC引脚高才能使用

spilcd\_WriteData(0x64);

spilcd\_WriteData(0x03);

spilcd\_WriteData(0X12);

spilcd\_WriteData(0X81);

spilcd\_WriteCmd(0xE8); //驱动时序控制AEXTC引脚高才能使用

spilcd\_WriteData(0x85);

spilcd\_WriteData(0x10);

spilcd\_WriteData(0x7A);

spilcd\_WriteCmd(0xCB); // 功耗控制器A

spilcd\_WriteData(0x39);

spilcd\_WriteData(0x2C);

spilcd\_WriteData(0x00);

spilcd\_WriteData(0x34);

spilcd\_WriteData(0x02);

spilcd\_WriteCmd(0xF7); //泵比控制

spilcd\_WriteData(0x20);

spilcd\_WriteCmd(0xEA); //驱动时序控制B

spilcd\_WriteData(0x00);

spilcd\_WriteData(0x00);

spilcd\_WriteCmd(0xC0); //功耗控制1

spilcd\_WriteData(0x1B); //VRH[5:0]

spilcd\_WriteCmd(0xC1); //功耗控制2

spilcd\_WriteData(0x01); //SAP[2:0];BT[3:0]

spilcd\_WriteCmd(0xC5); //VCOM电压控制1

spilcd\_WriteData(0x30);

spilcd\_WriteData(0x30);

spilcd\_WriteCmd(0xC7); //VCOM 电压控制2

spilcd\_WriteData(0XB7);

spilcd\_WriteCmd(0x36);

//spilcd\_WriteData((1<<5)|(0<<6)|(1<<7)|(1<<3)); yuanlaide

spilcd\_WriteData((0<<7)|(1<<6)|(1<<5)|(1<<3)); //uiot ok

spilcd\_WriteCmd(0x3A);

spilcd\_WriteData(0x55); //设置颜色为16位色

spilcd\_WriteCmd(0xB1);

spilcd\_WriteData(0x00);

spilcd\_WriteData(0x1A);

spilcd\_WriteCmd(0xB6); // Display Function Control

spilcd\_WriteData(0x0A);

spilcd\_WriteData(0xA2);

spilcd\_WriteCmd(0xF2); // 3Gamma Function Disable

spilcd\_WriteData(0x00);

spilcd\_WriteCmd(0x26); //Gamma curve selected

spilcd\_WriteData(0x01);

spilcd\_WriteCmd(0xE0); //Set Gamma

spilcd\_WriteData(0x0F);

spilcd\_WriteData(0x2A);

spilcd\_WriteData(0x28);

spilcd\_WriteData(0x08);

spilcd\_WriteData(0x0E);

spilcd\_WriteData(0x08);

spilcd\_WriteData(0x54);

spilcd\_WriteData(0XA9);

spilcd\_WriteData(0x43);

spilcd\_WriteData(0x0A);

spilcd\_WriteData(0x0F);

spilcd\_WriteData(0x00);

spilcd\_WriteData(0x00);

spilcd\_WriteData(0x00);

spilcd\_WriteData(0x00);

spilcd\_WriteCmd(0XE1); //Set Gamma

spilcd\_WriteData(0x00);

spilcd\_WriteData(0x15);

spilcd\_WriteData(0x17);

spilcd\_WriteData(0x07);

spilcd\_WriteData(0x11);

spilcd\_WriteData(0x06);

spilcd\_WriteData(0x2B);

spilcd\_WriteData(0x56);

spilcd\_WriteData(0x3C);

spilcd\_WriteData(0x05);

spilcd\_WriteData(0x10);

spilcd\_WriteData(0x0F);

spilcd\_WriteData(0x3F);

spilcd\_WriteData(0x3F);

spilcd\_WriteData(0x0F);

spilcd\_WriteCmd(0x2B);

spilcd\_WriteData(0x00);

spilcd\_WriteData(0x00);

spilcd\_WriteData(0x01);

spilcd\_WriteData(0x3f);

spilcd\_WriteCmd(0x2A);

spilcd\_WriteData(0x00);

spilcd\_WriteData(0x00);

spilcd\_WriteData(0x00);

spilcd\_WriteData(0xef);

spilcd\_WriteCmd(0x11);

msleep(120);

spilcd\_WriteCmd(0x29);

}

static struct spilcd\_data \*spilcd\_probe\_dt(struct device \*dev)

{

struct spilcd\_data \*pdata;

struct device\_node \*node = dev->of\_node;

enum of\_gpio\_flags of\_flags;

if (!node) {

return ERR\_PTR(-EINVAL);

}

pdata = devm\_kzalloc(dev,sizeof(struct spilcd\_data), GFP\_KERNEL);

if (!pdata)

return ERR\_PTR(-ENOMEM);

if (of\_find\_property(node, "dc-gpios", NULL)) {

pdata->dc\_pin= of\_get\_named\_gpio\_flags(node, "dc-gpios", 0, &of\_flags);

}

if (of\_find\_property(node, "reset-gpios", NULL)) {

pdata->reset\_pin= of\_get\_named\_gpio\_flags(node, "reset-gpios", 0, &of\_flags);

}

gpio\_direction\_output(pdata->dc\_pin, 1);

gpio\_direction\_output(pdata->reset\_pin, 1);

msleep(50);

gpio\_set\_value(pdata->reset\_pin,0);

msleep(200);

gpio\_set\_value(pdata->reset\_pin,1);

msleep(100);

return pdata;

}

static struct fb\_info \*spilcd\_fb\_init(void)

{

struct fb\_info \*fbi;

fbi = framebuffer\_alloc(0, NULL);

strcpy(fbi->fix.id, "spilcd");

fbi->fix.smem\_len = 240\*320\*16/8;

fbi->fix.type = FB\_TYPE\_PACKED\_PIXELS;

fbi->fix.visual = FB\_VISUAL\_TRUECOLOR;

fbi->fix.line\_length = 320\*2;

fbi->var.xres = 320;

fbi->var.yres = 240;

fbi->var.xres\_virtual = 320;

fbi->var.yres\_virtual = 240;

fbi->var.bits\_per\_pixel = 16;

fbi->var.red.offset = 11;

fbi->var.red.length = 5;

fbi->var.green.offset = 5;

fbi->var.green.length = 6;

fbi->var.blue.offset = 0;

fbi->var.blue.length = 5;

fbi->var.activate = FB\_ACTIVATE\_NOW;

fbi->fbops = &spilcd\_fb\_ops;

fbi->screen\_size = 240\*320\*16/8;

fbi->screen\_base = dma\_alloc\_writecombine(NULL,fbi->fix.smem\_len, (dma\_addr\_t \*)&fbi->fix.smem\_start, GFP\_KERNEL);

register\_framebuffer(fbi);

return fbi;

}

void set\_logodata(struct fb\_info \*fbi)

{

u8 \*buf= (u8\*)(fbi->screen\_base);//内存基地址

u32 len=fbi->fix.smem\_len;

memcpy(buf,gImage\_logo,len);

}

void show\_fb(struct fb\_info \*fbi, struct spi\_device \*spi)

{

/\*DMA模式 now work ok\*/

u32 len;

u8 tmp;

u32 i;

len=fbi->fix.smem\_len;

if(colorbuf==NULL)

colorbuf=kzalloc(len,GFP\_KERNEL);

memcpy(colorbuf,fbi->screen\_base,len);

for(i=0;i<len;i=i+2)

{

tmp=colorbuf[i];

colorbuf[i]=colorbuf[i+1];

colorbuf[i+1]=tmp;

}

spilcd\_WriteCmd(0x2A);

spilcd\_WriteData(0);

spilcd\_WriteData(0);

spilcd\_WriteData(0X01);

spilcd\_WriteData(0X3F);

spilcd\_WriteCmd(0X2B);

spilcd\_WriteData(0);

spilcd\_WriteData(0);

spilcd\_WriteData(0);

spilcd\_WriteData(0Xef);

spilcd\_WriteCmd(0X2C);

gpio\_set\_value(spilcd->data->dc\_pin, 1);

spi\_write(spilcd->spidev,(u8\*)colorbuf,len);

gpio\_set\_value(spilcd->data->dc\_pin, 0);

}

static void delayed\_work\_handler(struct work\_struct \*work)

{

//set\_fbdata();

show\_fb(spilcd->fbi,spilcd->spidev);

schedule\_delayed\_work(&spilcd->delayed\_worker,HZ/30);

}

static int spi\_lcd\_probe(struct spi\_device \*spi)

{

struct spilcd\_data \*pdata;

struct fb\_info \*fbi;

struct device \*dev;

int ret;

dev=&spi->dev;

pdata = dev->platform\_data;

if (!pdata) {

pdata=spilcd\_probe\_dt(dev);

if (IS\_ERR(pdata))

{

ret = PTR\_ERR(pdata);

return ret;

}

}

spilcd=devm\_kzalloc(dev,sizeof(struct spilcd), GFP\_KERNEL);

if (!spilcd){

return -ENOMEM;

}

spilcd->spidev=spi;

spilcd->data=pdata;

spilcd\_init(spi);

fbi=spilcd\_fb\_init();

spilcd->fbi=fbi;

set\_logodata(spilcd->fbi);

INIT\_DELAYED\_WORK(&spilcd->delayed\_worker,delayed\_work\_handler);

schedule\_delayed\_work(&spilcd->delayed\_worker,HZ/30);

return 0;

}

static int spi\_lcd\_remove(struct spi\_device \*spi)

{

struct fb\_info \*fbi;

fbi = spilcd->fbi;

cancel\_delayed\_work\_sync(&spilcd->delayed\_worker);

unregister\_framebuffer(fbi);

dma\_free\_writecombine(NULL, fbi->fix.smem\_len, fbi->screen\_base, fbi->fix.smem\_start);

framebuffer\_release(fbi);

kfree(colorbuf);

return 0;

}

static const struct of\_device\_id dt\_ids[] = {

{ .compatible = "ilitek,ili9341", },

{},

};

MODULE\_DEVICE\_TABLE(of, dt\_ids);

static struct spi\_driver spi\_lcd\_drv = {

.driver = {

.name = "spilcd",

.owner = THIS\_MODULE,

.of\_match\_table = of\_match\_ptr(dt\_ids),

},

.probe = spi\_lcd\_probe,

.remove = spi\_lcd\_remove,

};

static long spilcd\_ioctl(struct file \*filp, unsigned int cmd, unsigned long arg)

{

printk("lcd ioctl function\n");

switch(cmd){

case SPILCD\_ON:

spilcd\_WriteCmd(0x29);

schedule\_delayed\_work(&spilcd->delayed\_worker,HZ/30);

printk("open lcd\n");

break;

case SPILCD\_OFF:

cancel\_delayed\_work\_sync(&spilcd->delayed\_worker);

spilcd\_WriteCmd(0x28); //display off

printk("close lcd\n");

break;

default:

break;

}

return 0;

}

static int spilcd\_open(struct inode \*inode, struct file \*filp)

{

return 0;

}

static int spilcd\_release(struct inode \*inode, struct file \*filp)

{

return 0;

}

static const struct file\_operations spilcd\_fops = {

.owner = THIS\_MODULE,

.open = spilcd\_open,

.unlocked\_ioctl = spilcd\_ioctl,

.release = spilcd\_release,

};

static int \_\_init spi\_lcd\_init(void)

{

major = register\_chrdev(0, "spilcd", &spilcd\_fops);

class = class\_create(THIS\_MODULE, "spilcd");

device\_create(class, NULL, MKDEV(major, 0), NULL, "spilcd");

return spi\_register\_driver(&spi\_lcd\_drv);

}

static void \_\_exit spi\_lcd\_exit(void)

{

spi\_unregister\_driver(&spi\_lcd\_drv);

device\_destroy(class, MKDEV(major, 0));

class\_destroy(class);

unregister\_chrdev(major, "spilcd");

}

module\_init(spi\_lcd\_init);

module\_exit(spi\_lcd\_exit);

MODULE\_DESCRIPTION("SPI lcd Driver");

MODULE\_LICENSE("GPL");

## 5.4 测试

打开adb模式cmd命令行，在没有登录板子情况下执行：

adb push “myfb.ko” “tmp”

adb push “HZK16” “tmp”

adb push “show\_font” “tmp”

连接调试串口

adb shell 登录到板子

cd /tmp/

insmod myfb.ko

chmod 777 show\_font

./show\_font 发现板子上面显示字符A和汉字中

# 6 触摸屏ft6236驱动

## 6.1 设计思路

Ft6236有数据时会产生中断。然后通过i2c接口读取数据。

## 6.2 配置dts

引用配置

rk3308-voice-module-mainboard-v10-aarch32.dtsi中引用rk3308.dtsi

&i2c3{

status = "okay";

ft6236:ft6236@0x38{

compatible = "ft6236";

status = "okay";

reg = <0x38>;

touchscreen-size-x=<240>;

touchscreen-size-y=<320>;

interrupts = <&gpio0 RK\_PC2 IRQ\_TYPE\_EDGE\_RISING>;

reset-gpios=<&gpio2 RK\_PA3 GPIO\_ACTIVE\_LOW>;

};

};

rK3308.dtsi中i2c3配置，不修改

i2c3: i2c@ff070000 {

compatible = "rockchip,rk3399-i2c";

reg = <0x0 0xff070000 0x0 0x1000>;

clocks = <&cru SCLK\_I2C3>, <&cru PCLK\_I2C3>;

clock-names = "i2c", "pclk";

interrupts = <GIC\_SPI 14 IRQ\_TYPE\_LEVEL\_HIGH>;

pinctrl-names = "default";

pinctrl-0 = <&i2c3m0\_xfer>;

#address-cells = <1>;

#size-cells = <0>;

status = "disabled";

};

## 6.3驱动例程

发现会有一定几率出现误触发问题，决定增加简单数据滤波。

根据误触发打印的坐标点，一般一次只有一两个点。而每次触摸至少5个点，而且每次按下时间超过10ms，因此可以采用按键的延时消抖策略。

## 6.4 测试

Insmod ft6236.ko

点击触摸屏会打印坐标点数据。

# 7 配置minigui支持

make menuconfig

搜索minigui 找到配置项

1、配置打开minigui

Target packages

rockchip BSP packages

[\*] Minigui core lib

2、make savedefconfig

3、make minigui

提示信息查找lib位置

/bin/mkdir -p '/home/chao/rk3308\_linux/buildroot/output/rockchip\_rk3308\_32\_debug/target/usr/lib/pkgconfig'

/bin/mkdir -p '/home/chao/rk3308\_linux/buildroot/output/rockchip\_rk3308\_32\_debug/target/usr/include/minigui'

/usr/bin/install -c -m 644 minigui.pc '/home/chao/rk3308\_linux/buildroot/output/rockchip\_rk3308\_32\_debug/target/usr/lib/pkgconfig'

/usr/bin/install -c -m 644 mgconfig.h '/home/chao/rk3308\_linux/buildroot/output/rockchip\_rk3308\_32\_debug/target/usr/include/minigui'

在下面文件夹找到了库：

rockchip\_rk3308\_32\_debug/target/usr/lib

ls -l libminigui\_ths\* 共找到四个文件。

lrwxrwxrwx 1 root root 27 Apr 2 01:12 libminigui\_ths-3.2.so.0 -> libminigui\_ths-3.2.so.0.0.0

-rwxr-xr-x 1 root root 1260268 Apr 2 01:12 libminigui\_ths-3.2.so.0.0.0

-rwxr-xr-x 1 root root 955 Apr 2 01:12 libminigui\_ths.la

lrwxrwxrwx 1 root root 27 Apr 2 01:12 libminigui\_ths.so -> libminigui\_ths-3.2.so.0.0.0

可见使用的都是libminigui\_ths-3.2.so.0.0.0

决定复制该库到开发板

root@ubuntu:/home/chao/rk3308\_linux/buildroot/output/rockchip\_rk3308\_32\_debug/target/usr/lib# ls libjpeg\* -l

-rwxr-xr-x 1 root root 901 Apr 2 01:10 libjpeg.la

lrwxrwxrwx 1 root root 16 Apr 2 01:10 libjpeg.so -> libjpeg.so.8.1.2

lrwxrwxrwx 1 root root 16 Apr 2 01:10 libjpeg.so.8 -> libjpeg.so.8.1.2

-rwxr-xr-x 1 root root 172852 Apr 2 01:10 libjpeg.so.8.1.2

可见都是指向 libjpeg.so.8.1.2，使用该库到开发板

turbojpeg针对ARM和X86对了优化，宣称其速度是libjpeg的2到4倍。

root@ubuntu:/home/chao/rk3308\_linux/buildroot/output/rockchip\_rk3308\_32\_debug/target/usr/lib# ls libturbojpeg\* -l

-rwxr-xr-x 1 root root 931 Apr 2 01:10 libturbojpeg.la

lrwxrwxrwx 1 root root 21 Apr 2 01:10 libturbojpeg.so -> libturbojpeg.so.0.1.0

lrwxrwxrwx 1 root root 21 Apr 2 01:10 libturbojpeg.so.0 -> libturbojpeg.so.0.1.0

-rwxr-xr-x 1 root root 193180 Apr 2 01:10 libturbojpeg.so.0.1.0

使用libturbojpeg.so.0.1.0

root@ubuntu:/home/chao/rk3308\_linux/buildroot/output/rockchip\_rk3308\_32\_debug/target/usr/lib# ls libpng\* -l

-rwxr-xr-x 1 root root 921 Apr 2 01:11 libpng16.la

lrwxrwxrwx 1 root root 19 Apr 2 01:11 libpng16.so -> libpng16.so.16.34.0

lrwxrwxrwx 1 root root 19 Apr 2 01:11 libpng16.so.16 -> libpng16.so.16.34.0

-rwxr-xr-x 1 root root 144604 Apr 2 01:11 libpng16.so.16.34.0

lrwxrwxrwx 1 root root 11 Apr 2 01:11 libpng.la -> libpng16.la

lrwxrwxrwx 1 root root 11 Apr 2 01:11 libpng.so -> libpng16.so

最终都是使用libpng16.so.16.34.0

具体复制哪几个库还不确定

等到实验后确定到底需要哪几个库。

编译程序需要的头文件

root@ubuntu:/home/chao/rk3308\_linux/buildroot/output/rockchip\_rk3308\_32\_debug/target/usr/include# ls

jconfig.h jerror.h jmorecfg.h jpeglib.h libpng16 minigui pngconf.h png.h pnglibconf.h turbojpeg.h

# 8 单独移植minigui

**Zlib 移植**

./configure --host= arm-rockchip-linux --prefix=/home/chao/minigui/build

Make

Make install

**Freetype移植：**

1> ./configure --host= arm-rockchip-linux --prefix=/home/chao/minigui/build

2>gedit Makefile

CC=arm-rockchip-linux-gnueabihf-gcc

CXX=arm-rockchip-linux-gnueabihf-g++

LD=arm-rockchip-linux-gnueabihf-ld

AS=arm-rockchip-linux-gnueabihf-as

3> make

4> make install

**Jpeg-7移植**

1>

./configure --enable-shared --host=arm-rockchip-linux --prefix=/home/chao/minigui/jpeg-7/libtest CC=arm-rockchip-linux-gnueabihf-gcc CXX=arm-rockchip-linux-gnueabihf-g++ LD=arm-rockchip-linux-gnueabihf-ld AS=arm-rockchip-linux-gnueabihf-as

**移植libpng**

1>

export CFLAGS="-I/home/chao/minigui/build/include/"

export LDFLAGS="-L/home/chao/minigui/build/lib"

2>

./configure --host=arm-rockchip-linux CC=arm-rockchip-linux-gnueabihf-gcc LD=arm-rockchip-linux-gnueabihf-ld AS=arm-rockchip-linux-gnueabihf-as AR=arm-rockchip-linux-gnueabihf-ar CXX=arm-rockchip-linux-gnueabihf-g++ --enable-shared --prefix=/home/chao/minigui/build

3> make

4> make install

如果使用过程发现格式不支持错误，用更新版本库替换

**移植minigui-res-be-3.0.12**

./configure --host=arm-rockchip-linux CC=arm-rockchip-linux-gnueabihf-gcc LD=arm-rockchip-linux-gnueabihf-ld AS=arm-rockchip-linux-gnueabihf-as AR=arm-rockchip-linux-gnueabihf-ar CXX=arm-rockchip-linux-gnueabihf-g++ --prefix=/home/chao/minigui/build

Make

Make install //安装到build/share里面

**移植libminigui**

./configure --prefix=/home/chao/minigui/build --host=arm-rockchip-linux --enable-share --with-targetname=fbcon --enable-video-fbcon --disable-static --enable-lite --enable-rbf16 --enable-tinyscreen --disable-cursor --disable-textmode --enable-tslibial  --enable-pngsupport --enable-jpegsupport --disable-svescreen CC=arm-rockchip-linux-gnueabihf-gcc LD=arm-rockchip-linux-gnueabihf-ld AS=arm-rockchip-linux-gnueabihf-as AR=arm-rockchip-linux-gnueabihf-ar CXX=arm-rockchip-linux-gnueabihf-g++

修改：gedit src/newgal/pcxvfb/Makefile 的194行，将/usr/include去掉

INCLUDES = -I$(abs\_top\_srcdir)/src/include -I$(abs\_top\_srcdir)/include \

-I$(abs\_top\_srcdir)/src/newgal/ **-I/usr/include/**

否则编译pcxvfb.c将出错

修改Makefile:添加

跟踪代码发现，这里应该要显示飞漫的启动动画的。 配置png支持的宏没有打开，最好先编译安装飞漫官方的png库， 配置\_MGIMAGE\_PNG, 重新编译核心库gpl，

./configure --enable-pngsupport  --enable这里其实jpeg库也没支持，先不管，用到再说，这时候编译same会出错，因为Makefile中

没有添加png的链接，Makefile中添加  LIBS +=-lpng -lts –ljpeg

上面是支持png,tslib和jpeg

为了解决如下错误

NEWGAL>FBCON:Error when setting the terminal to graphics mode: Inappropriate ioctl for device

添加--disable-textmode：--disable-textmode

配置文件mgconfig.h添加如下支持jpg和png

#define \_MGIMAGE\_PNG 1

#define \_MGIMAGE\_JPG 1

所有包含\_MGIMAGE\_PNG 和\_MGIMAGE\_JPG的文件都要加上#inlcude “mgconfig.h”

而且每次配置完毕都要检查mgconfig.里面这两个宏是否定义，发现重配置后mgconfig会改变

要记住同时删除库里面的include/minigui目录和libminigui库

编译程序出现

未定义引用错误WndClientRect，找到方法了：  
$cd libminigui-3.0.12-linux  
$grep -r "WndClientRect" .  
发现WndClientRect在libminigui-3.0.12-linux/src/newgdi/gdi.c中，将inline void WndRect(HWND hWnd, PRECT prc)和inline void WndClientRect(HWND hWnd, PRECT prc)的inline去掉，再make clean;make;make install重新生成库，再去编例子就ok了。还有一个函数pix2rgb一样方法处理。

找不到tslib.h，在tslibal.c添加绝tslib.h的绝对路径

/home/chao/rk3308\_linux/buildroot/output/rockchip\_rk3308\_32\_debug/build/tslib-1.15/src

Jpeg.c里面

找不到jpeglib.h,jerror.h

</home/chao/minigui/build/include/jpeglib.h>

#include </home/chao/minigui/build/include/jerror.h>

编译应用程序时提示：

Unable find “ts\_open” “ts\_fd”

修改应用run.sh编译脚本添加：

arm-rockchip-linux-gnueabihf-gcc hello.c -I/home/chao/minigui/build/include/ -L/home/chao/minigui/build/lib -lminigui\_ths -ljpeg -lm -lpthread -ldl -lpng -L/home/chao/rk3308\_linux/buildroot/output/rockchip\_rk3308\_32\_debug/target/usr/lib -lts

如果出现如下错误：

KERNEL>InitGUI: Count not init mouse cursor!

KERNEL>InitGUI: Init failure, please check your MiniGUI configuration or resource.

缺少资源文件,资源路径不对，可能是最后少一个/，应用层缺少

修改MiniGui.cfg

# Edit following line to specify cursor files path

cursorpath=/userdata/minigui/res/cursor/

[resinfo]

respath=/userdata/minigui/

拷贝minigui到userdata/minigui命令

adb push “minigui/” “userdata/minigui”

禁止鼠标，启动画面和屏保

./configure --prefix=/home/chao/minigui/build --host=arm-rockchip-linux --with-targetname=fbcon --disable-static --disable-cursor --disable-textmode --enable-tslibial  --enable-pngsupport --enable-jpgsupport --disable-savescreen --disable-splash --disable-screensaver --disable-mousecalibrate --disable-savebitmap --disable-consoleps2 --disable-consoleimps2 --disable-consolems --disable-consolems3 --disable-consolegpm CC=arm-rockchip-linux-gnueabihf-gcc LD=arm-rockchip-linux-gnueabihf-ld AS=arm-rockchip-linux-gnueabihf-as AR=arm-rockchip-linux-gnueabihf-ar CXX=arm-rockchip-linux-gnueabihf-g++

cp src/newgal/pcxvfb/Makefilebackup src/newgal/pcxvfb/Makefile

cp Makefilebackup Makefile

# 9移植openssl和curl

编译openssl1.0.0g

因为curl需要openssl1.0.0，不能用1.1.0版本。而版本过低会导致

找不到SSL\_COMP\_free\_compression\_methods();

1>./config no-asm shared --prefix=/home/chao/curl/build/

2>修改Makefile

CC= arm-rockchip-linux-gnueabihf-gcc

去掉所有-m64,

3> make

4> make install

no-asm： 是在交叉编译过程中不使用汇编代码代码加速编译过程，原因是它的汇编代码是对arm格式不支持的。

shared ：生成动态连接库。

--prefix ：指定make install后生成目录的路径

如果在linux下安装openssl，执行config和make之后，在执行make install时如果出现下面的错误

cms.pod around line 457: Expected text after =item, not a number

cms.pod around line 461: Expected text after =item, not a number   
cms.pod around line 465: Expected text after =item, not a number   
cms.pod around line 470: Expected text after =item, not a number   
cms.pod around line 474: Expected text after =item, not a number

POD document had syntax errors at /usr/bin/pod2man [line](http://cpro.baidu.com/cpro/ui/uijs.php?c=news&cf=1001&ch=0&di=128&fv=17&jk=51aa8e399b449d99&k=line&k0=line&kdi0=0&luki=1&n=10&p=baidu&q=wangyuead_cpr&rb=0&rs=1&seller_id=1&sid=999d449b398eaa51&ssp2=1&stid=0&t=tpclicked3_hc&tu=u1828791&u=http%3A%2F%2Fwww.myexception.cn%2Fprogram%2F1811081.html&urlid=0) 69.

则在root权限下，执行rm -f /usr/bin/pod2man  然后重新make install

Makefile里面的

LIBDIR=lib64改为lib

删除linux-x86\_64

#CONFIGURE\_ARGS=linux-x86\_64 no-asm shared --prefix=/home/chao/minigui/curl/build/

CONFIGURE\_ARGS= no-asm shared --prefix=/home/chao/minigui/curl/build/

编译curl7.50

./configure --enable-shared --host=arm-rockchip-linux --prefix=/home/chao/curl/build/ CC=arm-rockchip-linux-gnueabihf-gcc CXX=arm-rockchip-linux-gnueabihf-g++

Make&&Make install

# 10 显示logo

## 10.1 uboot显示logo

函数调用流程分析：

Uboot 中显示logo的函数在

int board\_late\_init(void)

{

#ifdef CONFIG\_DRM\_ROCKCHIP

rockchip\_show\_logo();

#endif

}

由于我们不使用DRM,而logo是必需的，因此需要去掉CONFIG\_DRM\_ROCKCHIP宏限制

static init\_fnc\_t init\_sequence\_r[] = {

#ifdef CONFIG\_BOARD\_LATE\_INIT

board\_late\_init,

#endif

}

void board\_init\_r(gd\_t \*new\_gd, ulong dest\_addr)

{

initcall\_run\_list(init\_sequence\_r)

}

10.2 分析显示logo的函数

void rockchip\_show\_logo(void)

{

struct display\_state \*s;

list\_for\_each\_entry(s, &rockchip\_display\_list, head) {

s->logo.mode = s->logo\_mode;

//查找logo

if (load\_bmp\_logo(&s->logo, s->ulogo\_name))

printf("failed to display uboot logo\n");

else

display\_logo(s);

/\* Load kernel bmp in rockchip\_display\_fixup() later \*/

}

}

如果想使用这套函数就只能移植到自己的驱动上面，比较繁琐。

## 10.3 测试uboot的spi驱动是否可用

Drivers/spi/rk\_spi.c添加如下：

static const struct udevice\_id rockchip\_spi\_ids[] = {

{ .compatible = "rockchip,rk3288-spi" },

{ .compatible = "rockchip,rk3368-spi" },

{ .compatible = "rockchip,rk3399-spi" },

{ .compatible = "rockchip,rk3308-spi" },

{ }

};

配置uboot

[\*] Rockchip SPI driver

Rk3308\_common.h里面打开显示调试信息

#define DEBUG

以后要去掉，影响启动速度。

## 10.2 内核显示logo

10.2.1 采用将图片数据直接加载到代码中的显示方式，这样方式简单。

# 11 裁剪内存

Cache和buffer

Cache是读磁盘时缓存，数据从硬盘发送到缓存，应用从缓存获取数据。

Buffer：是写磁盘的缓存。应用写的数据先写到buffer,然后从buffer中再写到硬盘。

因为cpu和硬盘速度差异过大，使用内存作为缓冲。

# free -m

total used free shared buffers cached

Mem: 59320 34200 25120 136 6564 10640

-/+ buffers/cache: 16996 42324

第一部分Mem行：

total：内存总数

used：已经使用的内存数

free：空闲的内存数

shared：可用的共享内存

buffers：内存缓冲数

cached：内存缓存数

第二部分:(-/+ buffers/cache)

used: 除去被用作buffers和cache内存后已用的内存

free：用作buffers和cache的内存加上Mem部分空闲的内存数

第三部分:（Swap）

用一部分磁盘当做内存用的“内存”

在我们来做一下数据统计

total=Mem\_used+Mem\_free

Mem\_used(34200)=Mem\_buffers(10640+6564)+Mem\_cached+(-/+buffers/cache)\_used1(6996)

Mem\_free=(-/+ buffers/cache)\_free-Mem\_buffers-Mem\_cached

如何减少缓存的占用呢？

在/etc/init.d/rcS里面添加如下

 echo 1 > /proc/sys/vm/drop\_caches

ulimit -s 4096 指定线程栈4096

修改后板子内存分配

# free -m

total used free shared buffers cached

Mem: 59320 25000 34320 132 404 6932

-/+ buffers/cache: 17664 41656

Swap: 0 0 0

删除adbd:adb调试使用

ntpd：时间同步

# 12 PWM背光

## 12.1 配置内核支持pwm背光驱动

驱动文件所在位置： drivers/video/backlight/pwm\_bl.c

Graphics support

[\*] Backlight & LCD device support

<\*> Lowlevel Backlight controls

<\*> Generic PWM based Backlight Driver

Device Drivers

[\*] Pulse-Width Modulation (PWM) Support

<\*> Rockchip PWM support

## 12.2 配置pwm背光设备节点

rk3308-voice-module-amic-mainboard-v10-aarch32设备树文件中添加背光设备节点：

在根节点配置。具体可以看rk的手册解释了pwm背光如何使用。

/ {

backlight: backlight {

compatible = "pwm-backlight";

pwms = <&pwm0 0 25000 0>;

brightness-levels = <

0 1 2 3 4 5 6 7

8 9 10 11 12 13 14 15

16 17 18 19 20 21 22 23

24 25 26 27 28 29 30 31

32 33 34 35 36 37 38 39

40 41 42 43 44 45 46 47

48 49 50 51 52 53 54 55

56 57 58 59 60 61 62 63

64 65 66 67 68 69 70 71

72 73 74 75 76 77 78 79

80 81 82 83 84 85 86 87

88 89 90 91 92 93 94 95

96 97 98 99 100 101 102 103

104 105 106 107 108 109 110 111

112 113 114 115 116 117 118 119

120 121 122 123 124 125 126 127

128 129 130 131 132 133 134 135

136 137 138 139 140 141 142 143

144 145 146 147 148 149 150 151

152 153 154 155 156 157 158 159

160 161 162 163 164 165 166 167

168 169 170 171 172 173 174 175

176 177 178 179 180 181 182 183

184 185 186 187 188 189 190 191

192 193 194 195 196 197 198 199

200 201 202 203 204 205 206 207

208 209 210 211 212 213 214 215

216 217 218 219 220 221 222 223

224 225 226 227 228 229 230 231

232 233 234 235 236 237 238 239

240 241 242 243 244 245 246 247

248 249 250 251 252 253 254 255>;

default-brightness-level = <100>;

enable-gpios = <&gpio1 13 GPIO\_ACTIVE\_HIGH>;//pwm使能引脚，高电平有效，未使//用该功能

};

}

&pwm1{

status = "okay";

};

**解释下**：pwms = <&pwm0 0 25000 0>;

**第一个参数** 表示此背光接在 pwm0 上;

**第二个参数** 表示 index 为 0，pwm0 下只有 1个 pwm，所以填0

**第三个参数** 表示周期为 25000ns，即 频率 为 40k

**第四个参数** 表示极性，0正极性，1 负极性  
正极性0表示 背光为正极0～255，占空比从0～100%变化  
负极性 1 表示 背光为负极255～0，占空比从 100～0% 变

解释下：default-brightness-level = <200>  
表示默认的背光,它存在于开机时候背光初始化到安卓设置下来新的背光这段时间, default-brightness-level = < 200 >表示为第 200个元素的背光亮度

## 12.2.3 背光驱动

背光驱动与应用层接口backlight.c

背光驱动pwm\_bl.c

**Backlight.c分析**

backlight\_class\_init

backlight\_class = class\_create(THIS\_MODULE, "backlight");//创建背光类

struct attribute {

const char \*name;

umode\_t mode;

};

static struct attribute \*bl\_device\_attrs[] = {

&dev\_attr\_bl\_power.attr,

&dev\_attr\_brightness.attr,

&dev\_attr\_actual\_brightness.attr,

&dev\_attr\_max\_brightness.attr,

&dev\_attr\_type.attr,

NULL,

};

struct device\_attribute {

struct attribute attr;

ssize\_t (\*show)(struct device \*dev, struct device\_attribute \*attr,

char \*buf);

ssize\_t (\*store)(struct device \*dev, struct device\_attribute \*attr,

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

};

//亮度属性

struct backlight\_properties {

int brightness;

int max\_brightness;

};

//背光设备，属性值和操作函数

struct backlight\_device {

struct backlight\_properties props;

const struct backlight\_ops \*ops;

};

//显示亮度属性

static ssize\_t brightness\_show(struct device \*dev,

struct device\_attribute \*attr, char \*buf)

{

struct backlight\_device \*bd = to\_backlight\_device(dev);

return sprintf(buf, "%d\n", bd->props.brightness);

}

//设置亮度值

int backlight\_device\_set\_brightness(struct backlight\_device \*bd,unsigned long brightness)

{

bd->props.brightness = brightness;

backlight\_update\_status(bd);

}

EXPORT\_SYMBOL(backlight\_device\_set\_brightness);

//设置亮度函数

static ssize\_t brightness\_store(struct device \*dev,struct device\_attribute \*attr, const char \*buf, size\_t count)

{

int rc;

struct backlight\_device \*bd = to\_backlight\_device(dev);

unsigned long brightness;

rc = kstrtoul(buf, 0, &brightness);//内核中字符串转换为unsigned long int

if (rc)

return rc;

rc = backlight\_device\_set\_brightness(bd, brightness);

return rc ? rc : count;

}

static DEVICE\_ATTR\_RW(brightness);

**pwm\_bl.c 分析**

//从设备树获取数据函数

static int pwm\_backlight\_parse\_dt(struct device \*dev,

struct platform\_pwm\_backlight\_data \*data)

{

struct device\_node \*node = dev->of\_node;

struct property \*prop;

int length;

u32 value;

int ret;

if (!node)

return -ENODEV;

memset(data, 0, sizeof(\*data));

/\* determine the number of brightness levels \*/

prop = of\_find\_property(node, "brightness-levels", &length);

if (!prop)

return -EINVAL;

//length应该是数组的长度，每项亮度值占用4子节，因此除以 sizeof(u32)就是亮度级//别的个数

printk("length:%d",length);

data->max\_brightness = length / sizeof(u32);

/\* read brightness levels from DT property \*/

if (data->max\_brightness > 0) {

//sizeof(\*data->levels)表示一个亮度值占用多少字节

size\_t size = sizeof(\*data->levels) \* data->max\_brightness;

data->levels = devm\_kzalloc(dev, size, GFP\_KERNEL);

if (!data->levels)

return -ENOMEM;

//将所有亮度值读到data->levels指向的内存

ret = of\_property\_read\_u32\_array(node, "brightness-levels",

data->levels,

data->max\_brightness);

if (ret < 0)

return ret;

ret = of\_property\_read\_u32(node, "default-brightness-level",

&value);//读取默认亮度级别

if (ret < 0)

return ret;

data->dft\_brightness = value;

data->max\_brightness--;//最大亮度索引，数组索引是数组个数-1

}

data->enable\_gpio = -EINVAL;

return 0;

}

//probe函数

static int pwm\_backlight\_probe(struct platform\_device \*pdev)

{

struct platform\_pwm\_backlight\_data \*data = dev\_get\_platdata(&pdev->dev);

struct platform\_pwm\_backlight\_data defdata;

struct backlight\_properties props;

struct backlight\_device \*bl;

struct device\_node \*node = pdev->dev.of\_node;

struct pwm\_bl\_data \*pb;

int initial\_blank = FB\_BLANK\_UNBLANK;

struct pwm\_args pargs;

int ret;

ret = pwm\_backlight\_parse\_dt(&pdev->dev, &defdata);

data = &defdata;

pb = devm\_kzalloc(&pdev->dev, sizeof(\*pb), GFP\_KERNEL);

if (data->levels) {//如果有亮度数组

unsigned int i;

//找出最大亮度值，最大亮度值不一定在最后

for (i = 0; i <= data->max\_brightness; i++)

if (data->levels[i] > pb->scale)

pb->scale = data->levels[i];//最大亮度值

pb->levels = data->levels;

}

pb->dev = &pdev->dev;

pb->power\_supply = devm\_regulator\_get(&pdev->dev, "power");

//获取pwm设备，根据设备树指示是哪个pwm获取对该pwm操作的函数

pb->pwm = devm\_pwm\_get(&pdev->dev, NULL);

//adjust the current PWM config to the PWM arguments

pwm\_adjust\_config(pb->pwm);

pwm\_get\_args(pb->pwm, &pargs);//获取设置的pwm周期和极性

pb->period = pargs.period;

if (!pb->period && (data->pwm\_period\_ns > 0))

pb->period = data->pwm\_period\_ns;

pb->lth\_brightness = data->lth\_brightness \* (pb->period / pb->scale);

memset(&props, 0, sizeof(struct backlight\_properties));

props.type = BACKLIGHT\_RAW;

props.max\_brightness = data->max\_brightness;

//注册背光设备，backlight里面设置

bl = backlight\_device\_register(dev\_name(&pdev->dev), &pdev->dev, pb,

&pwm\_backlight\_ops, &props);

bl->props.brightness = data->dft\_brightness;

bl->props.power = initial\_blank;

backlight\_update\_status(bl);

platform\_set\_drvdata(pdev, bl);

return 0;

**}**

/\*如何获取一个Pwm设备

获取pwm设备就是获取到了设备寄存器的操作函数，这些都是由内核函数隐形的调用。

操作函数是芯片初始化的时候注册的。

\*/

struct pwm\_args {

unsigned int period;

enum pwm\_polarity polarity;

};

struct pwm\_device {

const char \*label;

unsigned long flags;

unsigned int hwpwm;

unsigned int pwm;

struct pwm\_chip \*chip;

void \*chip\_data;

struct pwm\_args args;

struct pwm\_state state;

};

devm\_pwm\_get(&pdev->dev, NULL);

pwm = pwm\_get(dev, con\_id);

device \*pwm\_get(struct device \*dev, const char \*con\_id)

of\_pwm\_get(dev->of\_node, con\_id);

of\_parse\_phandle\_with\_args(np, "pwms", "#pwm-cells",

index,&args);//解析属性值

\_\_of\_parse\_phandle\_with\_args(np, list\_name, cells\_name, 0,

index, out\_args);

of\_node\_to\_pwmchip(args.np)//找到芯片pwm操作函数

pwm = pc->of\_xlate(pc, &args); //最终生成struct pwm\_device类型*.*

static const struct backlight\_ops pwm\_backlight\_ops = {

.update\_status = pwm\_backlight\_update\_status,//所有占空比设置最终都是通过该函//数设置的

.check\_fb = pwm\_backlight\_check\_fb,

};

# 13 wifi 配置

## 13.1配置设备树

rk3308-voice-module-mainboard-v10-aarch32.dtsi wireless-wlan {

compatible = "wlan-platdata";

rockchip,grf = <&grf>;

clocks = <&cru SCLK\_WIFI>;

clock-names = "clk\_wifi";

ref-clock-frequency = <24000000>;

pinctrl-names = "default";

pinctrl-0 = <&wifi\_wake\_host>;

//wifi\_chip\_type = "rtl8723ds";

wifi\_chip\_type = "rtl8189fs";//具体名字通过内核搜索rtl8189\*找到

WIFI,host\_wake\_irq = <&gpio0 RK\_PA0 GPIO\_ACTIVE\_LOW>;

status = "okay";

}

## 13.2 配置内核

Device Drivers

[\*] Network device support

[\*] Wireless LAN

[\*] Network device support

[\*] Wireless LAN

[\*] Rockchip Wireless LAN support

<\*> Realtek 8189F SDIO WiFi

## 13.3 命令行配网

### 13.3.1 buildroot选择配置文件

根据对应WiFi选择相应配置：

Target packages

[\*] rockchip BSP packages

[\*] rkwifibt

wifi chip support (RTL8189FS)

### 13.3.2 串口终端输入

1>查看是否启动了wpa\_supplicant，并且找到配置文件

ps|grep wpa\_supplicant

282 root 3548 S wpa\_supplicant -B -i wlan0 -c /data/cfg/wpa\_supplica

2>修改配置文件

#cd /data/cfg/wpa\_supplicant.conf

# cat wpa\_supplicant.conf

ctrl\_interface=/var/run/wpa\_supplicant

update\_config=1

ap\_scan=1

network={

ssid="ZGWLCS"

psk="ZGWLceshi"

key\_mgmt=WPA-PSK

}

3>重新连接wifi

 killall wpa\_supplicant //先杀死wpa\_supplicant

//重新启动wpa\_supplicant

wpa\_supplicant -B -c/data/cfg/wpa\_supplicant.conf -iwlan0

4> iwlist wlan0 scanning

## 13.4 手机配网

在板子上设置一个初始连接名字为ssid=Note5的wifi，手机热点改为该wifi。设置密码为abcdefgh,加密方式WPA PSK。那么板子启动后后自动连接该wifi。使用手机发送网络名字和密码。

板子上面的wifi配置

ctrl\_interface=/var/run/wpa\_supplicant

update\_config=1

ap\_scan=1

network={

ssid="Note5"

psk="abcdefgh"

}

执行命令查看状态

wpa\_cli -iwlan0 status

ping [www.baidu.com](http://www.baidu.com)

**方法1：**:手机使用网络调试助手启动一个服务器

服务器地址：192.168.43.1 端口号5000

板子上面启动联网客户端

./tcpclientrk3308 192.168.43.1 5000

手机发送ssid和密码。

客户端收到ssid和密码后写入到data/cfa/ wpa\_supplicant

然后执行wpa\_cli reconnect重新连接网络

方案缺点：

手机助手的ip是不确定的，所以通过ip连接手机服务器不可行

**方法2：开发板启动一个服务器**

开发板把自身的ip地址显示到屏幕上，然后手机连接开发板的服务器，然后发送ssid和密码。开发板服务器程序收到ssid和密码后写入wpa\_supplicant.conf，然后重新连接。可以实现。

如何实现：

手机启动一个名字为UIOT的wifi热点，密码abcdefgh。

屏幕上点击配网按钮，启动一个获取wlan0的ip地址的进程，获取之后显示在ip地址栏。启动服务器进程等待手机客户端连接接收ssid和password,收到之后写入wpa\_supplicant.conf，执行wpa\_cli reconnect命令，执行完成退出服务器。

## 13.5 iw工具使用

ifconfig wlan0 up                 /\* 打开无线网卡 \*/

ifconfig wlan0 down                 /\* 关闭无线网卡 \*/

iw list                                   /\* 列出WIFI网卡的性能\*/

iw dev wlan0 scan   // 扫描WIFI AP

iw wlan0 connect linux      // 连接到WIFI AP：linux (open)  没有设置密码

iw dev wlan0 link                /\* 查看连接状态 \*/

 iw wlan0 disconnect           /\* 关闭WIFI连接 \*/

ifconfig wlan0 192.168.1.111              配置无线网卡wlan0

有密码连接方式：

iw wlan0 connect linux keys d:0:8888888888888    /\* 密码需要设置成13位 \*/

### 13.6 wpa\_supplicant 使用

ifconfig wlan0 up

wpa\_supplicant -Dnl80211 -i wlan0 -c /data/wpa\_supplicant.conf -B

/\*根据配置文件/etc/wpa\_supplicant.conf来初始化wlan0如果wpa\_supplicant.conf文件为空,则不会自动连接,需要连接一次网络才行\*/

**wpa\_supplicant.conf文件的意义:**

    用来保存用户连接过的WIFI热点,好处在于,然后每次开启WIFI时,则会自动根据该文件来连接热点

**注意:**该文件保存的每个热点都有个priority值,用来系统优先选择哪个热点

wpa\_cli -i wlan0 scan //扫描附近热点

wpa\_cli -i wlan0 scan\_result //打印热点信息

**设置网络**

1. 添加网络0

# wpa\_cli -i wlan0 add\_network

0 /\* 添加一个网络连接,并返回网络ID号,如果wpa\_supplicant.conf文件为空,则返回0,表示第一个热点,然后该ID号会在连接热点用到\*/

**添加网络连接的意义:**用来区分用户连接过的每个热点

2>设置网络id号为0的网络的ssid和psk

# wpa\_cli -i wlan0 set\_network 0 ssid '"ZGWLCS"'

//设置ID为0的热点SSID

# wpa\_cli -i wlan0 set\_network 0 psk '"ZGWLceshi"'

//设置ID为0的热//点的密码

**PS:**如果是没有加密的公共网络,则输入以下密码:

wpa\_cli -i wlan0 set\_network 0 key\_mgmt NONE

3>设置网络

wpa\_cli -i wlan0 set\_network 0 priority 0 //设置当前网络优先级,默认为0

wpa\_cli -i wlan0 set\_network 0 scan\_ssid 0 //设置ssid默认为1即可

wpa\_cli -i wlan0 select\_network 0 //选择网络0(这样做的好处在于,可以取消先前的其它网络连接)测试可以代替enable\_network

wpa\_cli -i wlan0 enable\_network 0 //启动连接ID为0的网络

udhcpc -b -i wlan0 -q //不需要

//此时只是连接上,所以还要自动分配IP地址,-b:后台(back) –i:指定接口 –q:获得续约后退出 ,如果想尽在前台运行,则去掉-b,加上-f

4> 保存连接的配置

wpa\_cli -i wlan0 save\_config //保存当前连接的网络配置到wpa\_supplicant.conf文件中,使得下次自动连接

5> 查看网卡当前连接的信息

wpa\_cli -i wlan0 status //查看当前网卡是处于什么状态

wpa\_cli -i wlan0 list\_network //查看当前连接的是哪个网络

打印如下,可以看到当前连接的ID号为0:

5> 断开热点并清除密码

wpa\_cli -i wlan0 disable\_network 0 //与id0的网络进行断开

wpa\_cli -i wlan0 remove\_network 0 //将id0的网络移除掉,必须先断开才行

wpa\_cli -i wlan0 save\_config //并更新wpa\_supplicant.conf文件

6> 关闭WIFI

killall udhcpc > /dev/null

killall wpa\_supplicant > /dev/null

ifconfig wlan0 down

**进入交互模式**

执行wpa\_cli 命令，然后enter

Status 查看状态，scan命令扫描wifi热点，scan\_results查看扫描的热点。

Add\_network,set\_network,select\_network,save\_config

## 13.5 驱动源码

rfkill-wlan.c

电源相关部分：

rfkill\_wlan\_init -> rfkill-waln.c

  platform\_driver\_register -> //platform\_driver是rfkill\_wlan\_driver， name是//wlan-platdata，dts中配置

    rfkill\_wlan\_probe ->

      wlan\_platdata\_parse\_dt //解析dts中关于wifi的配置,dts的说明参考前面一篇文章

      gpio\_direction\_output //默认关闭WiFi电源

      rockchip\_wifi\_voltage\_select //设置IO power.

驱动部分文件夹

Drivers/rtl8189fs

## 13.6 rtl8189f 功能引脚

其他引脚如电源等可以看手册和rk3308原理图

|  |  |
| --- | --- |
| XI | 晶振输入 |
| XO | 晶振输出 |
| RF\_INOUT | 天线 |
| CHIP\_EN | 芯片使能 |
| SD\_D2 | SDIO数据线 |
| SD\_D3 | SDIO数据线 |
| SD\_CMD | SDIO命令线 |
| SD\_CLK | SDIO CLK |
| SD\_D0 | SDIO数据线 |
| SD\_D1 | SDIO数据线 |
| INT/GPIO0 | WIFI\_WAKE\_HOST |

## 13.7 调试遇到问题

Uiot板子循环打印如下信息，iwlist无法获取wifi热点

[ 6.104404] RTL871X: nolinked power save leave

[ 7.657442] RTL871X: nolinked power save enter

打印该信息的c文件为：

rtw\_pwrctrl.c (drivers\net\wireless\rockchip\_wlan\rtl8189fs\core) line 90 :

在函数\_ips\_enter()里面打印

DBG\_871X\_LEVEL(\_drv\_always\_, "nolinked power save enter\n");

开发板可以正常连接wifi,拔掉天线，也会打印上述信息。

可见，很可能是天线部分设计问题。

发现天线的电感没有焊接,焊接完依然有问题，又发现晶振也没有焊接，焊接了晶振还是有问题。晶振应该是26MHZ,但是实际板子采用了cpu的24MHZ输出，所以导致了问题。

## 13.8 wifi连接断开监测

# 14 高精度定时器hrtimer

### 14.1 hrtimer函数的简单介绍

**hrtimer\_init()**

说明：初始化定时器工作模式。

原型：hrtimer\_init(&timer, CLOCK\_MONOTONIC, HRTIMER\_MODE\_REL);

参数：

CLOCK\_MONOTONIC：从系统启动这一刻起开始计时,不受系统时间被用户改变的影响。

CLOCK\_REALTIME:系统实时时间,随系统实时时间改变而改变。

HRTIMER\_MODE\_REL：相对时间

设置回调函数：

timer.function = timer\_func;

定时器到时该函数将被调用。

回调函数原型：static enum hrtimer\_restart timer\_func(struct hrtimer \*timer)

//注：该回调函数为原子操作不能被中断

**hrtimer\_start()**

说明：启动定时器

原型：hrtimer\_start hrtimer\_start(&timer, ktime\_set(value / 1000, (value % 1000) \* 1000000),HRTIMER\_MODE\_REL);.

返回值：enum hrtimer\_restart {

HRTIMER\_NORESTART,  /\* Timer is not restarted \*/

HRTIMER\_RESTART,  /\* Timer must be restarted \*/

};

**hrtimer\_cancel()**

说明：取消一个hrtimer

原型：

int hrtimer\_cancel(struct hrtimer \*timer);

### 14.2 hrtimer结构

**hrtimer结构体**

struct hrtimer {

struct timerqueue\_node node;

ktime\_t \_softexpires;

enum hrtimer\_restart (\*function)(struct hrtimer \*);

struct hrtimer\_clock\_base \*base;

unsigned **long** state;

......

};

**State：**标识当前定时器状态：

#define HRTIMER\_STATE\_INACTIVE 0x00 // 定时器未激活

#define HRTIMER\_STATE\_ENQUEUED 0x01 // 定时器已经被排入红黑树中

#define HRTIMER\_STATE\_CALLBACK 0x02 // 定时器的回调函数正在被调用

#define HRTIMER\_STATE\_MIGRATE 0x04 // 定时器正在CPU之间做迁移

hrtimer的到期时间：

可以基于以下几种时间基准系统

enum hrtimer\_base\_type {

HRTIMER\_BASE\_MONOTONIC,  // 单调递增的monotonic时间，不包含休眠时间

HRTIMER\_BASE\_REALTIME,  // 平常使用的墙上真实时间

HRTIMER\_BASE\_BOOTTIME,  // 单调递增的boottime，包含休眠时间

HRTIMER\_MAX\_CLOCK\_BASES,  // 用于后续数组的定义

};

## 14.3 hrtimer使用

添加一个定时器：

void hrtimer\_init(struct hrtimer \*timer, clockid\_t which\_clock,

enum hrtimer\_mode mode);

which\_clock可以是CLOCK\_REALTIME、CLOCK\_MONOTONIC、CLOCK\_BOOTTIME中的一种，mode则可以是相对时间HRTIMER\_MODE\_REL，也可以是绝对时间HRTIMER\_MODE\_ABS。设定回调函数：

timer.function = hr\_callback;

如果定时器无需指定一个到期范围，可以在设定回调函数后直接使用hrtimer\_start激活该定时器：

**int** hrtimer\_start(struct hrtimer \*timer, ktime\_t tim,

const enum hrtimer\_mode mode);

如果需要指定到期范围，则可以使用hrtimer\_start\_range\_ns激活定时器

hrtimer\_start\_range\_ns(struct hrtimer \*timer, ktime\_t tim,

unsigned **long** range\_ns, const enum hrtimer\_mode mode);

要取消一个hrtimer，使用hrtimer\_cancel：

**int** hrtimer\_cancel(struct hrtimer \*timer);

以下两个函数用于推后定时器的到期时间：

extern u64 hrtimer\_forward(struct hrtimer \*timer, ktime\_t now, ktime\_t interval);

/\* Forward a hrtimer so it expires after the hrtimer's current now \*/

static inline u64 hrtimer\_forward\_now(struct hrtimer \*timer,

ktime\_t interval)

{

return hrtimer\_forward(timer, timer->base->get\_time(), interval);

}

使用例子如下

hrtimer\_forward\_now(timer,ktime\_set(DELAY\_SECOND, DELAY\_NANOSECOND));

ktime\_set(DELAY\_SECOND, DELAY\_NANOSECOND)//构造定时时间，第一个参数单位是秒

//第二个参数单位是纳秒

# 15 红外感应开关

## 15.1 分离器件电路

### 15.1.1 设计思路

红外发射电路以1Khz频率发射红外线，红外接收器通过接收的红外线拉高拉低电平，如果在1s内接收到800以上的电平反转，就认为有人。

发射端驱动程序采用高精度定时器，500us翻转一次IO口电平。

接收端采用中断方式对io口计数，上升沿中断计数一次。

1s判断一次计数值。实现方法为在高精度定时器里面做时间计数，每当计数到2000次就是1s，此时就判断一次中断计数，如果计数值超过800就认为检测到人。

### 15.1.2 设备树配置

infrared: infrared {

compatible = "infrared-switch";

interrupts = <&gpio0 RK\_PC2 IRQ\_TYPE\_EDGE\_RISING>;

reset-gpios=<&gpio2 RK\_PA3 GPIO\_ACTIVE\_LOW>;

}

## 15.2 采用集成红外芯片

### 15.2.1 设计思路

该芯片只能在有人活动时检测，不能检测静止状态的人。检测到有人活动就会发出一个

2ms左右的高电平。

设计思路：每100ms检测一次电平，连续三次电平都是高电平就认为是有人活动。向应用层发出一个通知信号，然后修改定时时间为2S,2S后继续100ms定时检测。

应用层收到信号，打开屏幕，延时10S后如果有人标志为0，则关闭屏幕。

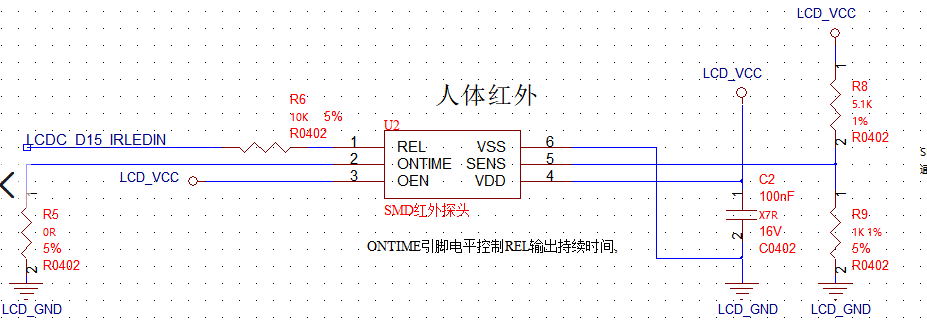
### 15.2.2 dts配置和电路图

infrared: infrared {

compatible = "infrared-switch";

rx-gpios = <&gpio1 RK\_PC3 GPIO\_ACTIVE\_HIGH>;

};



### 15.2.3 驱动代码

#include <linux/module.h>

#include <linux/init.h>

#include <linux/fs.h>

#include <linux/interrupt.h>

#include <linux/irq.h>

#include <linux/sched.h>

#include <linux/pm.h>

#include <linux/slab.h>

#include <linux/sysctl.h>

#include <linux/proc\_fs.h>

#include <linux/delay.h>

#include <linux/platform\_device.h>

#include <linux/input.h>

#include <linux/workqueue.h>

#include <linux/gpio.h>

#include <linux/of.h>

#include <linux/of\_platform.h>

#include <linux/of\_gpio.h>

#include <linux/of\_irq.h>

#include <linux/platform\_device.h>

#include<linux/uaccess.h>

//读IO口定时时间 100/10×10ms=100ms

#define DELAY1 HZ/10

#define DELAY2 HZ\*2

struct infrared\_data {

int rx\_pin; //检测IO口

int rx\_flags; //设备树中设置的高电平代表有人还是低电平代表有人

};

struct infrared {

struct platform\_device \*pdev;

int major;

struct class \*class;

struct timer\_list timer;;

unsigned char human\_flag; //有人标志 1代表有人，0代表没有人

struct infrared\_data \*data;

struct fasync\_struct \*async\_queue;

};

struct infrared \*infrared;

//传感器只有在人来和人走时候都会给出一个高电平信号，现在只要检测到高电平就发送信号。

static void infrared\_timer\_func(unsigned long data)

{

//连续3次检查io口，如果都是一样则认为没有误报

static unsigned char new\_val[3],count;

unsigned char val\_sum;

int pin\_val;//io口电平

mod\_timer(&infrared->timer,jiffies + DELAY1);

pin\_val = gpio\_get\_value(infrared->data->rx\_pin);

if(pin\_val == 0)

new\_val[count] = 0;

else if(pin\_val == 1)

new\_val[count] = 1;

count++;

printk("pin\_val:%u",pin\_val);

if(count >= 3){

val\_sum = new\_val[0]+new\_val[1]+new\_val[2];

printk("count:%u,val\_sum:%u\n",count,val\_sum);

if(val\_sum == 3){

kill\_fasync(&infrared->async\_queue,SIGIO,POLL\_IN);

mod\_timer(&infrared->timer,jiffies + DELAY2);

}

count = 0;

memset(new\_val,0,3);

}

}

int infrared\_open(struct inode \*inode, struct file \*filp)

{

init\_timer(&infrared->timer);

infrared->timer.function = infrared\_timer\_func;

infrared->timer.expires = jiffies + DELAY1;

add\_timer(&infrared->timer);

return 0;

}

static int infrared\_fasync(int fd,struct file \*filp,int mode)

{

return fasync\_helper(fd,filp,mode,&infrared->async\_queue);

}

int infrared\_release(struct inode \*inode, struct file \*filp)

{

infrared\_fasync(-1,filp,0);

del\_timer(&infrared->timer);

return 0;

}

static const struct file\_operations infrared\_fops = {

.owner = THIS\_MODULE,

.open = infrared\_open,

.fasync= infrared\_fasync,

.release = infrared\_release,

};

static struct infrared\_data \*infrared\_probe\_dt(struct device \*dev)

{

struct infrared\_data \*pdata = NULL;

struct device\_node \*np = dev->of\_node;

enum of\_gpio\_flags of\_flags = OF\_GPIO\_ACTIVE\_LOW ;

if (!np) {

return ERR\_PTR(-EINVAL);

}

pdata = devm\_kzalloc(dev,sizeof(struct infrared\_data), GFP\_KERNEL);

if (!pdata)

return ERR\_PTR(-ENOMEM);

if (of\_find\_property(np, "rx-gpios", NULL)) {

pdata->rx\_pin = of\_get\_named\_gpio\_flags(np, "rx-gpios", 0, &of\_flags);

}

gpio\_direction\_input(pdata->rx\_pin);

return pdata;

}

static int infrared\_probe(struct platform\_device \*pdev)

{

struct infrared\_data \*pdata;

struct device \*dev;

int ret;

printk("probe infrared device\n");

dev=&pdev->dev;

pdata = dev->platform\_data;

if(!pdata) {

pdata=infrared\_probe\_dt(dev);

if (IS\_ERR(pdata)){

ret = PTR\_ERR(pdata);

return ret;

}

}

infrared=devm\_kzalloc(dev,sizeof(struct infrared), GFP\_KERNEL);

if (!infrared){

return -ENOMEM;

}

infrared->data=pdata;

infrared->major = register\_chrdev(0, "infrared", &infrared\_fops);

infrared->class = class\_create(THIS\_MODULE, "infrared");

device\_create(infrared->class, NULL, MKDEV(infrared->major, 0), NULL, "infrared");

return 0;

}

static int infrared\_remove(struct platform\_device \*pdev)

{

del\_timer(&infrared->timer);

device\_destroy(infrared->class, MKDEV(infrared->major, 0));

class\_destroy(infrared->class);

unregister\_chrdev(infrared->major, "infrared");

return 0;

}

static const struct of\_device\_id infrared\_of\_match[] = {

{ .compatible = "infrared-switch", },

{ },

};

MODULE\_DEVICE\_TABLE(of,infrared\_of\_match);

static struct platform\_driver infrared\_driver = {

.probe = infrared\_probe,

.remove = infrared\_remove,

.driver = {

.name = "infrared-switch",

.of\_match\_table = of\_match\_ptr(infrared\_of\_match),

}

};

static int \_\_init infrared\_init(void)

{

return platform\_driver\_register(&infrared\_driver);

}

static void \_\_exit infrared\_exit(void)

{

platform\_driver\_unregister(&infrared\_driver);

}

module\_init(infrared\_init);

module\_exit(infrared\_exit);

MODULE\_DESCRIPTION("infrared switch Driver");

MODULE\_LICENSE("GPL");

### 15.2.4 应用测试代码

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<fcntl.h>

#include<sys/types.h>

#include<sys/stat.h>

#include<sys/times.h>

#include<sys/select.h>

#include<signal.h>

#include<string.h>

static int human\_flag=0;

static void signalio\_handler(int signum)

{

human\_flag = 1;

printf("Here comes a human\n");

}

int main(void)

{

int fd\_infra,fd\_back;

int oflags;

char \*str="255";

fd\_infra=open("/dev/infrared",O\_RDWR,S\_IRUSR|S\_IWUSR);

if(fd\_infra != -1){

printf("open infraed ok\n");

signal(SIGIO,signalio\_handler);

fcntl(fd\_infra,F\_SETOWN,getpid());

oflags=fcntl(fd\_infra,F\_GETFL);

fcntl(fd\_infra,F\_SETFL,oflags|FASYNC);

}

fd\_back = open("/sys/class/backlight/backlight/brightness", O\_RDWR);

if(fd\_back <= 0){

printf("can not open backlight device .\n");

return -1;

}

while(1){

if(human\_flag == 1){

human\_flag = 0;

str = "255";

write(fd\_back,str,strlen(str));

printf("turn on lcd\n");

sleep(10);

}else{

str = "0";

write(fd\_back,str,strlen(str));

printf("turn off lcd\n");

sleep(2);

}

}

close(fd\_back);

close(fd\_infra);

return 0;

}

### 15.2.5 测试

执行测试代码

./sigio 当有人靠近屏幕亮，10s没有人活动屏幕灭。

# 16 ADC按键驱动

Firefly-RK3399开发板上的 AD 接口有两种，分别为：温度传感器 (Temperature Sensor)、逐次逼近ADC (Successive Approximation Register)。其中：

TS-ADC(Temperature Sensor)：支持两通道，时钟频率必须低于800KHZ

SAR-ADC(Successive Approximation Register)：支持六通道单端10位的SAR-ADC，时钟频率必须小于13MHZ。

内核采用工业 I/O 子系统来控制 ADC，该子系统主要为 AD 转换或者 DA 转换的传感器设计。 下面以SAR-ADC为例子，介绍 ADC 的基本配置方法。

## 16.1 配置内核支持ADC和adc-keys驱动

Device Drivers

<\*> Industrial I/O support

Analog to digital converters

<\*> Rockchip SARADC driver //ADC驱动

Device Drivers

Input device support

[\*] Keyboards

<\*> ADC Ladder Buttons

## 16.2 配置设备树

Adc 设备树配置

saradc: saradc@ff1e0000 {

compatible = "rockchip,rk3308-saradc", "rockchip,rk3399-saradc";

reg = <0x0 0xff1e0000 0x0 0x100>;

interrupts = <GIC\_SPI 37 IRQ\_TYPE\_LEVEL\_HIGH>;

#io-channel-cells = <1>;

clocks = <&cru SCLK\_SARADC>, <&cru PCLK\_SARADC>;

clock-names = "saradc", "apb\_pclk";

resets = <&cru SRST\_SARADC\_P>;

reset-names = "saradc-apb";

status = "disabled";

};

rk3308-voice-module-v10-aarch32.dtsi文件中配置

adc-keys {

compatible = "adc-keys";

io-channels = <&saradc 1>;//申请adc通道1

io-channel-names = "buttons";//驱动根据该名字搜索设备树

poll-interval = <100>;//按键查询时间

keyup-threshold-microvolt = <1800000>;//最高电压

esc-key {

linux,code = <KEY\_MICMUTE>;

label = "micmute";

press-threshold-microvolt = <1130000>;

};

home-key {

linux,code = <KEY\_MODE>;

label = "mode";

press-threshold-microvolt = <901000>;

};

menu-key {

linux,code = <KEY\_PLAY>;

label = "play";

press-threshold-microvolt = <624000>;

};

vol-down-key {

linux,code = <KEY\_VOLUMEDOWN>;

label = "volume down";

press-threshold-microvolt = <300000>;

};

vol-up-key {

linux,code = <KEY\_VOLUMEUP>;

label = "volume up";

press-threshold-microvolt = <18000>;

};

根据电路图计算电压阈值。

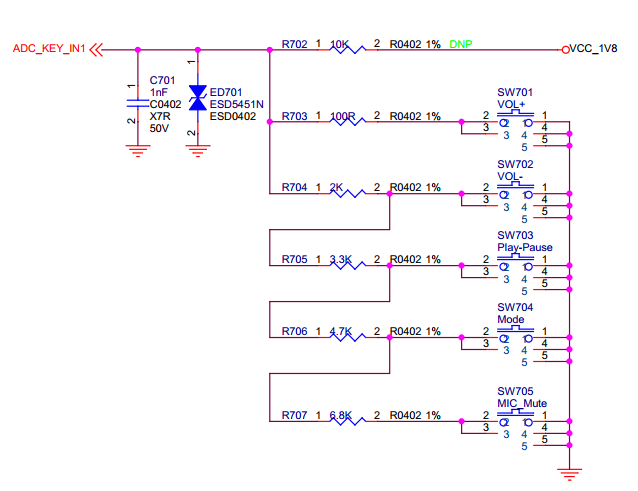
电压值均扩大1000000倍，1.8V=1800000

对于vol+按键按下，ADC键值应该为1800000\*100/(10000+100)=1798

对于vol-按键按下，ADC键值应该为1800000\*2000/(10000+2000)=300000

对于play按键按下，ADC键值应该为1800000\*(2000+3300)/(10000++3300+2000)=623529

同理计算其他键值，因为驱动中上报的键值是测量值最接近的那个。



## 16.3 linux IIO子系统使用

IIO子系统就是 Industrial I/O subsystem（工业 I/O 子系统），主要用在ADC和DAC中。

获取 AD 通道

struct iio\_channel \*chan; //定义 IIO 通道结构体chan = iio\_channel\_get(&pdev->dev, NULL);

//获取 IIO 通道结构体

注：iio\_channel\_get 通过 probe 函数传进来的参数 pdev 获取 IIO 通道结构体，probe 函数如下：

static int XXX\_probe(struct platform\_device \*pdev);

读取 AD 采集到的原始数据

int val,ret;

ret = iio\_read\_channel\_raw(chan, &val);

调用 iio\_read\_channel\_raw 函数读取 AD 采集的原始数据并存入 val 中。

计算采集到的电压

使用标准电压将 AD 转换的值转换为用户所需要的电压值。其计算公式如下：

Vref / (2^n-1) = Vresult / raw

注：

Vref 为标准电压

n 为 AD 转换的位数

Vresult 为用户所需要的采集电压

raw 为 AD 采集的原始数据

例如，标准电压为 1.8V，AD 采集位数为 10 位，AD 采集到的原始数据为 568，则：

Vresult = (1800mv \* 568) / 1023;

## 接口说明

struct iio\_channel \*iio\_channel\_get(struct device \*dev, const char \*consumer\_channel);

功能：获取 iio 通道描述

参数：

dev: 使用该通道的设备描述指针

consumer\_channel: 该设备所使用的 IIO 通道描述指针

void iio\_channel\_release(struct iio\_channel \*chan);

功能：释放 iio\_channel\_get 函数获取到的通道

参数：

chan：要被释放的通道描述指针

int iio\_read\_channel\_raw(struct iio\_channel \*chan, int \*val);

功能：读取 chan 通道 AD 采集的原始数据。

参数：

chan：要读取的采集通道指针

val：存放读取结果的指针

获取所有ADC值

有个便捷的方法可以查询到每个SARADC的值：

cat /sys/bus/iio/devices/iio\:device0/in\_voltage\*\_raw

为何按上面的步骤申请SARADC，会出现申请报错的情况？

驱动需要获取ADC通道来使用时，需要对驱动的加载时间进行控制，必须要在saradc初始化之后。saradc是使用module\_platform\_driver()进行平台设备驱动注册，最终调用的是module\_init()。所以用户的驱动加载函数只需使用比module\_init()优先级低的，例如：late\_initcall()，就能保证驱动的加载的时间比saradc初始化时间晚，可避免出错。

## 16.4 ADC按键驱动分析

搜索设备树adc-keys：

adc-keys.c (drivers\input\keyboard) line 193 : { .compatible = "adc-keys", },

rockchip-saradc.c是ADC的驱动，注册对ADC的操作函数。

驱动中的结构体

struct adc\_keys\_button {

u32 voltage;

u32 keycode;

};

struct adc\_keys\_state {

struct iio\_channel \*channel;

u32 num\_keys;

u32 last\_key;

u32 keyup\_voltage;

const struct adc\_keys\_button \*map;

};

### 16.3.1 驱动中配置设备树

static const struct of\_device\_id adc\_keys\_of\_match[] = {

{ .compatible = "adc-keys", },

{ }

};

### 16.3.2 注册adc\_keys驱动

static struct platform\_driver \_\_refdata adc\_keys\_driver = {

.driver = {

.name = "adc\_keys",

.of\_match\_table = of\_match\_ptr(adc\_keys\_of\_match),

},

.probe = adc\_keys\_probe,

};

### 16.3.3 probe函数

adc\_keys\_probe(struct platform\_device \*pdev)

//获取buttons使用的adc通道

st->channel = devm\_iio\_channel\_get(dev, "buttons");

channel = iio\_channel\_get(dev, channel\_name);

channel = of\_iio\_channel\_get\_by\_name(dev->of\_node,channel\_name);

iio\_channel\_get\_sys(name, channel\_name);

//获取通道的类型

iio\_get\_channel\_type(st->channel, &type);

if (type != IIO\_VOLTAGE) { //只支持电压型

dev\_err(dev, "Incompatible channel type %d\n", type);

return -EINVAL;

}

//设备树中的电压设置最大值1800000，实际1.8V，除以1000就是1800mv

device\_property\_read\_u32(dev, "keyup-threshold-microvolt",&st->keyup\_voltage)

st->keyup\_voltage /= 1000;

adc\_keys\_load\_keymap(dev, st);//获取按键电压阈值和键码

st->num\_keys = device\_get\_child\_node\_count(dev);

map = devm\_kmalloc\_array(dev, st->num\_keys, sizeof(\*map), GFP\_KERNEL);

//对该设备下所有子节点进行分析

device\_for\_each\_child\_node(dev, child) {

//读取每个按键的分压值

ifwnode\_property\_read\_u32(child, "press-threshold-microvolt",&map[i].voltage))

map[i].voltage /= 1000;

//读取按键码

fwnode\_property\_read\_u32(child, "linux,code",&map[i].keycode))

i++;

}

st->map = map;

//分配一个支持轮询的设备，select-poll函数

poll\_dev = devm\_input\_allocate\_polled\_device(dev);

device\_property\_read\_u32(dev, "poll-interval", &value)//读取轮询间隔时间

poll\_dev->poll = adc\_keys\_poll;//select-poll函数进入内核执行该函数

poll\_dev->private = st;

input = poll\_dev->input;

\_\_set\_bit(EV\_KEY, input->evbit);//设置支持按键事件

for (i = 0; i < st->num\_keys; i++)//设置支持哪个按键

\_\_set\_bit(st->map[i].keycode, input->keybit);

//是否支持重复类事件

if (device\_property\_read\_bool(dev, "autorepeat"))

\_\_set\_bit(EV\_REP, input->evbit);

input\_register\_polled\_device(poll\_dev);

## 16.4 应用层测试程序

E9驱动笔记里面有读取按键的应用程序，可以更改后使用

Include/uapi/linux/input.h

struct input\_event {

struct timeval time;

\_\_u16 type;

\_\_u16 code;

\_\_s32 value;

}

## 16.5 自定义reset键码值添加

内核有两处：一个用于设备树，一个用于内核

Rk-input.h和input-event-codes.h中

添加#define KEY\_RESET 254

编译器头文件

/home/chao/rk3308\_linux/buildroot/output/rockchip\_rk3308\_32\_debug/host/arm-rockchip-linux-gnueabihf/sysroot/usr/include/linux

添加

#define KEY\_RESET 254

编译器头文件添加也可用改在按键源码中添加宏定义。

# 17 串口驱动

## 17.1 配置内核

采用的是8250通用驱动，类型是16550A

Device Drivers ‐‐‐>

Character devices ‐‐‐>

Serial drivers ‐‐‐>

[\*] 8250/16550 and compatible serial support

[ ] Support 8250\_core.\* kernel options (DEPRECATED)

[\*] Console on 8250/16550 and compatible serial port 8250串口开启console功

能

[ ] DMA support for 16550 compatible UART controllers

(5) Maximum number of 8250/16550 serial ports 一般填最大串口数

(5) Number of 8250/16550 serial ports to register at runtime 一般填最大串口数

[ ] Extended 8250/16550 serial driver options

[\*] Support for Synopsys DesignWare 8250 quirks

## 17.2 配置设备树

rk3308-voice-module-v10-aarch32.dtsi文件中配置

&uart0 {

status = "okay";

};

## 17.3 串口配置

### 17.3.1 串口名字

驱动起来后会先注册5个ttySx设备。但如果没有经过2.3.1使能的串口，虽然也有设备节点，但是是不能操作的。

驱动会根据aliase，来对应串口编号，如下： serial0最终会生成ttyS0，serial3会生成ttyS3设备。

aliases {

serial0 = &uart0;

serial1 = &uart1;

serial2 = &uart2;

serial3 = &uart3;

serial4 = &uart4;

};

### 17.3.2 pinctrl配置

有时一个串口有多组IOMUX配置，需要根据实际使用配置

其中uart0\_cts和uart0\_rts是硬件流控脚，这只代表引脚有配置为相应的功能脚，并不代表使能硬件流控。使能硬件流控需要从运用层设置下来。需要注意的是，如果使能流控，uart0\_cts和uart0\_rts必须同时配上。如果不需要流控，可以把uart0\_cts和uart0\_rts。

去掉。

### 17.3.2 DMA使用

需要使用DMA时需要以下配置，如果没有需要自己手动添加：

dma-names = "tx", "rx"; 使能DMA发送和接收

dma-names = "!tx", "!rx"; 禁止DMA发送和接收

dmas = <&dmac\_peri 0>, <&dmac\_peri 1>; 这里的0和1是外设和DMAC连接的通道号，DMAC通过这个号来识别外设。

### 17.3.3波特率配置

目前的代码会根据波特率大小来设置时钟，一般1.5M以下的波特率都可以分出来。1.5M以上的波特率，可能会经过小数分频或整数分频

## 17.4 linux打印串口配置

FIQ debugger, ttyFIQ0设备作为console

使能DTS节点。

rk3308-voice-module-v10-aarch32.dtsi文件中配置

&fiq\_debugger {

status = "okay";

};

rk3308.dtsi文件中配置

fiq\_debugger: fiq‐debugger {

compatible = "rockchip,fiq‐debugger";

rockchip,serial‐id = <2>; /\*设置串口id，如果想换不同的串口就改这个ID\*/

rockchip,wake‐irq = <0>;

rockchip,irq‐mode‐enable = <0>; /\* If 1， uart uses irq instead of fiq \*/

rockchip,baudrate = <1500000>; /\* Only 115200 and 1500000 \*/

pinctrl‐names = "default";

pinctrl‐0 = <&uart2c\_xfer>; /\*换了不同的串口后，需要配置iomux\*/

interrupts = <GIC\_SPI 150 IRQ\_TYPE\_LEVEL\_HIGH 0>; /\* 配置signal irq，一般可以是该SOC最大中断号加1 \*/

status = "disable

};

该节点驱动加载动加载后会注册/dev/ttyFIQ0设备，需要注意的是rockchip,serial-id 即便改了，注册的也是ttyFIQ0。rockchip,irq-mode-enable = <0>; 这个如果为1，串口中断方式采用的是irq，一般不会遇到问题。但如果是0，用的是FIQ模式，有些带有trust firmeware的平台就需要谨慎用，这可能会因为trust firmeware版本和内核版本不匹配出问题。

**使能early printk功能**

添加一下参数，其中0xff1a0000是uart2的物理基地址，不同的串口基地址不一样。

一般后面参数不加115200等波特率，用uboot或loader起来后配置的波特率即可。

如果配了波特率可能会出问题，因为内核early con对这块的支持不是很好。

chosen {

bootargs ="earlycon=uart8250,mmio32,0xff1a0000";

};

# 17.5 测试板子串口和ubuntu串口是否通

stty查看串口参数

stty -F /dev/ttyS0 -a

查看串口1（/dev/ttyS0）当前的参数，包括波特率、数据位等。

stty设置串口参数

stty -F /dev/ttyS0 ispeed 115200 ospeed 115200 cs8

该命令将串口1（/dev/ttyS0）设置成115200波特率，8位数据模式。一般情况下设置这两个参数就可以了，如果显示数据乱码，可能还需要设置其它参数，使用man查看stty其它设置选项。

cat打印串口数据

cat /dev/ttyS0

串口数据就可以在终端上显示了。

**实验：**

ubutu映射的usb转串口名字是ttyUSB0,板子上是ttyS4

ubuntu操作

chao@ubuntu: sudo stty -F /dev/ttyUSB0 -a

chao@ubuntu: sudo stty -F /dev/ttyUSB0 ispeed 115200 ospeed 115200 cs8

chao@ubuntu:~$ sudo su //必须切换到root用户，否则易出错

root@ubuntu:/home/chao# cat /dev/ttyUSB0

“hello”

板子操作

stty -F /dev/ttyS4 -a

stty -F /dev/ttyS4 ispeed 115200 ospeed 115200 cs8

echo “hello” > /dev/ttyS4

## 17.6 通过串口实现GDB 调试

### 17.6.1配置buildroot生成gdb调试工具

│ Symbol: BR2\_PACKAGE\_GDB [=n]

│ Type : boolean

│ Prompt: gdb

│ Location:

│ -> Target packages

│ (3) -> Debugging, profiling and benchmark

│ Defined at package/gdb/Config.in:14

│ Depends on: BR2\_TOOLCHAIN\_HAS\_THREADS [=y] && BR2\_TOOLCHAIN\_HAS\_THREADS\_DEBUG [=y] && BR2\_PACKAGE\_GDB\_ARCH\_SUPPORTS

│ Selects: BR2\_PACKAGE\_GDB\_SERVER [=n]

选中gdb选项

生成gdbserver 命令工具和arm-linux-gdb

ubuntu安装adb

sudo apt-get install android-tools-adb

## 17.6.2 gdb调试测试

gdb调试之前必须先配置串口，按17.5的操作进行配置波特率

发现串口调试非常不稳定，容易连不上

板子上面执行命令

# gdbserver /dev/ttyS4 /oem/agdb\_example

ubuntu执行如下命令

arm-linux-gdb /home/chao/code/gdb/agdb\_example

target remote /dev/ttyUSB0

下面是在ubuntu调试的例子

root@ubuntu:/home/chao# arm-linux-gdb /home/chao/code/gdb/agdb\_example

GNU gdb (GDB) 7.11.1

Copyright (C) 2016 Free Software Foundation, Inc.

License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>

This is free software: you are free to change and redistribute it.

There is NO WARRANTY, to the extent permitted by law. Type "show copying"

and "show warranty" for details.

This GDB was configured as "--host=x86\_64-pc-linux-gnu --target=arm-rockchip-linux-gnueabihf".

Type "show configuration" for configuration details.

For bug reporting instructions, please see:

<http://www.gnu.org/software/gdb/bugs/>.

Find the GDB manual and other documentation resources online at:

<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".

Type "apropos word" to search for commands related to "word"...

Reading symbols from /home/chao/code/gdb/agdb\_example...done.

(gdb) target remote /dev/ttyUSB0

/dev/ttyUSB0: 输入/输出错误.

(gdb) target temote /dev/ttyUSB0

Undefined target command: "temote /dev/ttyUSB0". Try "help target".

(gdb) target remote /dev/ttyUSB0

Remote debugging using /dev/ttyUSB0

Reading /lib/ld-linux-armhf.so.3 from remote target...

warning: File transfers from remote targets can be slow. Use "set sysroot" to access files locally instead.

Reading /lib/ld-linux-armhf.so.3 from remote target...

Reading symbols from target:/lib/ld-linux-armhf.so.3...(no debugging symbols found)...done.

0xb6fd79e0 in \_start () from target:/lib/ld-linux-armhf.so.3

(gdb)

(gdb)

(gdb) list 1

1 int add(int a,int b)

2 {

3 return a\*b;

4 }

5 int main()

6 {

7 int sum[10]={0,0,0,0,0,0,0,0,0,0,};

8 int i;

9 int array1[10]={48,56,77,33,33,11,226,544,78,90};

10 int array2[10]={85,99,66,0x199,393,11,1,2,3,4};

(gdb) lisnt 11

Undefined command: "lisnt". Try "help".

(gdb) list 11

6 {

7 int sum[10]={0,0,0,0,0,0,0,0,0,0,};

8 int i;

9 int array1[10]={48,56,77,33,33,11,226,544,78,90};

10 int array2[10]={85,99,66,0x199,393,11,1,2,3,4};

11 for(i=0;i<10;i++)

12 {

13 sum[i]=add(array1[i],array2[i]);

14 }

15 }

(gdb) b 3

Breakpoint 1 at 0x10392: file gdb\_example.c, line 3.

(gdb) b 13

Breakpoint 2 at 0x1040c: file gdb\_example.c, line 13.

(gdb) c

Continuing.

Reading /lib/libc.so.6 from remote target...

Breakpoint 2, main () at gdb\_example.c:13

13 sum[i]=add(array1[i],array2[i]);

(gdb)

Continuing.

Breakpoint 1, add (a=48, b=85) at gdb\_example.c:3

3 return a\*b;

(gdb) print a

$1 = 48

(gdb) print b

$2 = 85

(gdb) c

Continuing.

Breakpoint 2, main () at gdb\_example.c:13

13 sum[i]=add(array1[i],array2[i]);

(gdb) print sum

$3 = {4080, 0, 0, 0, 0, 0, 0, 0, 0, 0}

(gdb)

(gdb) c

Continuing.

Breakpoint 1, add (a=56, b=99) at gdb\_example.c:3

3 return a\*b;

(gdb) print a

$4 = 56

(gdb) print b

$5 = 99

(gdb) c

Continuing.

Breakpoint 2, main () at gdb\_example.c:13

13 sum[i]=add(array1[i],array2[i]);

(gdb) print sum

$6 = {4080, 5544, 0, 0, 0, 0, 0, 0, 0, 0}

(gdb)

# 18 音频驱动

## 18.1 功率放大器

Cs8508E：单声道音频功率放大器。

AB/D类切换。

SD:掉电控制管脚，高电平有效。低电平用于设置静音模式？

ABD:AB类/D类切换选择，L选择AB类模式，H选择D类模式

IN+:音频输入正端。

IN-:音频输入负端.

VDD:电源

VO+:正相音频输出。接喇叭

VO-:反相音频输出。接喇叭

GND

原理图 芯片引脚

LINEOUT\_L：CODEC\_LINEOUT\_L

LINEOUT\_R：CODEC\_LINEOUT\_R

MICP8 : CODEC\_MICP8

MICN8 : CODEC\_MICN8

MICP7 : CODEC\_MICP7

MICN7 : CODEC\_MIC7N

MICBIAS1: CODEC\_MICBIAS1

MICBIAS2：CODEC\_MICBIAS2

## 18.2 CODEC配置

### 18.2.1 UIOT板子codec配置

adc共8个通道，2个一组

rk3308-voice-module-board-v10-aarch32.dts

&acodec {

status = "okay";

rockchip,no-deep-low-power;

/\* ADC7-8 for 2ch capture, loopback to ADC3-4 \*/

rockchip,adc-grps-route = <3 1 2 0>;//第4组adc是MIC,第一组adcloopback

/\* The grp2 (ADC5-6) keeps enabled always for VAD \*/

rockchip,en-always-grps = <3>;

};

rk3308-voice-module-v10-aarch32.dtsi

vad-sound {

status = "okay";

compatible = "rockchip,multicodecs-card";

rockchip,card-name = "rockchip,rk3308-vad";

rockchip,codec-hp-det;

rockchip,cpu = <&i2s\_8ch\_2>;

rockchip,codec = <&acodec>, <&vad>;

};

&acodec {

status = "disabled";

rockchip,no-hp-det; //耳机检测功能禁止

rockchip,delay-start-play-ms = <200>;

rockchip,loopback-grp = <1>;//回采电路连接在2\_3

#ifdef UIOT

//spk-ctl-gpios = <&gpio1 RK\_PC5 GPIO\_ACTIVE\_HIGH>;

去掉该项，单独控制。

#else

spk-ctl-gpios = <&gpio0 RK\_PA5 GPIO\_ACTIVE\_HIGH>;

#endif

};

&vad {

status = "okay";

rockchip,audio-src = <&i2s\_8ch\_2>;

rockchip,det-channel = <0>;

rockchip,mode = <1>;

rockchip,buffer-time-ms = <500>;

#sound-dai-cells = <0>;

};

/etc/asound.conf里面配置，选择了2通道，2mic\_loopback就是mic设备名字

pcm.multi\_2\_2 {

type multi

slaves.a.pcm "hw:0,0"

slaves.a.channels 4

bindings.0.slave a

bindings.0.channel 0

bindings.1.slave a

bindings.1.channel 1

bindings.2.slave a

# bindings.2.channel 2

bindings.2.channel 3

bindings.3.slave a

# bindings.3.channel 3

bindings.3.channel 2

}

pcm.2mic\_loopback

{

type plug

# slave.pcm "multi\_2\_1"

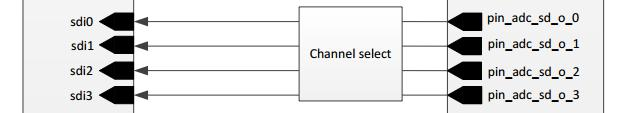
slave.pcm "multi\_2\_2"

}

### 18.2.2 rk3308 codec设备树配置知识

RK3308 CODEC 的属性比较多， dts 里能使用到的属性都可以在 kernel 代码的：  
Documentation/devicetree/bindings/sound/rockchip,rk3308-codec.txt  
找到， 这里介绍一些平时开发中常用的属性， 以便于理解：

● rockchip,adc-grps-route  
这个属性是可选项， 可以调整各路 ADC 与 i2s sdi 的映射关系。 注意， 这里不是指 MIC  
和 ADC 的映射， 是 CODEC IP 出来的 sdo 与连接的 i2s IP 的 sdi 的映射关系。 具体框图可以参考 RK3308 TRM Chapter 6 Audio Subsystem：



如果不指定， 默认为一一映射的关系。 比如 CODEC 和 i2s\_8ch\_2 连接， 录制 8ch 时。  
ADC\_MIC 0-7 分别对应 i2s\_8ch\_2 的 sdi 0-7:  
\* sdi\_0 <-- sdo\_0 <-- MIC\_0\_1  
\* sdi\_1 <-- sdo\_1 <-- MIC\_2\_3  
\* sdi\_2 <-- sdo\_2 <-- MIC\_4\_5  
\* sdi\_3 <-- sdo\_3 <-- MIC\_6\_7  
如果有这样一个场景， 我们希望再录制 4ch 的时候， MIC\_2\_3 移到最后作为回采功能，  
MIC\_4\_5 移到最前面， dts 里可以增加这样的描述：  
rockchip,adc-grps-route = <2 1 3 0>;  
此时的 ADC 通路的映射关系：  
\* sdi\_0 <-- sdo\_1 <-- MIC\_2\_3 // ch0 and ch1  
\* sdi\_1 <-- sdo\_0 <-- MIC\_0\_1 // ch2 and ch3  
\* sdi\_2 <-- sdo\_3 <-- MIC\_6\_7 // not used  
\* sdi\_3 <-- sdo\_2 <-- MIC\_4\_5 // not used  
由于我们只用到 4ch， i2s 的 sdi2 和 sdi3 并没有被使用到， 所以后面 2 租的描述可以忽略。

● rockchip,loopback-grp  
这个属性指定的是模拟 PA 对应的连接的 ADC group， 通过这个属性， codec driver  
会在合适的时间打开回采， 以节省功耗。 比如， 回采电路连接在 ADC\_0\_1 上， 我们可以在dts 指定：  
rockchip,loopback-grp = <0>;

● rockchip,no-hp-det  
该属性表明 CODEC 就不会去使能 hp-det 的功能。 如果目标板硬件上没有用 CODEC的耳机检测功能， CODEC hp-det pin 悬空， 该属性强烈建议加上， 否则会引起耳机插入误报的现象。

● hp-ctl-gpios  
该属性指定了控制耳机通路的 gpio pin。 在耳机通路使能下， 播放/关闭音乐的时候，  
打开/关闭耳机通路模块。 比如：  
hp-ctl-gpios = <&gpio0 1 GPIO\_ACTIVE\_HIGH>;

● spk-ctl-gpios  
该属性指定了控制喇叭通路的 gpio pin。 在喇叭通路使能下， 播放/关闭音乐的时候，  
打开/关闭 PA 通路模块。 比如：  
spk-ctl-gpios = <&gpio0 5 GPIO\_ACTIVE\_HIGH>;

● rockchip,en-always-grps  
该属性可以让指定的 ADC group 打开一次之后就常开， 主要应用于与 VAD 配合的场  
景， 即在休眠的时候不关闭与 VAD 相关的 ADC， 达到快速响应的功能。

● rockchip,delay-loopback-handle-ms  
因为在实际开发过程中， 选用的 PA 的启动时延不同。 该属性指定了打开回采后， 需要等待的稳定时延才重新打开对应的 ADC， 避免回采数据抖动。 比如：

rockchip,delay-loopback-handle-ms = <200>;  
● rockchip,no-deep-low-power  
该属性表明在系统休眠的时候 CODEC 不进入低功耗模式， 以适应更快速的响应需求，  
适合对功耗不是很在意的场景。

### 18.2.3 Asound.conf 配置说明

**asound.conf 的作用**  
asound.conf 根据应用需求对 kernel 中的声卡进行配置， 常见的配置有放音增益设置和录音声道的转换， Alsa 通过 PCM 插件来实现所需配置， 本文重点讲解插件如下：  
Asym 插件  
Hooks 插件  
Soft volume 插件  
dmix 插件  
Ladspa 插件  
Multi 插件  
Plug 插件  
ctl\_elems 插件  
Dsnoop 插件  
Dshare 插件

**Asym 插件配置 arecord/ aplay 默认播放设备**pcm.!default

{

type asym

playback.pcm {

type plug

slave.pcm "softvol"

}

capture.pcm {

type plug

slave {

pcm "hw:0,0"

}

}

}  
Playback.pcm 是播放 plug 插件， 依赖设备 softvol（第一节中所描述插件） ， capture.pcm 是录音 plug 插件， 依赖设备是 hw:0,0 该设备可以通过 arecord -l 命令去查看。

**Softvol 插件设置 PCM 增益的范围和分辨率**

pcm.softvol {

type softvol

slave.pcm "playback"

control {

name "Master Playback Volume" //这就是播放音量的名字

card 0

}

min\_dB -40.0

max\_dB 0.0

resolution 100

}

Softvol 插件依赖 playback 设备， 其作用用来设置增益的范围和分辨率  
min\_dB -40.0 最小增益  
max\_dB 0.0 最大增益 ， 可以大于 0DB  
resolution 100 分辨率 100 等份  
备注： 该增益调节的仅仅是 PCM 的数字增益。

**Dmix 插件用来配置播放参数【虚拟声卡】**

pcm.playback {

type dmix

ipc\_key 5978293 # must be unique for all dmix plugins!!!!

ipc\_key\_add\_uid yes

slave {

pcm "hw:7,0,0" //播放设备名字

channels 2 //指定通道数

period\_size 1024

buffer\_size 4096

}

bindings {

0 0

1 1

}

}

Playback 设备依赖 hw:7,0,0(该设备是内核虚拟的声卡设备， 和实际的硬件未发生关联), 可以通过aplay -l 去查看  
channels 2 ---通道数  
period\_size 1024 ----单次传输 1024 个字节  
buffer\_size 4096 -----总 BUFFER 大小  
bindings {  
0 0 --- playback 0 通道映射 hw:7,0,0 0 通道  
1 1 --- playback 1 通道映射 hw:7,0,0 1 通道  
}

**Plug 插件提供一个录音接口**  
pcm.fake\_record {  
type plug  
slave.pcm "hw:7,1,0"  
}

后台启动一个进程， 从 fake\_record 录制声音， 该插件的依赖设备是虚拟声卡。

**Plug 插件提供一个放音接口**  
pcm.fake\_play {  
type plug  
slave.pcm "ladspa"  
}

第四节提到的后台进程， 从 fake\_record 录制的声音送到 fake\_play

**Ladspa 插件实现自定义算法**  
pcm.ladspa {  
type ladspa  
slave.pcm "plug:real\_playback"  
channels 2  
path "/usr/lib"  
playback\_plugins [{  
label eq\_drc\_stereo  
input {  
controls [4]  
}  
}]  
}  
如果想拦截送往声卡 PCM 经过自定义算法， 可以通过这个插件来配置  
plug:real\_playback---- 依赖放音插件 real\_playback  
channels 2 ----算法的通道数  
path "/usr/lib" ---- 算法库所在的路径  
label eq\_drc\_stereo --- 算法库名  
controls [4] --- 参数

**Dshare 插件配置播放参数【物理声卡】**

pcm.real\_playback {  
type dshare  
ipc\_key 5978293 # must be unique for all dmix plugins!!!!  
ipc\_key\_add\_uid yes  
slave {  
pcm "hw:0,0"  
channels 2  
rate 48000  
period\_size 1024  
buffer\_size 4096  
}b  
indings {  
0 0  
1 1  
}  
}  
real\_playback 设备依赖 hw:0,0 可以通过 aplay -l 去查看  
channels 2 ---通道数  
period\_size 1024 ----单次传输 1024 个字节  
buffer\_size 4096 -----总 BUFFER 大小  
bindings {  
0 0 --- real\_playback 0 通道映射 hw:0,0 0 通道  
1 1 --- real\_playback 1 通道映射 hw:0,0 1 通道  
}

**multi 插件来实现各种录音的通道的组合**  
上层算法使用的顶层录音节点是：  
2mic\_loopback  
4mic\_loopback  
6mic\_loopback  
目前已经有的 multi 插件设备如下：  
multi\_2\_2（2MIC + 2 LOOPBACK）  
multi\_2\_1 (2MIC + 1 LOOPBACK)  
multi\_4\_1(4MIC + 1 LOOPBACK)  
multi\_8 (6MIC + 2 LOOPBACK) ---- 目前 KERNEL DTS 将多通道的虚拟成一个声卡， 因此该插件仅作参考。

接下来以 multi\_2\_1 和 multi\_8 为做详细讲解。

pcm.multi\_2\_1 {

type multi ------ 插件类型

slaves.a.pcm "hw:0,0" ------ a 依赖设备

//通道是4原因是有两个mic和一个loopback,三个通道，必须是偶数所以就是4

slaves.a.channels 4 ------ a 设备的通道， 该通道必须是偶数， 且小于依赖设备的通道

bindings.0.slave a

bindings.0.channel 0 ---- multi\_2\_1 的 0 通道 对应 a 设备的 0 通道

bindings.1.slave a

bindings.1.channel 1 ---- multi\_2\_1 的 1 通道 对应 a 设备的 1 通道

bindings.2.slave a

bindings.2.channel 2 ---- multi\_2\_1 的 2 通道 对应 a 设备的 2 通道

}

**使用 dsnoop 插件来配置录音参数**  
有些应用往往不设置录音参数， 使用默认参数会导致底层 BUFFER 开的太大， DMA 出错。 所以可以使用 dsnoop 插件来设置录音参数。

pcm.dsnooped\_6\_2 {  
ipc\_key 1027  
type dsnoop  
slave {  
pcm "hw:0,0"  
channels 8  
rate 16000  
period\_size 1024  
buffer\_size 4096  
}  
}

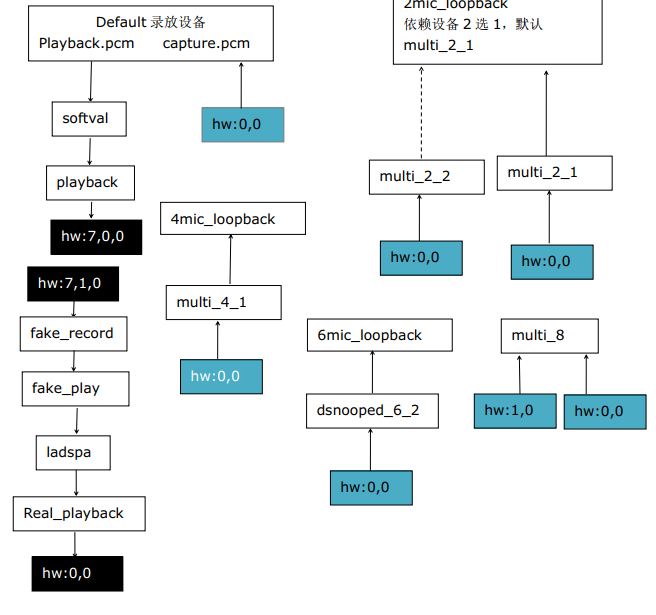
Ipc\_key 1027 ---- dsnoop 插件多进程共享， 需要一个共享内存， 该值代表共享内存 ID， 每个 dsnoop  
插件必须分配一个唯一的 ipc\_key  
type dsnoop ---- 指定该插件的类型为 dsnoop  
pcm "hw:0,0" ---- 从设备为 hw:0,0  
channels 8 --- 通道为 8  
rate 16000 --- 采样率 16K  
period\_size 1024 --- DMA 中断一次， 传输 1024 个采样点  
buffer\_size 4096 --- BUFFER SIZE， 访问 ALSA 接口使用的 BUFFER 大小， 单位字节。

物理 MIC 对应的录音声卡通道， 请参考 DTS 相关配置说明。

**Rk3308 asound.conf 各种插件关系图**

其中蓝色 hw:0,0 是录音声卡， 通过 arecord -l 查看

黑色 hw:0,0, hw:7,0,0 是放音声卡， 通过 aplay -l 查看



## 18.3 录音播放

模拟麦克风和模拟功放。使用的是ADC采样音频信号，DAC播放声音文件。

四个通道，实际是3个通道，两个mic和一个回采。成组配置所以是4

使用alsa录放音频文件

arecord -D 2mic\_loopback -c 4 -r 16000 -f S16\_LE /tmp/hetangyuese.wav

aplay -D "hw:7,0,0" hetangyuese.wav -c 4 -r 16000 -f S16\_LE

使用tinyalsa录放音频文件

● 播放命令

播放 wav 文件输出到声卡 0 上：  
tinyplay /data/2k\_1k0db\_48K.wav -D 2mic\_loopback

● 录音命令

录制采样率 44100Hz、 位深 16bit、 双声道格式的 wav 文件到声卡 0 上：  
tinycap /tmp/my\_record.wav -r 44100 -b 16 -D 2mic\_loopback -c 2

//-D指定设备，-r采样率，-b位数，-c 通道数

● 同时录放回采命令  
RK3308 EVB 支 持 将 Lineout 2CH 输 出 的 同 时 ， loopback 到 内 置 CODEC  
ADC7/ADC8 通道。 我们可以在后台启动一个 arecord 进程的同时， 去播放需要回采的 wav文件：  
arecord -D 2mic\_loopback -c 4 -r 16000 -f S16\_LE /tmp/loopback.wav &

aplay /tmp/yijiangnan.wav

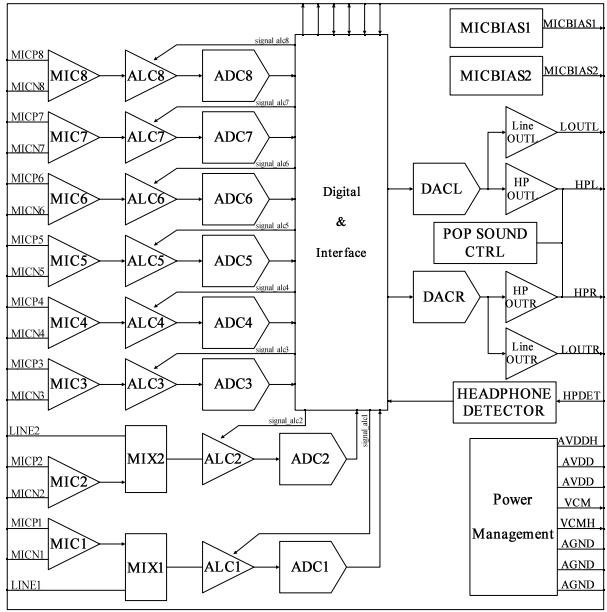
## 18.4 CODEC 音频增益调节

ADC增益配置就是采样信号的放大倍数，增益越大，录音声音越大。

音量调节DAC放大倍数。

RK3308 CODEC 内部支持多级输入/输出增益调节， 它们的所有状态都可以用 amixer

或者 tinymix 工具来查看。使用 amixer contents 查看， 输出的内容比较多， 不是很直观：我们可以使用 tinymix contents， 这样每个通道的每个增益范围都一目了然。



RK3308 CODEC 包含了 8 个 ADC 输入， 将其分组的话， ADC1/ADC2作为 Group 0， ADC3/ADC4 作为 Group 1， ADC5/ADC6 作为 Group 2， ADC7/ADC8作为 Group 3。  
“ALC AGC”前缀的表示对 AGC 自动调节增益的配置， 以及调节对应通道的“Max”和  
“Min”部分， 来控制 AGC 的范围。 目前默认 AGC 是关闭状态， 所以可以暂时忽略这些  
 Volume。  
“ADC MIC”前缀表示调节前级 MIC PGA 线性放大增益。  
“ADC ALC”前缀表示调节后级 ALC 线性放大增益。

“DAC LINEOUT”前缀表示调节后级 LINEOUT 线性放大增益。  
“DAC HPOUT”前缀表示调节后级 HPOUT 线性放大增益。  
“DAC HPMIX”前缀表示调节前级线性放大增益， 它作为 LINEOUT 和 HPOUT 共同的  
前级。  
另外， 每个通道都有各自的调节范围， 比如：

示”ADC ALC Group 0 Left Volume”量化了 0～ 31 共 32 个量化等级， 当前的  
Volume 值是 12。 如果我们想改变这个 Volume 为 20， 可以通过 tinymix set 去设置它：  
# tinymix set "ADC ALC Group 0 Left Volume" 20  
通过 tinymix get 可以去读取指定 Volume 的状态：  
# tinymix get "ADC ALC Group 0 Left Volume"  
20 (range 0->31)

# tinymix contents

Number of controls: 52

ctl type num name value

0 INT 1 ALC AGC Group 0 Left Volume 12 (range 0->31)

1 INT 1 ALC AGC Group 0 Right Volume 12 (range 0->31)

2 INT 1 ALC AGC Group 1 Left Volume 12 (range 0->31)

3 INT 1 ALC AGC Group 1 Right Volume 12 (range 0->31)

4 INT 1 ALC AGC Group 2 Left Volume 12 (range 0->31)

5 INT 1 ALC AGC Group 2 Right Volume 12 (range 0->31)

6 INT 1 ALC AGC Group 3 Left Volume 12 (range 0->31)

7 INT 1 ALC AGC Group 3 Right Volume 12 (range 0->31)

8 INT 1 ALC AGC Group 0 Left Max Volume 7 (range 0->7)

9 INT 1 ALC AGC Group 0 Right Max Volume 7 (range 0->7)

10 INT 1 ALC AGC Group 1 Left Max Volume 7 (range 0->7)

11 INT 1 ALC AGC Group 1 Right Max Volume 7 (range 0->7)

12 INT 1 ALC AGC Group 2 Left Max Volume 7 (range 0->7)

13 INT 1 ALC AGC Group 2 Right Max Volume 7 (range 0->7)

14 INT 1 ALC AGC Group 3 Left Max Volume 7 (range 0->7)

15 INT 1 ALC AGC Group 3 Right Max Volume 7 (range 0->7)

16 INT 1 ALC AGC Group 0 Left Min Volume 0 (range 0->7)

17 INT 1 ALC AGC Group 0 Right Min Volume 0 (range 0->7)

18 INT 1 ALC AGC Group 1 Left Min Volume 0 (range 0->7)

19 INT 1 ALC AGC Group 1 Right Min Volume 0 (range 0->7)

20 INT 1 ALC AGC Group 2 Left Min Volume 0 (range 0->7)

21 INT 1 ALC AGC Group 2 Right Min Volume 0 (range 0->7)

22 INT 1 ALC AGC Group 3 Left Min Volume 0 (range 0->7)

23 INT 1 ALC AGC Group 3 Right Min Volume 0 (range 0->7)

24 INT 1 ADC MIC Group 0 Left Volume 0 (range 0->3)

25 INT 1 ADC MIC Group 0 Right Volume 0 (range 0->3)

26 INT 1 ADC MIC Group 1 Left Volume 0 (range 0->3)

27 INT 1 ADC MIC Group 1 Right Volume 0 (range 0->3)

28 INT 1 ADC MIC Group 2 Left Volume 0 (range 0->3)

29 INT 1 ADC MIC Group 2 Right Volume 0 (range 0->3)

30 INT 1 ADC MIC Group 3 Left Volume 0 (range 0->3)

31 INT 1 ADC MIC Group 3 Right Volume 0 (range 0->3)

32 INT 1 ADC ALC Group 0 Left Volume 12 (range 0->31)

33 INT 1 ADC ALC Group 0 Right Volume 12 (range 0->31)

34 INT 1 ADC ALC Group 1 Left Volume 12 (range 0->31)

35 INT 1 ADC ALC Group 1 Right Volume 12 (range 0->31)

36 INT 1 ADC ALC Group 2 Left Volume 12 (range 0->31)

37 INT 1 ADC ALC Group 2 Right Volume 12 (range 0->31)

38 INT 1 ADC ALC Group 3 Left Volume 12 (range 0->31)

39 INT 1 ADC ALC Group 3 Right Volume 12 (range 0->31)

40 ENUM 1 ADC Group 0 HPF Cut-off 20Hz245Hz612Hz, Off

41 ENUM 1 ADC Group 1 HPF Cut-off 20Hz245Hz612Hz, Off

42 ENUM 1 ADC Group 2 HPF Cut-off 20Hz245Hz612Hz, Off

43 ENUM 1 ADC Group 3 HPF Cut-off 20Hz245Hz612Hz, Off

44 INT 1 DAC LINEOUT Left Volume 0 (range 0->3)

45 INT 1 DAC LINEOUT Right Volume 0 (range 0->3)

46 INT 1 DAC HPOUT Left Volume 0 (range 0->30)

47 INT 1 DAC HPOUT Right Volume 0 (range 0->30)

48 INT 1 DAC HPMIX Left Volume 0 (range 0->1)

49 INT 1 DAC HPMIX Right Volume 0 (range 0->1)

50 BOOL 1 vad switch Off

51 INT 2 Master Playback Volume 99, 99 (range 0->99)

LINEOUT接功放，HPOUT接耳机。

调节输出音量

# tinymix get 51 //获取当前音量大小

# tinymix set 48 1 //输出前级增益

# tinymix set 49 1

# tinymix set 44 3 //输出后级增益

# tinymix set 45 3

# tinymix set 51 50 //调节输出音量0-99

aplay -vv /tmp/yijiangnan.wav //播放测试音乐

Vad 开关

# tinymix get 50

off

# tinymix set 50 1 //打开vad

调节loopback

为什么需要Loopback？假如codec在播放音乐，而此时有人说话。那么说话声音和播放音乐的声音都会被mic采集到，这时候就会影响到识别效果。Loopback就是把codec播放的音乐在内部电路中采样，用mic的采集减去Loopback的采样就是人声音。

Loopback用的是ADC3,通道编号2

# tinymix set 2 16

# tinymix set 26 3

# tinymix set 34 18

调节Mic增益

Rk公司不建议修改ALC AGC参数，因此最终保持默认12，ADC MIC 配置3，ADC ALC配置为

22，再大就会破音。

使用ADC的7和8，对应通道号为6和7

# tinymix set 6 16 //ALC AGC Group 0，范围0-31最大16，再大破音

# tinymix set 7 16

# tinymix set 30 3 //ADC MIC Group 3，范围0-3

# tinymix set 31 3

# tinymix set 38 18 //ADC ALC Group 3，范围0-31,最大18，20破音

# tinymix set 39 18

arecord -D 2mic\_loopback -c 2 -r 44100 -f S16\_LE /tmp/1.wav

S是有符号   U是无符号  
BE是大端（低地址存高位）

LE是小端（低地址存低位）

**调节音量**  
amixer cset name='Master Playback Volume' 99

范围0-99  
为便于应用控制音量， rk3308 SDK 在 ALSA 配置文件 asound.conf 增加了软件音量调节  
插件， 即在 alsa control 中增加了一个 mixer： ”Master Playback Volume”， 如下图所示。  
pcm.softvol {  
type softvol  
slave.pcm "playback"  
control {  
name "Master Playback Volume"  
card 0  
} m  
in\_dB -40.0  
max\_dB 0.0  
resolution 100

**CODEC** 高通滤波器配置

//截止频率有20Hz245Hz612Hz三种，下面设置截止频率为20HZ

# amixer sset 'ADC Group 3 HPF Cut-off' 20Hz

Simple mixer control 'ADC Group 3 HPF Cut-off',0

Capabilities: enum

Items: '20Hz' '245Hz' '612Hz' 'Off'

Item0: '20Hz'

# amixer sget 'ADC Group 3 HPF Cut-off'

Simple mixer control 'ADC Group 3 HPF Cut-off',0

Capabilities: enum

Items: '20Hz' '245Hz' '612Hz' 'Off'

Item0: '20Hz'

## 18.5 静音控制驱动

//mute\_creat.c 创建gpio用户层操作接口

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<fcntl.h>

#include<sys/types.h>

#include<sys/stat.h>

#include<sys/times.h>

#include<sys/select.h>

#include<signal.h>

#include<string.h>

static void mute\_init()

{

int export\_fd, direction\_fd;

char \*gpionum = "53";

export\_fd = open("/sys/class/gpio/export", O\_WRONLY);

if(-1 == export\_fd){

printf("open export error\n");

exit(1);

}

if(-1 == write(export\_fd, gpionum, strlen(gpionum))){

printf("write file operation error\r\n");

exit(1);

}

close(export\_fd);

//gpio53 is a softlink to /sys/devices/platform/pinctrl/gpio/gpio53

direction\_fd = open("/sys/class/gpio/gpio53/direction",O\_WRONLY);

if(-1 == direction\_fd){

printf("open direction error\n");

exit(1);

}

if(-1 == write(direction\_fd, "out", strlen("out"))){

close(direction\_fd);

exit(1);

}

close(direction\_fd);

}

int main(int argc,char \*argv[])

{

mute\_init();

return 0;

}

//mute\_ctrl.c 通过gpio用户层接口控制gpio输出电平

#include<stdio.h>

#include<stdlib.h>

#include<unistd.h>

#include<fcntl.h>

#include<sys/types.h>

#include<sys/stat.h>

#include<sys/times.h>

#include<sys/select.h>

#include<signal.h>

#include<string.h>

void mute\_ctrl(char value)

{

int gpiovalue\_fd,ret;

gpiovalue\_fd = open("/sys/class/gpio/gpio53/value", O\_RDWR);

if(-1 == gpiovalue\_fd){

printf("open gpiovalue error\n");

exit(1);

}

if(value == 1)

ret = write(gpiovalue\_fd,"1",1);

else if(value == 0)

ret = write(gpiovalue\_fd,"0",1);

if(ret == -1)

printf("write gpiovalue\_fd error\n");

close(gpiovalue\_fd);

}

int main(int argc,char \*argv[])

{

char value;

if (argc != 2){

printf("Usage:\n");

printf("%s 0/1\n", argv[0]);

return -1;

}

value = \*argv[1]-0x30;

printf("value:%x",value);

mute\_ctrl(value);

return 0;

}

**实验：**

./mute\_creat

./mute\_ctrl 1 //非静音

aplay /oem/tts\_sample.wav

./mute\_ctrl 0 //静音

aplay /oem/tts\_sample.wav

# 19 新建nandflash绝对地址分区

Parameter里面配置分区表，预留最后4k空间。

FIRMWARE\_VER:8.1

MACHINE\_MODEL:RK3308

MACHINE\_ID:007

MANUFACTURER: RK3308

MAGIC: 0x5041524B

ATAG: 0x00200800

MACHINE: 3308

CHECK\_MASK: 0x80

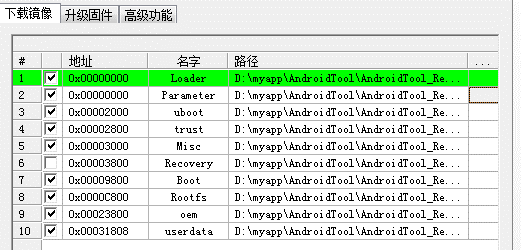
PWR\_HLD: 0,0,A,0,1

TYPE: GPT

CMDLINE:mtdparts=rk29xxnand:0x00000800@0x00002000(uboot),0x00000800@0x00002800(trust),0x00000800@0x00003000(misc),0x00006000@0x00003800(recovery),0x00003000@0x00009800(boot),0x00017000@0x0000C800(rootfs),0x0000E000@0x00023800(oem),0x0000E7F8@0x0031808(userdata:grow)

uuid:rootfs=614e0000-0000-4b53-8000-1d28000054a9

0x31808之前预留了8\*512=4KB的空间。



# 20 音频学习

## 20.1 ALSA 学习

### 20.1.1 ALSA名词解释

channel

通道，即我们熟知的声道数。左/右声道，5.1channel等等

sample

A sample is a single value that describes the amplitude of the audio signal at a single point in time, on a *single channel*.sample即一次采样，通常的sample bit指的是一个channel上，一次采样的bit数（常见的sample bit 8/16/24/32bits）

即采样值或取样值，用来衡量声音波动变化的参数，是指声卡在采集和播放声音文件时所使用数字声音信号的二进制位数。声卡的位客观地反映了数字声音信号对输入声音信号描述的准确程度。 声卡的主要的作用之一是对声音信息进行录制与回放，在这个过程中采样的位数和采样的频率决定了声音采集的质量。8位代表2的8次方——256，16位则代表2的16次方——64K。比较一下，一段相同的音乐信息，16位声卡能把它分为64K个精度单位进行处理，而8位声卡只能处理256个精度单位，造成了较大的信号损失，最终的采样效果自然是无法相提并论的。

frame

When we talk about working with digital audio, we often want to talk about the data that represents all channels at a single point in time. This is a collection of samples, one per channel, and is generally called a "frame". When we talk about the passage of time in terms of frames, its roughly equivalent to what people when they measure in terms of samples, but is more accurate; more importantly, when we're talking about the amount of data needed to represent all the channels at a point in time, its the only unit that makes sense.

一个frame是一次采样时所有channel上的sample bit.即frame = channels \* (sample bit).

**采样频率**

[采样频率](https://baike.baidu.com/item/%E9%87%87%E6%A0%B7%E9%A2%91%E7%8E%87)是指录音设备在一秒钟内对声音信号的采样次数，采样频率越高声音的还原就越真实越自然。在当今的主流声卡上，采样频率一般共分为11KHz、22.05KHz、44.1KHz三个等级，22.05只能达到FM广播的声音品质，44.1KHz则是理论上的CD音质界限，48KHz则更加精确一些。对于高于48KHz的采样频率人耳已无法辨别出来了，所以在电脑上没有多少使用价值。

peroid

A period is the number of frames in between each hardware interrupt.

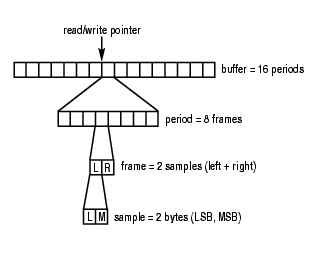
每当hardware buffer 中有peroid size个frame的空间时，硬件就产生中断，来通知alsa driver来往硬件写数据。音频设备一次处理所需要的桢数，对于音频设备的数据访问以及音频数据的存储，都是以此为单位。

buffer size

This determines how large the hardware buffer is.

hardware buffer size 是由多个peroid组成。buffer size = peroid size \* peroids。

声卡接口中有一个指针来指示声卡硬件缓存区中当前的读写位置。只要接口在运行，这个指针将循环地指向缓存区中的某个位置。  
frame size = sizeof(one sample) \* nChannels  
alsa中配置的缓存(buffer)和周期(size)大小在runtime中是以帧(frames)形式存储的。  
period\_bytes = frames\_to\_bytes(runtime, runtime->period\_size);   
bytes\_to\_frames()



Data access and layout

interleaveda: data layout arrangement where the samples of each channel that will be played at the same time follow each other sequentially. See "non-interleaved"non-interleaveda: data layout where the samples for a single channel follow each other sequentially; samples for another channel are either in another buffer or another part of this buffer. Contrast with "interleaved"

在一个period以内(interleaved和non-interleaved是在一个period里面排)，数据是按照channel1排完了再排channel2呢，还是一个frame一个frame的来排（frame在alsa里指的是一次采样时间内，两个channel的数据放一块儿就是一个frame）上图是interleaved。

交错模式(interleaved)：是一种音频数据的记录方式，在交错模式下，数据以连续桢的形式存放，即首先记录完桢1的左声道样本和右声道样本（假设为立体声格式），再开始桢2的记录。而在非交错模式下，首先记录的是一个周期内所有桢的左声道样本，再记录右声道样本，数据是以连续通道的方式存储。不过多数情况下，我们只需要使用交错模式就可以了。

Hardware parameter

These are parameters that directly affect the hardware of the audio interface.

 Hardware parameter是作用于声卡硬件的，包括sample rate, sample format, interupt intervals, data access and layout, buffer size

Software parameter

These are parameters that control the operation of the device driver rather than the hardware itself. Most programs that use the ALSA Audio API will not need to set any of these; a few will need set a subset of them.

software parameter 是作用于alsa core的。通常用来控制When to start the device, what to do about xruns, Available minimum space/data for wakeup,Transfer chunk size.

**When to start the device**

When you open the audio interface, ALSA ensures that it is not active - no data is being moved to or from its external connectors. Presumably, at some point you want this data transfer to begin. There are several options for how to make this happen.

 The control point here the start threshold, which defines the number of frames of space/data necessary to start the device automatically. If set to some value other than zero for playback, it is necessary to prefill the playback buffer before the device will start. If set to zero, the first data written to the device (or first attempt to read from a capture stream) will start the device.

 You can also start the device explicitly using snd\_pcm\_start, but this requires buffer prefilling in the case of the playback stream. If you attempt to start the stream without doing this, you will get -EPIPE as a return code, indicating that there is no data waiting to deliver to the playback hardware buffer.

我们可以通过API snd\_pcm\_sw\_params\_set\_start\_threshold来设置什么时候开始启动声卡。对于playback，假设第设置start threshold 为320,那么就是说，当用户调用writei，写入的数据，将暂时存在alsa驱动空间里，当这个数据量达到 320帧时，alsa驱动才开始将数据写入hardware buffer，并启动DA转换。

**What to do about xruns**

If an xrun occurs, the device driver may, if requested, take certain steps to handle it. Options include stopping the device, or silencing all or part of the hardware buffer used for playback.

 the stop threshold

*if the number of frames of data/space available meets or exceeds this value, the driver will stop the interface.*

the silence threshold

*if the number of frames of space available for a playback stream meets or exceeds this value, the driver will fill part of the playback hardware buffer with silence.*

silence size

*when the silence threshold level is met, this determines how many frames of silence are written into the playback hardware buffer*

 xrun指的是，声卡period一到，引发一个中断，告诉alsa驱动，要填入数据，或读走数据，但是，问题在于alsa的读取和写入操作必须用户调用writei和readi才会发生的，它不会去缓存数据。如果上层没有用户调用writei和readi，那么就会产生 overrun（录制时，数据都满了，还没被alsa驱动读走）和underrun（需要数据来播放，alsa驱动却不写入数据），统称为xrun。当xrun发生时，可以在空余空间超过stop threshold时，stop audio interface.

也可以通过设置silence threshold，当空余空间超过silence threshold时，就hardware buffer 写入silence.

Available minimum space/data for wakeup

Programs that use poll(2) or select(2) to determine when audio data may be transferred to/from the interface may set this to control at what point relative to the state of the hardware buffer, they wish to be woken up.

 这个software parameters仅用在interrupt-driven模式。这个模式是alsa驱动层的，不是硬件interrupt。它的意思是，用户使用 snd\_pcm\_wait()时，这个实际封装的是系统的poll调用，表示用户在等待，那么在等待什么呢？对于playback来讲，就是等待下面的声卡的hardware buffer里有一定数量的空间，可以放入新的数据了，对于record来讲，就是等待下面声卡新采集的数据达到了一定数量了。这个一定数量，就是用 snd\_pcm\_sw\_params\_set\_avail\_min来设置，单位是frame。

Transfer chunk size

this determines the number of frames used when transferring data to/from the device hardware buffer.

ALSA表示高级Linux声音体系结构(Advanced Linux Sound Architecture)。它由一系列内核驱动，应用程序编译接口(API)以及支持Linux下声音的实用程序组成。这篇文章里，我将简单介绍 ALSA项目的基本框架以及它的软件组成。主要集中介绍PCM接口编程，包括您可以自动实践的程序示例。

声音由变化的气压组成。它被麦克风这样的转换器转换成电子形式。模/数(ADC)转换器将模拟电压转换成离散的样本值。声音以固定的时间间隔被采样，采样的速率称为采样率。把样本输出到数/模(DAC)转换器，比如扩音器，最后转换成原来的模拟信号。

样本大小以位来表示。样本大小是影响声音被转换成数字信号的精确程度的因素之一。另一个主要的因素是采样率。奈奎斯特(Nyquist)理论中，只要离散系统的奈奎斯特频率高于采样信号的最高频率或带宽，就可以避免混叠现象。

ALSA由许多声卡的声卡驱动程序组成，同时它也提供一个称为libasound的API库。应用程序开发者应该使用libasound而不是内核中的 ALSA接口。因为libasound提供最高级并且编程方便的编程接口。并且提供一个设备逻辑命名功能，这样开发者甚至不需要知道类似设备文件这样的低层接口。相反，OSS/Free驱动是在内核系统调用级上编程，它要求开发者提供设备文件名并且利用ioctrl来实现相应的功能。

为了向后兼容，ALSA提供内核模块来模拟OSS，这样之前的许多在OSS基础上开发的应用程序不需要任何改动就可以在ALSA上运行。另外，libaoss库也可以模拟OSS，而它不需要内核模块。

ALSA包含插件功能，使用插件可以扩展新的声卡驱动，包括完全用软件实现的虚拟声卡。ALSA提供一系列基于命令行的工具集，比如混音器(mixer)，音频文件播放器(aplay)，以及控制特定声卡特定属性的工具。

### 20.1.1 ALSA测试声卡

ALSA官方网站

<https://www.alsa-project.org/wiki/SoundcardTesting>

播放音乐

In most sound cards, you'll need to unmute and adjust *Master* and *PCM* volumes. For testing the simple playback, use aplay program. Prepare a WAV file and simply run like:

% aplay -vv somefile.wav

With the option -vv, aplay shows the verbose(详细信息) information of the PCM device, and a VU-peak meter during playing the file. If you want to test the secondary card, pass -Ddefault:1 in addition. It means the default PCM on the secondary card.For testing a multi-channel PCM configuration, use speaker-test program. For testing 5.1 output, run like:

% speaker-test -Dplug:surround51 -c6 -twav

The option -Dplug:surround51 specifies that you're using 5.1-surround configuration with the automatic-conversion layer. Without this option, the default PCM would be chosen, and usually only two channels are used (and the rest channels are dropped silently). The -c6 specifies that you use 6 channels. The last -twav is to choose the WAV output mode, where speaker-test will tell you the speaker position in a nice soft voice.

## 20.2 [ALSA音频工具](http://www.cnblogs.com/cslunatic/p/3227655.html) mixer,aplay,arecord

从官网[http://www.alsa-project.org](http://www.alsa-project.org/)下载alsa-lib和alsa-utils

### **20.2.1声音录制命令ar**ecord

**执行arecord命令获取帮助信息**

arecord

Usage: arecord [OPTION]... [FILE]...

-h, --help help

--version print current version

-l, --list-devices list all soundcards and digital audio devices

-L, --list-pcms list device names

-D, --device=NAME select PCM by name

-q, --quiet quiet mode

-t, --file-type TYPE file type (voc, wav, raw or au)

-c, --channels=# channels

-f, --format=FORMAT sample format (case insensitive)

-r, --rate=# sample rate

-d, --duration=# interrupt after # seconds

-s, --samples=# interrupt after # samples per channel

-M, --mmap mmap stream

-N, --nonblock nonblocking mode

-F, --period-time=# distance between interrupts is # microseconds

-B, --buffer-time=# buffer duration is # microseconds

--period-size=# distance between interrupts is # frames

--buffer-size=# buffer duration is # frames

-A, --avail-min=# min available space for wakeup is # microseconds

-R, --start-delay=# delay for automatic PCM start is # microseconds

(relative to buffer size if <= 0)

-T, --stop-delay=# delay for automatic PCM stop is # microseconds from xrun

-v, --verbose show PCM structure and setup (accumulative)

-V, --vumeter=TYPE enable VU meter (TYPE: mono or stereo)

-I, --separate-channels one file for each channel

-i, --interactive allow interactive operation from stdin

-m, --chmap=ch1,ch2,.. Give the channel map to override or follow

--disable-resample disable automatic rate resample

--disable-channels disable automatic channel conversions

--disable-format disable automatic format conversions

--disable-softvol disable software volume control (softvol)

--test-position test ring buffer position

--test-coef=# test coefficient for ring buffer position (default 8)

expression for validation is: coef \* (buffer\_size / 2)

--test-nowait do not wait for ring buffer - eats whole CPU

--max-file-time=# start another output file when the old file has recorded

for this many seconds

--process-id-file write the process ID here

--use-strftime apply the strftime facility to the output file name

--dump-hw-params dump hw\_params of the device

--fatal-errors treat all errors as fatal

Recognized sample formats are: S8 U8 S16\_LE S16\_BE U16\_LE U16\_BE S24\_LE S24\_BE U24\_LE U24\_BE S32\_LE S32\_BE U32\_LE U32\_BE FLOAT\_LE FLOAT\_BE FLOAT64\_LE FLOAT64\_BE IEC958\_SUBFRAME\_LE IEC958\_SUBFRAME\_BE MU\_LAW A\_LAW IMA\_ADPCM MPEG GSM SPECIAL S24\_3LE S24\_3BE U24\_3LE U24\_3BE S20\_3LE S20\_3BE U20\_3LE U20\_3BE S18\_3LE S18\_3BE U18\_3LE U18\_3BE G723\_24 G723\_24\_1B G723\_40 G723\_40\_1B DSD\_U8 DSD\_U16\_LE DSD\_U32\_LE DSD\_U16\_BE DSD\_U32\_BE

Some of these may not be available on selected hardware

The available format shortcuts are:

-f cd (16 bit little endian, 44100, stereo)

-f cdr (16 bit big endian, 44100, stereo)

-f dat (16 bit little endian, 48000, stereo)

Arecord –L 列出所有设备名字

# arecord -L

null

Discard all samples (playback) or generate zero samples (capture)

sysdefault:CARD=rockchiprk3308v

rockchip,rk3308-vad,

Default Audio Device

sysdefault:CARD=Loopback

Loopback, Loopback PCM

Default Audio Device

arecord -l列出声卡和数字音频设备

# arecord -l

\*\*\*\* List of CAPTURE Hardware Devices \*\*\*\*

card 0: rockchiprk3308v [rockchip,rk3308-vad], device 0: dailink-multicodecs multicodec-0 []

Subdevices: 1/1

Subdevice #0: subdevice #0

card 7: Loopback [Loopback], device 0: Loopback PCM [Loopback PCM]

Subdevices: 8/8

Subdevice #0: subdevice #0

Subdevice #1: subdevice #1

Subdevice #2: subdevice #2

Subdevice #3: subdevice #3

Subdevice #4: subdevice #4

Subdevice #5: subdevice #5

Subdevice #6: subdevice #6

Subdevice #7: subdevice #7

card 7: Loopback [Loopback], device 1: Loopback PCM [Loopback PCM]

Subdevices: 7/8

Subdevice #0: subdevice #0

Subdevice #1: subdevice #1

Subdevice #2: subdevice #2

Subdevice #3: subdevice #3

Subdevice #4: subdevice #4

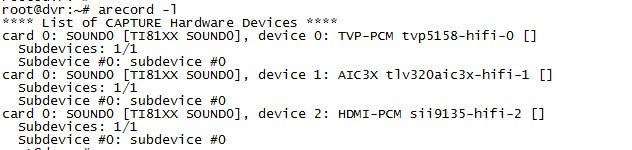
Subdevice #5: subdevice #5

Subdevice #6: subdevice #6

Subdevice #7: subdevice #7

录制命令学习

下面是网友开发板声卡列表：



-D参数用于指定音频设备PCM 以hwx,x开头 根据上面l列出的设备,如果选择tvp5158来录制声音的话那么pcm设备就位hw0,0,如果是tlv320aic3x则pcm设备为hw0,1,sii9135则pcm设备为hw0,2。 声卡号,设备号 -r指定采样频率:5512/8000/11025/16000/22050/32000/44100/48000/64000/88200/96000/176400/192000 -f指定采样格式上面列出了:cd/cdr/dat/S16\_LE/S32\_LE/...

arecord -D default:CARD=mcasp0 -d 100000   -f cd -t raw mysong15.raw

arecord -D default:CARD=mcasp0 -d 100000   -f dat -t raw mysong10.raw

arecord -D default:CARD=mcasp0 -d 100000   -f S16\_LE -r 16000 -t raw mysong10.raw

arecord –D hw:0,1 –r 8000 -f cd /a/1.wmv

arecord –D hw:0,0 -r8000 -f S16\_LE -c 2 /a/1.wmv  arecord -Dhw:0,2 -r48000 -f S32\_LE -c 2 /a/1.wmv

arecord -D 2mic\_loopback -c 4 -r 16000 -f S16\_LE /tmp/1.wav

alplay

# aplay /tmp/haiou.wav

## 20.3 tinyalsa

目前linux中主流的音频体系结构是ALSA（Advanced Linux Sound Architecture），ALSA在内核驱动层提供了alsa-driver，在应用层提供了alsa-lib，应用程序只需要调用alsa-lib提供的API就可以完成对底层硬件的操作。说的这么好，但是Android中没有使用标准的ALSA，而是一个ALSA的简化版叫做tinyalsa。Android中使用tinyalsa控制管理所有模式的音频通路，我们也可以使用tinyalsa提供的工具进行查看、调试。

编译tinyalsa后生成四个小工具：

1. tinymix
2. tinyplay
3. tinycap
4. tinypcminfo

**查看当前系统的声卡**

# cat /proc/asound/cards

0 [rockchiprk3308v]: rockchip\_rk3308 - rockchip,rk3308-vad

rockchip,rk3308-vad

7 [Loopback ]: Loopback - Loopback

Loopback 1

**tinymix**

# tinymix –help //帮助

usage: tinymix [options] <command>

options:

-h, --help : prints this help message and exists

-v, --version : prints this version of tinymix and exists

-D, --card NUMBER : specifies the card number of the mixer

commands:

get NAME|ID : prints the values of a control

set NAME|ID VALUE : sets the value of a control

controls : lists controls of the mixer

contents : lists controls of the mixer and their contents

# tinymix controls // lists controls of the mixer

Number of controls: 52

ctl type num name

0 INT 1 ALC AGC Group 0 Left Volume

1 INT 1 ALC AGC Group 0 Right Volume

2 INT 1 ALC AGC Group 1 Left Volume

3 INT 1 ALC AGC Group 1 Right Volume

4 INT 1 ALC AGC Group 2 Left Volume

5 INT 1 ALC AGC Group 2 Right Volume

6 INT 1 ALC AGC Group 3 Left Volume

7 INT 1 ALC AGC Group 3 Right Volume

8 INT 1 ALC AGC Group 0 Left Max Volume

9 INT 1 ALC AGC Group 0 Right Max Volume

10 INT 1 ALC AGC Group 1 Left Max Volume

11 INT 1 ALC AGC Group 1 Right Max Volume

12 INT 1 ALC AGC Group 2 Left Max Volume

13 INT 1 ALC AGC Group 2 Right Max Volume

14 INT 1 ALC AGC Group 3 Left Max Volume

15 INT 1 ALC AGC Group 3 Right Max Volume

16 INT 1 ALC AGC Group 0 Left Min Volume

17 INT 1 ALC AGC Group 0 Right Min Volume

18 INT 1 ALC AGC Group 1 Left Min Volume

19 INT 1 ALC AGC Group 1 Right Min Volume

20 INT 1 ALC AGC Group 2 Left Min Volume

21 INT 1 ALC AGC Group 2 Right Min Volume

22 INT 1 ALC AGC Group 3 Left Min Volume

23 INT 1 ALC AGC Group 3 Right Min Volume

24 INT 1 ADC MIC Group 0 Left Volume

25 INT 1 ADC MIC Group 0 Right Volume

26 INT 1 ADC MIC Group 1 Left Volume

27 INT 1 ADC MIC Group 1 Right Volume

28 INT 1 ADC MIC Group 2 Left Volume

29 INT 1 ADC MIC Group 2 Right Volume

30 INT 1 ADC MIC Group 3 Left Volume

31 INT 1 ADC MIC Group 3 Right Volume

32 INT 1 ADC ALC Group 0 Left Volume

33 INT 1 ADC ALC Group 0 Right Volume

34 INT 1 ADC ALC Group 1 Left Volume

35 INT 1 ADC ALC Group 1 Right Volume

36 INT 1 ADC ALC Group 2 Left Volume

37 INT 1 ADC ALC Group 2 Right Volume

38 INT 1 ADC ALC Group 3 Left Volume

39 INT 1 ADC ALC Group 3 Right Volume

40 ENUM 1 ADC Group 0 HPF Cut-off

41 ENUM 1 ADC Group 1 HPF Cut-off

42 ENUM 1 ADC Group 2 HPF Cut-off

43 ENUM 1 ADC Group 3 HPF Cut-off

44 INT 1 DAC LINEOUT Left Volume

45 INT 1 DAC LINEOUT Right Volume

46 INT 1 DAC HPOUT Left Volume

47 INT 1 DAC HPOUT Right Volume

48 INT 1 DAC HPMIX Left Volume

49 INT 1 DAC HPMIX Right Volume

50 BOOL 1 vad switch

51 INT 2 Master Playback Volume

**使用tinyplay播放wav音乐**

*tinyplay /sdcard/0\_16.wav*

5.tinycap使用

root@android:/ # tinycap /sdcard/test.wav

可以进行录音。

## 20.4 alsa-lib应用层接口分析

ALSA lib接口调用简介

ALSA逻辑，在我当前看来，总共有两条线：1、录放音流控，2、amixer cset控件。

录放音流控（自定义名称），相当于操作OSS的/dev/dsp设备，可以设置三大参数等，并且启动录放音。这里，aplay，mplayer等播放器，调用ALSA lib中的snd\_pcm\_\*\*\*等，标准接口函数，再通过该函数，操作/dev/snd/pcmC0D0c pcmC0D0p pcmC0D1c pcmC0D1p pcmC0D2c等流控设备节点，完成设置音频三大参数，DMA buffer使用大小，预缓冲buffer阀值等，并且启动录放音。因为ALSA功能的丰富，这里录放音，并不是使用write,read这些接口函数，而应该是通过ioctl接口，来完成的。

amixer cset控件，相当于OSS驱动中的/dev/mixer设备节点的操作，可以调节录放音的音量，静音，录放音通道切换等。amixer cset通过调用ALSA lib中的标准函数，操作/dev/snd/controlC0设备节点，来操作底层codec\_\*\*\*.c驱动中，定义的kcontrol控件。并通过kcontrol控件，直接读写codec中的相关寄存器。这是ALSA与OSS较大不同的一点，将codec相关功能寄存器，零件化，提供给上层，由上层自由设置。这里，也是通过 controlC0设备节点的ioctl接口，来读写codec的kcontrol。

**一、ALSA流控接口操作方法，可参考alsa utils源码中的aplay.c文件：**

1、snd\_pcm\_open(&handle, pcm\_name, stream, open\_mode);

&handle，是static snd\_pcm\_t \*handle;该结构体指针，在后面的所有操作中，都要用到。

pcm\_name，是打开的pcm设备节点，可以是“default” ,也可以是“hw:0,0”。具体有何区别，可网上查询。stream，对应放音与录音操作，SND\_PCM\_STREAM\_PLAYBACK，SND\_PCM\_STREAM\_CAPTURE。open\_mode，是一些相关功能设置，比如：是否阻塞，是否重采样，是否软件调音量等。

2、snd\_pcm\_info(handle, info)；

一般用来获取录放音声卡与设备等相关信息。不是必须的接口。

3、snd\_pcm\_nonblock(handle, 1);

如果需要非阻塞方式，则将第二个参数，设置为1。snd\_pcm\_open后，默认为阻塞方式，因此，该接口可以不用调用。

4、snd\_pcm\_hw\_params\_alloca(&params);

申请一段snd\_pcm\_hw\_params\_t \*params;结构体空间。使用该结构体，来配置底层ALSA的相关硬件参数，比如：声道，采样率，位宽，buffer大小等。

stream == SND\_PCM\_STREAM\_PLAYBACK

playback(argv[optind++])--->playback\_go(fd, 0, pbrec\_count, FORMAT\_AU, name)--->set\_params()-->snd\_pcm\_hw\_params\_alloca

5、snd\_pcm\_hw\_params\_any(handle, params);

6 snd\_pcm\_hw\_params\_set\_access(handle, params,SND\_PCM\_ACCESS\_RW\_INTERLEAVED);设置为交错方式。

snd\_pcm\_hw\_params\_set\_rate\_resample(handle, params, soft\_resample);

7、snd\_pcm\_hw\_params\_set\_format(handle, params, hwparams.format);设置采样位宽格式。

8、snd\_pcm\_hw\_params\_set\_channels(handle, params, hwparams.channels);设置声道数。

9、snd\_pcm\_hw\_params\_set\_rate\_near(handle, params, &hwparams.rate, 0);设置采样率，之所以是rate\_near，是因为上层设置的采样率&hwparams.rate，底层很可能不支持，因此，会返回上层一个底层支持的相近的采样率。

10、snd\_pcm\_hw\_params\_get\_buffer\_time\_max(params, &buffer\_time, 0);获取底层支持的最大buffer大小，换算为时间。

snd\_pcm\_hw\_params\_set\_period\_time\_near(handle, params, &period\_time, 0); 设置底层period区间大小，以时间为单位，之所以为time\_near，是因为上层设置的大小很可能底层不支持，底层会返回相近的&period\_time。

snd\_pcm\_hw\_params\_set\_period\_size\_near(handle, params, &period\_frames, 0);设置底层period区间大小，以字节为单位，size\_near，原因类似。该接口与set\_period\_time\_near接口，都是设置period大小，只是单位不同，二选一即可。

snd\_pcm\_hw\_params\_set\_buffer\_time\_near(handle, params, &buffer\_time, 0);设置底层buffer区间大小，以时间为单位。与上面接口类似。在底层，buffer是一个大DMA空间，而period是将buffer平均分成的几个小区间。

snd\_pcm\_hw\_params\_set\_buffer\_size\_near(handle, params, &buffer\_frames);与上类似。

上面设置完成后，都需要调用下面两个函数，获取最终设置的period与buffer大小。

snd\_pcm\_hw\_params\_get\_period\_size(params, &chunk\_size, 0);

snd\_pcm\_hw\_params\_get\_buffer\_size(params, &buffer\_size);

11、snd\_pcm\_hw\_params\_is\_monotonic(params);？？？

snd\_pcm\_hw\_params\_can\_pause(params);查询底层是否支持暂停。

12、snd\_pcm\_hw\_params(handle, params);将上面设置的硬件配置参数，加载，并且会自动调用snd\_pcm\_prepare（ ）将stream状态置为SND\_PCM\_STATE\_PREPARED。

13、snd\_pcm\_sw\_params\_alloca(&swparams);申请一段snd\_pcm\_sw\_params\_t \*swparams;结构体空间，存放软件配置参数。

14、snd\_pcm\_sw\_params\_current(handle, swparams);获取当前的软件参数配置。

15、snd\_pcm\_sw\_params\_set\_avail\_min(handle, swparams, n);？？？

16、snd\_pcm\_sw\_params\_set\_start\_threshold(handle, swparams, start\_threshold);设置开始的阀值，可以起到预缓冲的效果。对于放音，当上层调用snd\_pcm\_writei超过阀值时，才会启动播放。

17、snd\_pcm\_sw\_params\_set\_stop\_threshold(handle, swparams, stop\_threshold);设置停止的阀值，避免硬件上的underrun或overrun出现。

18、snd\_pcm\_sw\_params(handle, swparams)；将上面的软件参数，加载到底层。

19、上面设置完成后，就可以开始read与write了。

20、snd\_pcm\_writei(snd\_pcm\_t \*pcm, const void \*buffer, snd\_pcm\_uframes\_t size)；播放交错格式的音频数据。snd\_pcm\_t \*pcm为snd\_pcm\_open时得到的。size，这里不是字节数，不是采样点数。而是采样帧数。双声道，一帧采样，有两个采样点。ALSA上层接口，一般都是以采样帧数为单位的。

stream == SND\_PCM\_STREAM\_PLAYBACK 播放

playback(argv[optind++])--->playback\_go(fd, 0, pbrec\_count, FORMAT\_AU, name)--->set\_params()设置完参数播放歌曲-->pcm\_write()--->writei\_func(handle, data, count)--->snd\_pcm\_writei--> \_snd\_pcm\_writei()-->snd\_pcm\_hw\_writei(alsa-lib-1.1.3/src/pcm/pcm\_hw.c)-->ioctl(fd, SNDRV\_PCM\_IOCTL\_WRITEI\_FRAMES, &xferi)调进内核

内核中，根据/soc/pcm\_native.c文件中.unlocked\_ioctl = snd\_pcm\_capture\_ioctl,可知snd\_pcm\_capture\_ioctl1被调用，根据SNDRV\_PCM\_IOCTL\_READI\_FRAMES参数可知snd\_pcm\_lib\_write(substream, xferi.buf, xferi.frames);被调用，最终snd\_pcm\_lib\_write1(,,,,snd\_pcm\_lib\_write\_transfer)被调用。根据transfer被调用可知snd\_pcm\_lib\_write\_transfer被调用，然后调用copy\_from\_user(hwbuf, buf, frames\_to\_bytes(runtime, frames)，可知，将alsa\_lib提供的一个指针所指的内存拷贝到dma端内存的数据，alsa库函数snd\_pcm\_readi、snd\_pcm\_writei实现了内存到内存的交互，或者近似地认为是内存到音频文件的交互。

snd\_pcm\_readi(snd\_pcm\_t \*pcm, void \*buffer, snd\_pcm\_uframes\_t size)；录制交错格式的音频数据。参数与上相同。

snd\_pcm\_writen与snd\_pcm\_readn，与上面两个函数，基本相同，只是录放音数据的格式是非交错的。既一帧数据，只包含一个声道数据，交错格式指，一帧数据中，包含了多个声道的数据。比如：双声道一帧数据中，包含了左声道与右声道，各一个采样点。

21、snd\_pcm\_format\_set\_silence(snd\_pcm\_format\_t format, void \*data, unsigned int samples)；

用于将某个buffer中的数据，转换为静音，也就是0。

22、snd\_pcm\_nonblock(handle, 0);打开或关闭阻塞模式。0，为阻塞模式，1，为非阻塞模式。

22、snd\_pcm\_drain(handle);等待buffer中的数据全部播放完成。

snd\_pcm\_drop(handle);不等待buffer数据播放完成，马上停止。

23、snd\_pcm\_close(handle)；关闭pcm handle。

24、ALSA lib中的dmix插件，只能用于放音通道数据的混音，不能用于录音通道数据的混音。

25、.asoundrc其实是指向asound.conf文件的软链接文件。

asound.conf，是针对pcm与ctl设备的功能插件调用流程表，需要根据具体的功能需求，来配置该文件。否则，会没有相关设备，或无法完成相关功能。

26、alsa lib中，plug插件，可完成重采样，声道转换等操作。asym插件，完成分别声明：播放与录音 下一级slave插件的功能。dsnoop，完成多来源录音 混音的功能，dmix，完成多来源放音，混音的功能。mutil，完成多声道数据，虚拟对应到多codec声道的功能。

二、ALSA amixer cset控件接口操作，可参考alsa utils源码中的amixer.c：

上层amixer程序，调用ALSA lib中的标准接口函数，操作/dev/controlC0设备（主要使用了ioctl接口），来完成对底层codec\_\*\*\*.c中kcontrol的一些操作。在内核kernel/sound/core/control.c中，定义了struct file\_operations snd\_ctl\_f\_ops，该operation会注册/dev/controlC\*设备的file operations。

比如：amixer cset设置某个element值时，最终会调到kernel/sound/core/control.c中，snd\_ctl\_ioctl（）函数。SNDRV\_CTL\_IOCTL\_ELEM\_READ中，来完成读取该element control的值。SNDRV\_CTL\_IOCTL\_ELEM\_WRITE中，则完成将上层设置的element control的值，写入kcontrol中。

这两个case中，分别调用snd\_ctl\_elem\_write（）和snd\_ctl\_elem\_read（）函数，最后调用到codec\_\*\*\*.c驱动中，snd\_kcontrol\_new结构体中，SOC\_SINGLE与SOC\_ENUM等宏中，.info，.get，.put，对应的读写codec寄存器的函数。

alsa\_amixer-用户态

|->snd\_ctl\_ioctl-系统调用

|->snd\_ctl\_elem\_write\_user-内核钩子函数

|->snd\_ctl\_elem\_wirte-

|->snd\_ctl\_find\_id-遍历 kcontrol 链表找到 name 字段匹配的 kctl

|->kctl->put()-调用 kctl 的成员函数 put()

|->snd\_soc\_put\_volsw

最近发现，ALSA codec\_\*\*\*.c驱动中，调用SOC\_SINGLE，SOC\_DOUBLE\_R\_TLV，SOC\_ENUM等宏，注册snd\_kcontrol\_new控件后，alsa上层调用alsa lib中的alsa-lib-v1.0.28/src/mixer/simple.c中的selems()-->show\_selem()-->snd\_mixer\_find\_selem（）函数，查找匹配的amixer cset控件时，名称与底层的名称不一致。

比如：

底层codec\*\*\*.c定义了SOC\_SINGLE("Aux In Switch", DIGITAL\_FILTER\_MODE, 1, 1, 0),

上层snd\_mixer\_find\_selem（）查找到的s->id->name = Aux In，将Switch省略了。

还有SOC\_DOUBLE\_R\_TLV("Master Playback Volume", LCH\_DIGITAL\_VOLUME, RCH\_DIGITAL\_VOLUME, 0, 0xcc, 1, dac\_tlv),上层获取s->id->name = Master。将Playback Volume省略了。

但个别的控件名称，没有省略，比如：SOC\_SINGLE("Digital Playback mute", MODE\_CONTROL\_3, 5, 1, 0),上层获取的s->id->name = Digital Playback mute。

经过追查在alsa-lib-v1.0.28/src/control/hcontrol.c中，有对控件名称的简化操作，将一些Switch,Volume,Playback之类的通用单词，省略掉了，只保留控件开头的单词。具体为何这样做，还没去弄清楚。

所以，上层在调用alsa-lib-v1.0.28/src/mixer/simple.c中的snd\_mixer\_find\_selem（）函数，查找匹配的amixer cset控件时，最好将s->id->name 打印出来，再设置，否则会查找不到对应名称的控件。

上层查找控件的示例代码如下：

static void alsa\_linein\_init(char \*alsa\_mix\_ctrl, snd\_mixer\_t \*\*alsa\_mix\_handle\_p, snd\_mixer\_elem\_t \*\*alsa\_mix\_elem\_p)

{

/\* 查找控件函数 \*/

int alsa\_mix\_index = 0;

char alsa\_mix\_dev[32] = {0};

snd\_mixer\_selem\_id\_t \*alsa\_mix\_sid = NULL;

snd\_mixer\_selem\_id\_alloca(&alsa\_mix\_sid);

snd\_mixer\_selem\_id\_set\_index(alsa\_mix\_sid, alsa\_mix\_index);

snd\_mixer\_selem\_id\_set\_name(alsa\_mix\_sid, alsa\_mix\_ctrl);

if(mozart\_ini\_getkey("/usr/data/system.ini", "audio", "playback", alsa\_mix\_dev))

strcpy(alsa\_mix\_dev,"default");

if ((snd\_mixer\_open(alsa\_mix\_handle\_p, 0)) < 0)

DEBUG ("Failed to open mixer");

if ((snd\_mixer\_attach(\*alsa\_mix\_handle\_p, alsa\_mix\_dev)) < 0)

DEBUG ("Failed to attach mixer");

if ((snd\_mixer\_selem\_register(\*alsa\_mix\_handle\_p, NULL, NULL)) < 0)

DEBUG ("Failed to register mixer element");

if (snd\_mixer\_load(\*alsa\_mix\_handle\_p) < 0)

DEBUG ("Failed to load mixer element");

\*alsa\_mix\_elem\_p = snd\_mixer\_find\_selem(\*alsa\_mix\_handle\_p, alsa\_mix\_sid);

if (!\*alsa\_mix\_elem\_p)

DEBUG ("Failed to find mixer element");

return;

}

{

snd\_mixer\_t \*alsa\_mix\_linein\_handle = NULL;

snd\_mixer\_elem\_t \*alsa\_mix\_linein\_elem = NULL;

char alsa\_linein\_ctrl[32];

if(mozart\_ini\_getkey("/usr/data/system.ini", "audio", "linein", alsa\_linein\_ctrl))

strcpy(alsa\_linein\_ctrl,"Aux In Switch"); //查找预设控件名称，该代码为私有代码，非标准。

alsa\_linein\_init(alsa\_linein\_ctrl, &alsa\_mix\_linein\_handle, &alsa\_mix\_linein\_elem); //查找控件elem

snd\_mixer\_selem\_set\_playback\_switch(alsa\_mix\_linein\_elem, 0, 1); //设置控件elem

if(alsa\_mix\_linein\_handle){

snd\_mixer\_close(alsa\_mix\_linein\_handle); //关闭

}

}

三、相关细节解释：

1、ALSA接口中，有交错，非交错的概念。分别对应上层的snd\_pcm\_writei与snd\_pcm\_writen。

交错指的是：同一时间采的一帧数据中，包含左右两个声道或多声道的采样点。非交错指的是：同一时间采的一帧数据中，只包含一个声道。我们目前遇到的，都是交错帧。非交错帧，相当于录音端，同一时间只录一个声道的采样点，这样的情况，应该是在一些特殊场合出现的。

上层除了调用的snd\_pcm\_write，snd\_pcm\_read函数有区别外，在snd\_pcm\_hw\_params（ ）设置硬件参数时，也需要调用snd\_pcm\_hw\_params\_set\_access(handle, params, SND\_PCM\_ACCESS\_RW\_INTERLEAVED)，通知底层为交错方式。

INTERLEAVED代表是交错方式，我们目前都用交错方式。

# linux iio子系统

最近由于工作的需要，接触了Linux iio子系统，对于这个目录其实以前是很少接触，接下了对 Linux iio 子系统进行分析。

**1、**首先 iio子系统在内核树中位置：drivers/staging/iio

详细的iio子系统说明文档位置：drivers/staging/iio/Documentation（文档是个好东西，详细阅读文档，有利于更深层次的理解iio子系统）

**2、**简介：  
iiO子系统全称是 Industrial I/O subsystem（工业 I/O 子系统），此子系统的目的在于填补那些分类时处在hwmon（硬件监视器）和输入子系统之间的设备类型。在某些情况下，iio和hwmon、Input之间的相当大的重叠。

**3、**iio目录结构架构：

iio

├── accel

├── adc

├── addac

├── cdc

├── dac

├── dds

├── Documentation

│   └── dac

├── gyro

├── impedance-analyzer

├── imu

│   └── mpu

│   └── inv\_test

├── light

├── magnetometer

├── meter

├── pressure

├── resolver

└── trigger

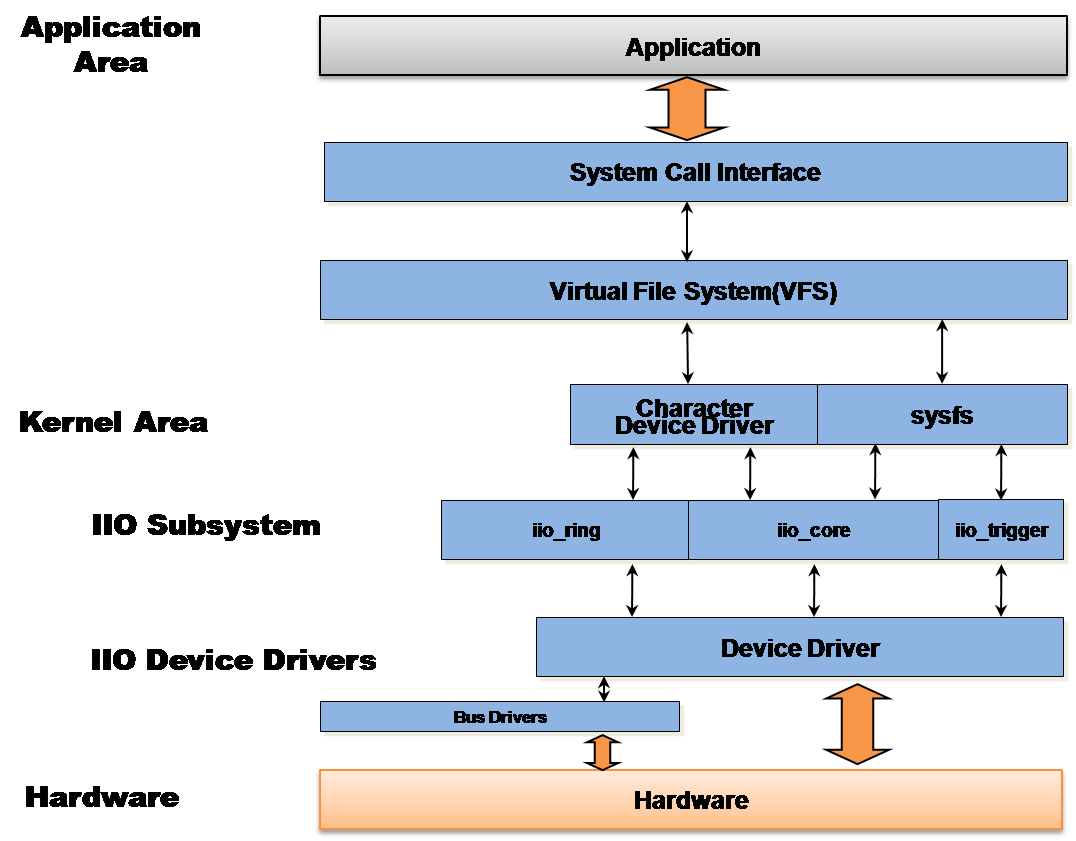
**4、**iio 目录结构说明：

进入相应目录，里面都有相关IC的驱动，网上查阅资料，对iio子系统做如下简要的说明：  
accel ：  
该文件夹下是一些加速度传感器，例如：adis16201、kxsd9、lis3l02dq、sca3000等  
adc ：  
该文件夹下是一些模数转换器，将模拟信号转换成数字信号，例如：ad7192、adt7310（数字温度传感器）等  
addac ：  
Temperature Sensor 温度传感器，例如：adt7316  
cdc ：  
电容数字转换，例如：ad7150  
dac ：

一些数模转换器，将数字信号转换成模拟信号，例如：ad5064、ad5791（单通道、20位、无缓冲电压输出DAC）等  
dds ：  
频率扫描仪，频率合成器，例如：ad5930、ad9951等  
Documentation ：  
iio子系统相关文档说明，相关说明比较详细，说明文档是个好东西……  
gyro：  
陀螺仪，例如：adis16060（角速度陀螺仪）、adis16260（数字陀螺仪）、adxrs450（角速率陀螺仪）等  
impedance-analyzer ：  
阻抗测量芯片，只有一个芯片ad5933  
imu：  
惯性陀螺仪、磁力计、加速度计，例如：adis16400，其中的mpu子目录有些重要的传感器（目前工作中正在使用）  
imu  
    └── mpu：有MPU3050（三轴）、MPU6050（六轴）、MPU9150（整合了MPU6050及AK8975电子罗盘）、MPU6515等  
light ：  
光学传感器，例如：isl29018、tsl2563等  
magnetometer ：  
地磁传感器、磁力计传感器，例如：hmc5843、ak8975  
meter ：   
有功功率和电能计量，例如：ade7759（电能计量数据转换器）、ade7753等  
pressure ：  
压力传感器，例如：bmp182  
resolver ：  
旋转变压器/数字转换器，例如：ad2s1200（旋转变压器输出的模拟信号转化为数字信号）等  
trigger：

触发器

**5、**iio子系统架构图：



**6、**iio子系统功能：

（1）、设备注册和处理

（2）、通过虚拟文件系统（VFS）轮训访问设备

（3）、chrdevs事件

chrdevs事件包括阈值检测器，自由下落检测器和更复杂的动作检测。chrdevs事件的输入，iio为底层硬件触发与用户空间通行提供了通道，chrdevs事件本身已经不仅仅是一个事件的代码和一个时间戳，与chrdevs事件相关联的任何数据必须通过轮询访问。

（4）、硬件环缓冲支持

现在很多传感器芯片上本身就包括 fifo / ring 缓冲，通过sensor自带 fifo / ring 缓冲，可以大大的减少主芯片处理器的负担。

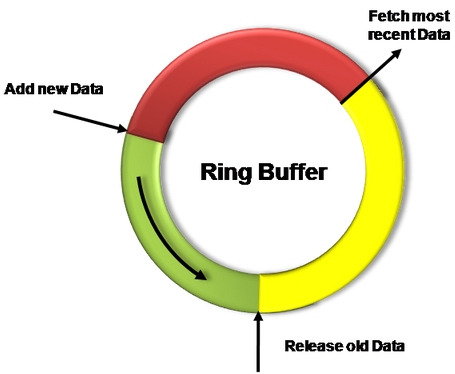
（6）、触发和软件缓冲区（kfifo）支持

在许多数据分析中，触发和软件缓冲区（kfifo）支持就显得非常有用，能够更加高效的捕捉到外部信号数据。这些触发包括（数据准备信号、GPIO线连接到外部系统、处理器周期中断、用户空间访问sysfs中的特定文件等），都会产生触发。

**7、**iio子系统的内核接口

为各种sensor提供了内核接口

**8、**iio子系统环形缓冲区（如下图）



iio子系统采用环形缓冲区，环形缓冲区本质是一个数据结构（单一，固定大小，可调并首尾相连），这种结构非常适合缓冲数据流。这些缓冲区通常用来解决生产者消费者问题，在一些应用中，它被设计成生产者会(例如一个ADC)覆盖消费者(例如一个用户空间应用程序)无法暂时处理的过期数据。但是通常这种缓冲会被设置为适当的大小，以使这种情况不会发生。

# 错误解决

Sh:vbetool not found 发现和input-event-daemon有关

/home/chao/rk3308\_linux/buildroot/output/rockchip\_rk3308\_32\_debug/target/etc/ input-event-daemon.conf是编译时临时从下面文件拷贝到当前文件的

/home/chao/rk3308\_linux/buildroot/board/rockchip/rk3308/fs-overlay/etc/input-event-daemon.conf

因此需要更改

/home/chao/rk3308\_linux/buildroot/board/rockchip/rk3308/fs-overlay/etc/input-event-daemon.conf

注释掉空闲事件

[Idle]

#1h 30m = vbetool dpms off

#reset = vbetool dpms on

发现触摸屏好像有误触发，未按下会有坐标

猜想应该是硬件上中断线没有上拉，加了上拉电阻，依然会出现。只是时间间隔变长了。