

Android 按键添加和处理的方案

需求：Android 机器上有个 Wifi 物理按键，现在需求通过点击“wifi 物理按键”能够快速的开启/关闭 wifi。

实现方案

经过思考之后，拟出下面几种方案：

方案一，在 linux kernel 的驱动中捕获“wifi 物理按键”。在 kernel 的按键驱动中截获“wifi”按键，并对其进行处理：若是“wifi”是开启的，则关闭 wifi；否则，打开 wifi。

方案二，在 Android 中添加一个服务，监听 wifi 按键消息。若监听到“wifi”按键，则读取 wifi 的状态：若是“wifi”是开启的，则关闭 wifi；否则，打开 wifi。

方案三，在 Android 的 input 输入子系统的框架层中捕获 wifi 按键，并进行相应处理。若捕获到“wifi”按键，则读取 wifi 的状态：若是“wifi”是开启的，则关闭 wifi；否则，打开 wifi。

方案一

方案思路：在 linux kernel 的驱动中捕获“wifi 物理按键”。在 kernel 的按键驱动中截获“wifi”按键，并对其进行处理：若是“wifi”是开启的，则关闭 wifi；否则，打开 wifi。

方案分析：若采用此方案需要解决以下问题

01，在 kernel 的按键驱动中捕获“wifi”按键。

— 这个问题很好实现。在 kernel 的按键驱动中，对按键值进行判断，若是 wifi 按键，则进行相应处理。

02，在 kernel 中读取并设置 wifi 的开/关状态。

— 这个较难实现。因为 wifi 驱动的开/关相关的 API 很难获取到。一般来说，wifi 模组的驱动都是 wifi 厂家写好并以 .ko 文件加载的。若需要获取 wifi 的操作 API，需要更厂家一起合作；让它们将接口开放，并让其它设备在 kernel 中可以读取到。

03，在 kernel 中将 wifi 的状态上报到 Android 系统中。若单单只是实现 02 步，只是简单的能开/关 wifi 了；但还需要想办法让 Android 系统直到 wifi 的开/关行为。

— 可以实现，但是太麻烦了。

方案结论：实现难度太大！

方案二

方案思路：在 Android 中添加一个服务，监听 wifi 按键消息。若监听到“wifi”按键，则读取 wifi 的状态：若是“wifi”是开启的，则关闭 wifi；否则，打开 wifi。

方案分析：若采用此方案需要解决以下问题

01，将 kernel 的 wifi 按键上传到 Android 系统中。

— 这个可以实现。首先，我们将 wifi 按键映射到一个 sys 文件节点上：按下 wifi 按键时，sys 文件节点的值为 1；未按下 wifi 按键时，sys 文件节点的值为 0。其次，通过 NDK 编程，读取该 sys 文件节点，并将读取的接口映射注册到 JNI 中。最后，通过 JNI，将该接口对应注册到 Android 系统中，使应用程序能够读取该接口。

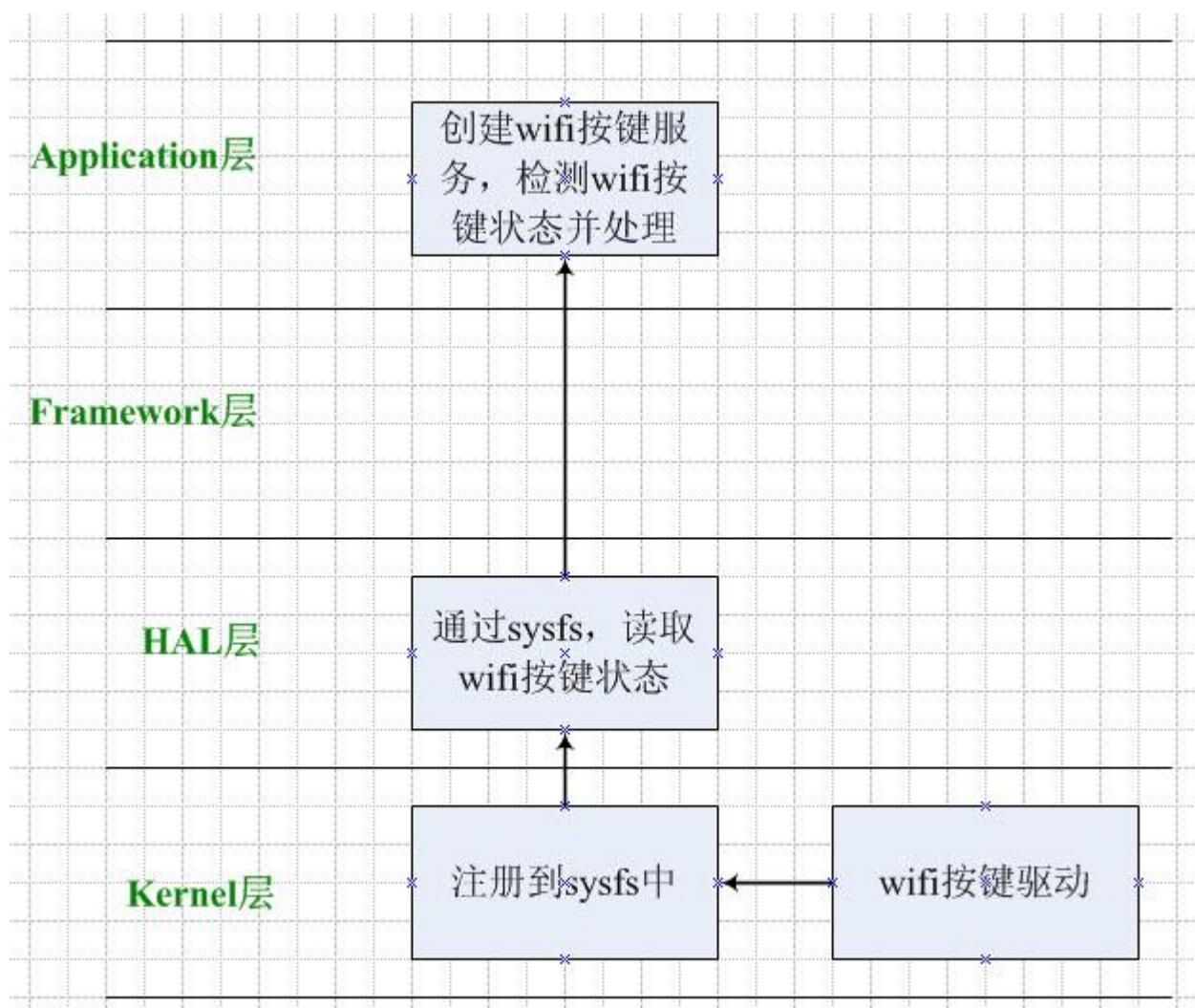
02，在 Android 系统中添加一个服务，不断读取 wifi 按键状态。

— 这个也可以实现。由于“01”中，我们已经将 wifi 的按键状态通过 JNI 注册到 Android 系统中；我们这里就可以读取到。

03，读取并设置 wifi 的开/关状态。

— 这个也可以实现。在 Android 系统中，我们可以通过 WifiManager 去读取/设置 wifi 的开/关状态。通过 WifiManager 设置的 wifi 状态，是全局的。

架构图：



具体实现:

通过驱动，将 wifi 按键状态映射到文件节点。由于不同平台差异，具体的代码接口可能有所差异；我所工作的平台是 RK3066，所以还是以此来进行介绍。

01 将 kernel 的 wifi 按键上传到 Android 系统中

在按键驱动中编辑 wifi 按键的驱动：主要的目的是将 wifi 按键映射到某个键值上，方便后面 Android 系统调用。因为 Android 系统使用的按键值和 Linux 内核使用的按键值不一样，Android 会通过一个键值映射表，将 Linux 的按键值和 Android 的按键值映射起来。

我们的项目中，wifi 按键是通过 ADC 值来捕获的，而不是中断。下面是“wifi 按键相关信息”，代码如下：



```
static struct rk29_keys_button key_button[] = {  
  
    ...  
    // 将 wifi 开关按键定义为 KEY_F16，  
    // 处理时，捕获 KEY_F16 进行处理即可。  
    {  
        .desc    = "wifi",  
        .code     = KEY_F16,  
        .adc_value = 4,  
        .gpio     = INVALID_GPIO,  
        .active_low = PRESS_LEV_LOW,  
    },  
    ...  
};
```



从中，我们可以看出 wifi 的 adc 值大概是 4，它所对应的按键值(即 code 值)是 KEY_F16。

这里，KEY_F16 是我们自己定义的(因为 linux 中没有 wifi 开关按键)，你也可以定义为别的值。记得两点：一，这里的所定义的 wifi 的 code，必须和 Android 中要处理的按键值(后面会讲到)保持一致；二，不要使用系统中已用到的值。另外，KEY_F16 的值为 186，可以参考“include/linux/input.h”文件去查看。

在按键驱动中，会将 key_button 注册到系统中。在按键驱动中，我们将下面的 callback 函数注册到 adc 总线上；adc 驱动会通过工作队列，判断的读取 adc 值，并调用 callback，从而判断是否有响应的按键按下。下面是 callback 函数：



```
static void callback(struct adc_client *client, void *client_param, int result)  
{  
    struct rk29_keys_drvdata *ddata = (struct rk29_keys_drvdata *)client_param;  
    int i;  
  
    if(result < EMPTY_ADVALUE)  
        ddata->result = result;  
  
    // 依次查找 key_button 中的按键，判断是否需要响应  
    for (i = 0; i < ddata->nbuttons; i++) {
```

```

struct rk29_button_data *bdata = &ddata->data[i];
struct rk29_keys_button *button = bdata->button;
if(!button->adc_value)
    continue;
int pre_state = button->adc_state;
if(result < button->adc_value + DRIFT_ADVALUE &&
    result > button->adc_value - DRIFT_ADVALUE) {

    button->adc_state = 1;
} else {
    button->adc_state = 0;
}
// 同步按键状态
synKeyDone(button->code, pre_state, button->adc_state);

if(bdata->state != button->adc_state)
    mod_timer(&bdata->timer,
        jiffies + msecs_to_jiffies(DEFAULT_DEBOUNCE_INTERVAL));
}
return;
}

```



前面已经说过，这个 callback 会不断的被 adc 检测的工作队列调用。若检测到 adc 值在“某按键定义的 adc 值范围”内，则该按键被按下；否则，没有按下。

下面是 synKeyDone() 的代码：



```

static void synKeyDone(int keycode, int pre_status, int cur_status)
{
    if (cur_status == pre_status)
        return ;

    if (keycode==KEY_F16)
        set_wifikey(cur_status);
}

```



它的作用是同步 wifi 按键按下状态，根据 wifi 按键状态，通过 set_wifikey() 改变对应 wifi 节点状态。

例如：wifi 键按下时，sys/devices/platform/misc_ctl/wifikey_onoff 为 1；wifi 未按下时，sys/devices/platform/misc_ctl/wifikey_onoff 为 0。

set_wifikey() 本身以及它相关的函数如下：



```

// 保存按键状态的结构体
typedef struct combo_module__t {

```

```

    unsigned char          status_wifikey;
}    combo_module_t    ;

static    combo_module_t    combo_module;

// 设置 wifi 状态。
// 这是对外提供的接口
void set_wifikey(int on)
{
    printk("%s on=%d\n", __func__, on);
    combo_module.status_wifikey = on;
}
EXPORT_SYMBOL(set_wifikey);

// 应用层读取 wifi 节点的回调函数
static    ssize_t show_wifikey_onoff    (struct device *dev, struct device_attribute *attr,
char *buf)
{
    return    sprintf(buf, "%d\n", combo_module.status_wifikey);
}

// 应用层设置 wifi 节点的回调函数
static    ssize_t set_wifikey_onoff    (struct device *dev, struct device_attribute *attr,
const char *buf, size_t count)
{
    unsigned int    val;
    if(!(sscanf(buf, "%d\n", &val))) {
        printk("%s error\n", __func__);
        return    -EINVAL;
    }

    if(!val) {
        combo_module.status_wifikey = 0;
    } else {
        combo_module.status_wifikey = 1;
    }
    printk("%s status_wifikey=%d\n", __func__, combo_module.status_wifikey);

    return 0;
}

// 将 wifi 的读取/设置函数和节点对应
static    ssize_t show_wifikey_onoff    (struct device *dev, struct device_attribute *attr,
char *buf);
static    ssize_t set_wifikey_onoff    (struct device *dev, struct device_attribute *attr,
const char *buf, size_t count);
static    DEVICE_ATTR(wifikey_onoff, S_IRWXUGO, show_wifikey_onoff, set_wifikey_onoff);

```



代码说明:

(01) `set_wifikey()` 提供的对外接口。用于在按键驱动中, 当 `wifi` 按键按下/松开时调用; 这样, 就对应的改变 `wifi` 节点的值。

(02) `DEVICE_ATTR(wifikey_onoff, S_IRWXUGO, show_wifikey_onoff, set_wifikey_onoff)`; 声明 `wifi` 的节点为 `wifikey_onoff` 节点, 并且设置节点的权限为 `S_IRWXUGO`, 设置“应用程序读取节点时的回调函数”为 `show_wifikey_onoff()`, 设置“应用程序设置节点时的回调函数”为 `set_wifikey_onoff()`,

`DEVICE_ATTR` 只是声明了 `wifi` 节点, 具体的注册要先将 `wifikey_onoff` 注册到 `attribute_group` 中; 并且将 `attribute_group` 注册到 `sysfs` 中才能在系统中看到文件节点。下面是实现代码:



// 将 `wifikey_onoff` 注册到 `attribute` 中

```
static struct attribute *control_sysfs_entries[] = {
    &dev_attr_wifikey_onoff.attr,
    NULL
};
```

```
static struct attribute_group control_sysfs_attr_group = {
    .name    = NULL,
    .attrs   = control_sysfs_entries,
};
```

// 对应的 `probe` 函数。主要作用是将 `attribute_group` 注册到 `sysfs` 中

```
static int control_sysfs_probe(struct platform_device *pdev)
{
    return sysfs_create_group(&pdev->dev.kobj, &control_sysfs_attr_group);
}
```

// 对应的 `remove` 函数。主要作用是将 `attribute_group` 从 `sysfs` 中注销

```
static int control_sysfs_remove(struct platform_device *pdev)
{
    sysfs_remove_group(&pdev->dev.kobj, &control_sysfs_attr_group);

    return 0;
}
```



02 将 Wifi 读取接口注册到 Android 系统中

通过 JNI, 将 `wifi` 读取接口注册到 Android 系统中, 下面是对应的 JNI 函数 `control_service.c` 的代码:



```
#include <stdlib.h>
#include <string.h>
```

```

#include <stdio.h>
#include <jni.h>
#include <fcntl.h>
#include <assert.h>

// 获取数组的大小
# define NELEM(x) ((int) (sizeof(x) / sizeof((x)[0])))
// 指定要注册的类, 对应完整的 java 类名
#define JNIREG_CLASS "com/skywang/control/ControlService"

// 引入 log 头文件
#include <android/log.h>

// log 标签
#define TAG "WifiControl"
// 定义 debug 信息
#define LOGD(...) __android_log_print(ANDROID_LOG_DEBUG, TAG, __VA_ARGS__)
// 定义 error 信息
#define LOGE(...) __android_log_print(ANDROID_LOG_ERROR, TAG, __VA_ARGS__)

#define WIFI_ONOFF_CONTROL "/sys/devices/platform/misc_ctl/wifikey_onoff"

// 设置 wifi 电源开关
JNIEXPORT jint JNICALL is_wifi_key_down(JNIEnv *env, jclass clazz)
{
    int fd;
    int ret;
    char buf[2];

    // LOGD("%s \n", __func__);
    if((fd = open(WIFI_ONOFF_CONTROL, O_RDONLY)) < 0) {
        LOGE("%s : Cannot access \"%s\"", __func__, WIFI_ONOFF_CONTROL);
        return; // fd open fail
    }

    memset((void *)buf, 0x00, sizeof(buf));
    ssize_t count = read(fd, buf, 1);
    if (count == 1) {
        buf[count] = '\0';
        ret = atoi(buf);
    } else {
        buf[0] = '\0';
    }

    // LOGD("%s buf=%s, ret=%d\n", __func__, buf, ret);
    close(fd);
}

```

```

    return ret;
}

// 清除 wifi 的按下状态
JNIEXPORT void JNICALL clr_wifi_key_status(JNIEnv *env, jclass clazz)
{
    int fd;
    int nwr;
    char buf[2];

    if((fd = open(WIFI_ONOFF_CONTROL, O_RDWR)) < 0) {
        LOGE("%s : Cannot access \"%s\"", __func__, WIFI_ONOFF_CONTROL);
        return; // fd open fail
    }

    nwr = sprintf(buf, "%d\n", 0);
    write(fd, buf, nwr);

    LOGE("%s \n", __func__);

    close(fd);
}

// Java 和 JNI 函数的绑定表
static JNINativeMethod method_table[] = {
    // wifi 按键相关函数
    { "is_wifi_key_down", "()I", (void*)is_wifi_key_down },
    { "clr_wifi_key_status", "()V", (void*)clr_wifi_key_status },
};

// 注册 native 方法到 java 中
static int registerNativeMethods(JNIEnv* env, const char* className,
    JNINativeMethod* gMethods, int numMethods)
{
    jclass clazz;
    clazz = (*env)->FindClass(env, className);
    if (clazz == NULL) {
        return JNI_FALSE;
    }
    if ((*env)->RegisterNatives(env, clazz, gMethods, numMethods) < 0) {
        return JNI_FALSE;
    }

    return JNI_TRUE;
}

int register_wifi_control(JNIEnv *env)
{
    // 调用注册方法

```



```

        return registerNativeMethods(env, JNIREG_CLASS,
                                     method_table, NELEM(method_table));
    }

JNIEXPORT jint JNI_OnLoad(JavaVM* vm, void* reserved)
{
    JNIEnv* env = NULL;
    jint result = -1;

    if ((*vm)->GetEnv(vm, (void**) &env, JNI_VERSION_1_4) != JNI_OK) {
        return result;
    }

    register_wifi_control(env);

    // 返回 jni 的版本
    return JNI_VERSION_1_4;
}

```



代码说明:

- (01) Android 的 JVM 会回调 JNI_OnLoad() 函数。在 JNI_OnLoad() 中, 调用 register_wifi_control(env)。
- (02) register_wifi_control(env) 调用 registerNativeMethods(env, JNIREG_CLASS, method_table, NELEM(method_table)) 将 method_table 表格中的函数注册到 Android 的 JNIREG_CLASS 类中。JNIREG_CLASS 为 com/skywang/control/ControlService, 所以它对应的类是 com.skywang.control.ControlService.java。
- (03) method_table 的内容如下:

```

JNINativeMethod method_table[] = {
    // wifi 按键相关函数
    { "is_wifi_key_down", "()I", (void*)is_wifi_key_down },
    { "clr_wifi_key_status", "()V", (void*)clr_wifi_key_status },
}

```

这意味着, 将该文件中的 is_wifi_key_down() 函数和 JNIREG_CLASS 类的 is_wifi_key_down() 绑定。
将该文件中的 clr_wifi_key_status() 函数和 JNIREG_CLASS 类的 clr_wifi_key_status() 绑定。

该文件对应的 Android.mk 的代码如下:



```

LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_MODULE     := control_service
LOCAL_SRC_FILES  := control_service.c
# 添加对 log 库的支持
LOCAL_LDLIBS := -L$(SYSROOT)/usr/lib -llog

```

注：若生成 static 的.a，只需添加 LOCAL_LDLIBS:=-llog

```
include $(BUILD_SHARED_LIBRARY)
```

```
LOCAL_PATH := $(call my-dir)
```



用 ndk-build 编译上面两个文件，得到 so 库文件 libcontrol_service.so。

关于 Android NDK 编程更详细的内容，请参考“[Android JNI 和 NDK 学习](#)”

03 Android 读取 wifi 的开关/状态

在 Android 创建一个 com.skywang.control.ControlService.java。例如，在 Launcher 的目录下创建 packages/apps/Launcher2/src/com/skywang/control/ControlService.java
ControlService.java 代码如下：



```
package com.skywang.control;
```

```
import android.os.IBinder;
import android.app.Service;
import android.content.Intent;
import android.content.Context;
import android.net.wifi.WifiManager;
import android.util.Log;
```

```
public class ControlService extends Service {
    private static final String TAG = "ControlService";

    private WifiManager mWM;
    private ReadThread mReadThread;
    private boolean bWifi;

    @Override
    public void onCreate() {
        super.onCreate();

        Log.e(TAG, "start ControlService");
        mWM = (WifiManager) this.getSystemService(Context.WIFI_SERVICE);
        mReadThread = new ReadThread();
        mReadThread.start();

        bWifi = mWM.isWifiEnabled();
    }

    @Override
```

```

public void onDestroy() {
    super.onDestroy();

    if (mReadThread != null)
        mReadThread.interrupt();
}

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

// 处理 wifi 按键
private synchronized void handleWifiKey() {
    if (is_wifi_key_down()==1) {

        // 清空 wifi 的按下状态。目的是“防止不断的产生 wifi 按下事件”
        clr_wifi_key_status();
        Log.d(TAG, "wifi key down");
        if (!mWM.isWifiEnabled()) {
            Log.e(TAG, "open wifi");
            mWM.setWifiEnabled(true);
        } else {
            Log.e(TAG, "close wifi");
            mWM.setWifiEnabled(false);
        }
    }
}

// 和 Activity 界面通信的接口
private class ReadThread extends Thread {


    @Override
    public void run() {
        super.run();

        while (!isInterrupted()) {
            handleWifiKey();
        }
    }
}

// wifi 按键相关函数
private native int is_wifi_key_down();
private native void clr_wifi_key_status();

static {
    // 加载本地 .so 库文件
    System.loadLibrary("control_service");
}

```

```
}  
}  

```

代码说明：

(01) `System.loadLibrary("control_service");` 这是在 `ControlService` 启动的时候自动执行的，目的是加载 `libcontrol_service.so` 库。即上一步所生成的 `so` 库文件。

(02) `ControlService.java` 是服务程序，它继承于 `Service`。`ReadThread` 是启动时会自动开启的线程。`ReadThread` 的作用就是不断的调用 `handleWifiKey()` 处理 `wifi` 按键值。

接下来，我们在 `AndroidManifest.xml` 中声明该服务，就可以在其它地方调用执行了。下面是 `manifest` 中声明 `ControlService` 的代码：

```
<service android:name="com.skywang.control.ControlService">  
    <intent-filter >  
        <action android:name="com.skywang.control.CONTROLSERVICE" />  
    </intent-filter>  
</service>
```

我们在 `Launcher.java` 的 `onCreate()` 函数中启动该服务。这样，随着系统系统服务就会一直运行了。启动服务的代码如下：

```
startService(new Intent("com.skywang.control.CONTROLSERVICE"));
```

方案结论：工作正常，但消耗系统资源较多，会增加系统功耗！

经过测试发现，此方案运行很正常。但存在一个问题：由于添加了一个不停运行的服务，消耗很多系统资源，导致机器的功能也增加了很多。

因此，再实现方案三，对比看看效果如何。

方案三

方案思路：在 `Android` 的 `input` 输入子系统的框架层中捕获 `wifi` 按键，并进行相应处理。若捕获到“`wifi`”按键，则读取 `wifi` 的状态：若是“`wifi`”是开启的，则关闭 `wifi`；否则，打开 `wifi`。

方案分析：若采用此方案需要解决以下问题

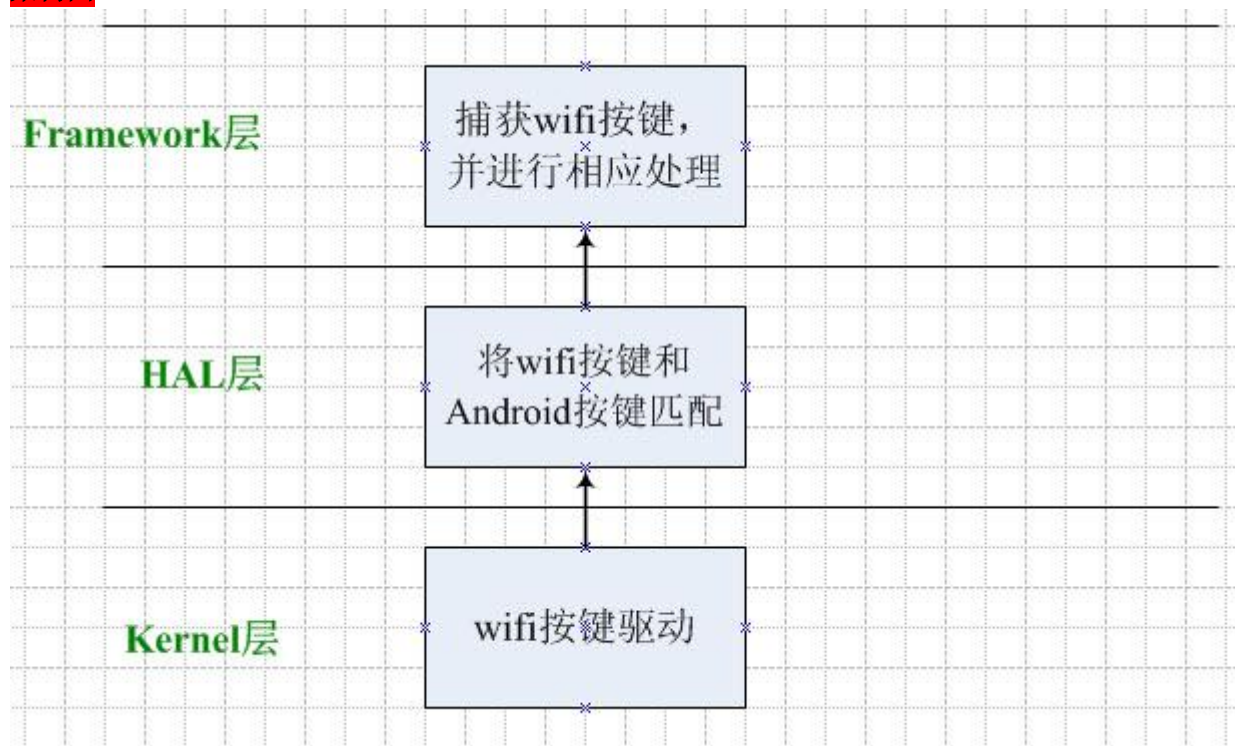
01，将 `kernel` 的 `wifi` 按键值映射到 `Android` 系统的某键值上。

— 这个可以实现。和“方案二”一样，我们通过 `ADC` 驱动将 `wifi` 按键映射到键值 `KEY_F16` 上；然后，将 `kernel` 的 `KEY_F16` 和 `Android` 的某一键值对应。

02, 在 Android 的 framework 层的键值处理函数中, 捕获按键, 并进行相应处理。

— 这个可以实现。在 input 子系统的 framework 层, 捕获到 wifi 按键对应的 Android 系统中的按键

架构图:



具体实现:

01, 将 kernel 的 wifi 按键值映射到 Android 系统的某键值上。

01.01, wifi 按键驱动

在按键驱动中编辑 wifi 按键的驱动: 主要的目的是将 wifi 按键映射到某个键值上, 方便后面 Android 系统调用。因为 Android 系统使用的按键值和 Linux 内核使用的按键值不一样, Android 会通过一个键值映射表, 将 Linux 的按键值和 Android 的按键值映射起来。

我们的项目中, wifi 按键是通过 ADC 值来捕获的, 而不是中断。下面是“wifi 按键相关信息”, 代码如下:



```
static struct rk29_keys_button key_button[] = {  
  
...  
// 将 wifi 开关按键定义为 KEY_F16,  
// 处理时, 捕获 KEY_F16 进行处理即可。  
{  
    .desc    = "wifi",  
    .code    = KEY_F16,  
    .adc_value = 4,  
    .gpio    = INVALID_GPIO,  
    .active_low = PRESS_LEV_LOW,  
},  
...  
}
```

```
};
```



从中，我们可以看出 wifi 的 adc 值大概是 4，它所对应的按键值(即 code 值)是 KEY_F16。

这里，KEY_F16 是我们自己定义的(因为 linux 中没有 wifi 开关按键)，你也可以定义为别的值。记得两点：一，这里的所定义的 wifi 的 code，必须和 Android 中要处理的按键值(后面会讲到)保持一致；二，不要使用系统中已用到的值。另外，KEY_F16 的值为 186，可以参考“include/linux/input.h”文件去查看。

在按键驱动中，会将 key_button 注册到系统中。在按键驱动中，我们将下面的 callback 函数注册到 adc 总线上；adc 驱动会通过工作队列，判断的读取 adc 值，并调用 callback，从而判断是否有响应的按键按下。下面是 callback 函数：



```
static void callback(struct adc_client *client, void *client_param, int result)
{
    struct rk29_keys_drvdata *ddata = (struct rk29_keys_drvdata *)client_param;
    int i;

    if(result < EMPTY_ADVALUE)
        ddata->result = result;

    // 依次查找 key_button 中的按键，判断是否需要响应
    for (i = 0; i < ddata->nbuttons; i++) {
        struct rk29_button_data *bdata = &ddata->data[i];
        struct rk29_keys_button *button = bdata->button;
        if(!button->adc_value)
            continue;
        int pre_state = button->adc_state;
        if(result < button->adc_value + DRIFT_ADVALUE &&
            result > button->adc_value - DRIFT_ADVALUE) {

            button->adc_state = 1;
        } else {
            button->adc_state = 0;
        }

        if(bdata->state != button->adc_state)
            mod_timer(&bdata->timer,
                jiffies + msecs_to_jiffies(DEFAULT_DEBOUNCE_INTERVAL));
    }
    return;
}
```



这里的 callback 和“方案二”中的 callback 有区别。这里没有调用 synKeyDone() 函数。

01.02, 键值映射

映射文件:

Linux 中的按键值和 Android 中的按键值不一样。它们是通过.kl 映射文件，建立对应关系的。

默认的映射文件是 qwerty.kl；但不同的平台可能有效的映射文件不同。用户可以通过查看

"/system/usr/keylayout/"目录下的.kl 映射文件，来进行验证哪个是有效的。映射方法：一，可以通过查看调用.kl 的代码。二，修改.kl 文件来验证。

在 rk3066 中，有效的映射文件是“rk29-keypad.kl”。在“rk29-keypad.kl”中添加以下代码将 wifi 按键和 Android 中的“AVR_POWER 按键”对应。

```
key 186 AVR_POWER
```

说明:

key -- 是关键字。固定值，不需要改变。

186 -- wifi 按键在 linux 驱动中对应的键值，这里对应 KEY_F16 的键值，即 wifi 对应的按键值。关于 linux 驱动中的各个键值，可以查看“include/linux/input.h”

AVR_POWER -- wifi 按键映射到 Android 中的按键，它对应是“KeycodeLabels.h”文件中的 KEYCODES 表格元素的“literal”值。

KeycodeLabels.h 中 KEYCODES 定义如下:



```
static const KeycodeLabel KEYCODES[] = {
    { "SOFT_LEFT", 1 },
    { "SOFT_RIGHT", 2 },
    { "HOME", 3 },
    { "BACK", 4 },
    { "CALL", 5 },
    { "ENDCALL", 6 },
    { "0", 7 },
    { "1", 8 },
    { "2", 9 },
    { "3", 10 },
    { "4", 11 },
    { "5", 12 },
    { "6", 13 },
    { "7", 14 },
    { "8", 15 },
    { "9", 16 },
    { "STAR", 17 },
    { "POUND", 18 },
    { "DPAD_UP", 19 },
    { "DPAD_DOWN", 20 },
    { "DPAD_LEFT", 21 },
    { "DPAD_RIGHT", 22 },
    { "DPAD_CENTER", 23 },
    { "VOLUME_UP", 24 },
    { "VOLUME_DOWN", 25 },
    { "POWER", 26 },
    { "CAMERA", 27 },
    { "CLEAR", 28 },
```

```
{ "A", 29 },
{ "B", 30 },
{ "C", 31 },
{ "D", 32 },
{ "E", 33 },
{ "F", 34 },
{ "G", 35 },
{ "H", 36 },
{ "I", 37 },
{ "J", 38 },
{ "K", 39 },
{ "L", 40 },
{ "M", 41 },
{ "N", 42 },
{ "O", 43 },
{ "P", 44 },
{ "Q", 45 },
{ "R", 46 },
{ "S", 47 },
{ "T", 48 },
{ "U", 49 },
{ "V", 50 },
{ "W", 51 },
{ "X", 52 },
{ "Y", 53 },
{ "Z", 54 },
{ "COMMA", 55 },
{ "PERIOD", 56 },
{ "ALT_LEFT", 57 },
{ "ALT_RIGHT", 58 },
{ "SHIFT_LEFT", 59 },
{ "SHIFT_RIGHT", 60 },
{ "TAB", 61 },
{ "SPACE", 62 },
{ "SYM", 63 },
{ "EXPLORER", 64 },
{ "ENVELOPE", 65 },
{ "ENTER", 66 },
{ "DEL", 67 },
{ "GRAVE", 68 },
{ "MINUS", 69 },
{ "EQUALS", 70 },
{ "LEFT_BRACKET", 71 },
{ "RIGHT_BRACKET", 72 },
{ "BACKSLASH", 73 },
{ "SEMICOLON", 74 },
{ "APOSTROPHE", 75 },
{ "SLASH", 76 },
{ "AT", 77 },
```



```
{ "NUM", 78 },
{ "HEADSETHOOK", 79 },
{ "FOCUS", 80 },
{ "PLUS", 81 },
{ "MENU", 82 },
{ "NOTIFICATION", 83 },
{ "SEARCH", 84 },
{ "MEDIA_PLAY_PAUSE", 85 },
{ "MEDIA_STOP", 86 },
{ "MEDIA_NEXT", 87 },
{ "MEDIA_PREVIOUS", 88 },
{ "MEDIA_REWIND", 89 },
{ "MEDIA_FAST_FORWARD", 90 },
{ "MUTE", 91 },
{ "PAGE_UP", 92 },
{ "PAGE_DOWN", 93 },
{ "PICTSYMBOLS", 94 },
{ "SWITCH_CHARSET", 95 },
{ "BUTTON_A", 96 },
{ "BUTTON_B", 97 },
{ "BUTTON_C", 98 },
{ "BUTTON_X", 99 },
{ "BUTTON_Y", 100 },
{ "BUTTON_Z", 101 },
{ "BUTTON_L1", 102 },
{ "BUTTON_R1", 103 },
{ "BUTTON_L2", 104 },
{ "BUTTON_R2", 105 },
{ "BUTTON_THUMBL", 106 },
{ "BUTTON_THUMBR", 107 },
{ "BUTTON_START", 108 },
{ "BUTTON_SELECT", 109 },
{ "BUTTON_MODE", 110 },
{ "ESCAPE", 111 },
{ "FORWARD_DEL", 112 },
{ "CTRL_LEFT", 113 },
{ "CTRL_RIGHT", 114 },
{ "CAPS_LOCK", 115 },
{ "SCROLL_LOCK", 116 },
{ "META_LEFT", 117 },
{ "META_RIGHT", 118 },
{ "FUNCTION", 119 },
{ "SYSRQ", 120 },
{ "BREAK", 121 },
{ "MOVE_HOME", 122 },
{ "MOVE_END", 123 },
{ "INSERT", 124 },
{ "FORWARD", 125 },
{ "MEDIA_PLAY", 126 },
```

```
{ "MEDIA_PAUSE", 127 },
{ "MEDIA_CLOSE", 128 },
{ "MEDIA_EJECT", 129 },
{ "MEDIA_RECORD", 130 },
{ "F1", 131 },
{ "F2", 132 },
{ "F3", 133 },
{ "F4", 134 },
{ "F5", 135 },
{ "F6", 136 },
{ "F7", 137 },
{ "F8", 138 },
{ "F9", 139 },
{ "F10", 140 },
{ "F11", 141 },
{ "F12", 142 },
{ "NUM_LOCK", 143 },
{ "NUMPAD_0", 144 },
{ "NUMPAD_1", 145 },
{ "NUMPAD_2", 146 },
{ "NUMPAD_3", 147 },
{ "NUMPAD_4", 148 },
{ "NUMPAD_5", 149 },
{ "NUMPAD_6", 150 },
{ "NUMPAD_7", 151 },
{ "NUMPAD_8", 152 },
{ "NUMPAD_9", 153 },
{ "NUMPAD_DIVIDE", 154 },
{ "NUMPAD_MULTIPLY", 155 },
{ "NUMPAD_SUBTRACT", 156 },
{ "NUMPAD_ADD", 157 },
{ "NUMPAD_DOT", 158 },
{ "NUMPAD_COMMA", 159 },
{ "NUMPAD_ENTER", 160 },
{ "NUMPAD_EQUALS", 161 },
{ "NUMPAD_LEFT_PAREN", 162 },
{ "NUMPAD_RIGHT_PAREN", 163 },
{ "VOLUME_MUTE", 164 },
{ "INFO", 165 },
{ "CHANNEL_UP", 166 },
{ "CHANNEL_DOWN", 167 },
{ "ZOOM_IN", 168 },
{ "ZOOM_OUT", 169 },
{ "TV", 170 },
{ "WINDOW", 171 },
{ "GUIDE", 172 },
{ "DVR", 173 },
{ "BOOKMARK", 174 },
{ "CAPTIONS", 175 },
```

```

{ "SETTINGS", 176 },
{ "TV_POWER", 177 },
{ "TV_INPUT", 178 },
{ "STB_POWER", 179 },
{ "STB_INPUT", 180 },
{ "AVR_POWER", 181 },
{ "AVR_INPUT", 182 },
{ "PROG_RED", 183 },
{ "PROG_GREEN", 184 },
{ "PROG_YELLOW", 185 },
{ "PROG_BLUE", 186 },
{ "APP_SWITCH", 187 },
{ "BUTTON_1", 188 },
{ "BUTTON_2", 189 },
{ "BUTTON_3", 190 },
{ "BUTTON_4", 191 },
{ "BUTTON_5", 192 },
{ "BUTTON_6", 193 },
{ "BUTTON_7", 194 },
{ "BUTTON_8", 195 },
{ "BUTTON_9", 196 },
{ "BUTTON_10", 197 },
{ "BUTTON_11", 198 },
{ "BUTTON_12", 199 },
{ "BUTTON_13", 200 },
{ "BUTTON_14", 201 },
{ "BUTTON_15", 202 },
{ "BUTTON_16", 203 },
{ "LANGUAGE_SWITCH", 204 },
{ "MANNER_MODE", 205 },
{ "3D_MODE", 206 },
{ "CONTACTS", 207 },
{ "CALENDAR", 208 },
{ "MUSIC", 209 },
{ "CALCULATOR", 210 },
{ "ZENKAKU_HANKAKU", 211 },
{ "EISU", 212 },
{ "MUHENKAN", 213 },
{ "HENKAN", 214 },
{ "KATAKANA_HIRAGANA", 215 },
{ "YEN", 216 },
{ "RO", 217 },
{ "KANA", 218 },
{ "ASSIST", 219 },

```

```

// NOTE: If you add a new keycode here you must also add it to several other files.
//       Refer to frameworks/base/core/java/android/view/KeyEvent.java for the full

```

list.

```
{ NULL, 0 }  
};
```



KeyCodeLabels.h 中的按键与 Android 框架层的 KeyEvent.java 中的按键值对应。

例如：“AVR_POWER”对应“KeyEvent.java 中的键值”如下：

```
public static final int KEYCODE_AVR_POWER = 181;
```

这样，我们发现：将驱动 wifi 按键就和 Android 系统中的 KEYCODE_AVR_POWER 就对应起来了！

02, 在 Android 的 framework 层的键值处理函数中，捕获按键，并进行相应处理。

在 framework 层的 input 系统中，加入对 wifi 按键的捕获。

添加的文件是：frameworks/base/policy/src/com/android/internal/policy/impl/PhoneWindowManager.java

添加的具体方法：在 PhoneWindowManager.java 的 interceptKeyBeforeQueueing() 函数中，捕获 wifi 按键。

代码如下：



```
public int interceptKeyBeforeQueueing(KeyEvent event, int policyFlags, boolean isScreenOn)
{
    final boolean down = event.getAction() == KeyEvent.ACTION_DOWN;
    final boolean canceled = event.isCanceled();
    final int keyCode = event.getKeyCode();

    final boolean isInjected = (policyFlags & WindowManagerPolicy.FLAG_INJECTED) != 0;

    final boolean keyguardActive = (mKeyguardMediator == null ? false :
        (isScreenOn ?
            mKeyguardMediator.isShowingAndNotHidden() :
            mKeyguardMediator.isShowing()));

    if (!mSystemBooted) {
        return 0;
    }

    if (DEBUG_INPUT) {
        Log.d(TAG, "interceptKeyTq keycode=" + keyCode
            + " screenIsOn=" + isScreenOn + " keyguardActive=" + keyguardActive);
    }

    if (down && (policyFlags & WindowManagerPolicy.FLAG_VIRTUAL) != 0
        && event.getRepeatCount() == 0) {
        performHapticFeedbackLw(null, HapticFeedbackConstants.VIRTUAL_KEY, false);
    }
}
```

```

if (keyCode == KeyEvent.KEYCODE_POWER) {
    policyFlags |= WindowManagerPolicy.FLAG_WAKE;
}
final boolean isWakeKey = (policyFlags & (WindowManagerPolicy.FLAG_WAKE
    | WindowManagerPolicy.FLAG_WAKE_DROPPED)) != 0;

int result;
if ((isScreenOn && !mHeadless) || (isInjected && !isWakeKey)) {
    // When the screen is on or if the key is injected pass the key to the application.
    result = ACTION_PASS_TO_USER;
} else {
    // When the screen is off and the key is not injected, determine whether
    // to wake the device but don't pass the key to the application.
    result = 0;
    if (down && isWakeKey) {
        if (keyguardActive) {
            // If the keyguard is showing, let it decide what to do with the wake key.
            mKeyguardMediator.onWakeKeyWhenKeyguardShowingTq(keyCode,
                mDockMode != Intent.EXTRA_DOCK_STATE_UNDOCKED);
        } else {
            // Otherwise, wake the device ourselves.
            result |= ACTION_POKE_USER_ACTIVITY;
        }
    }
}

// Handle special keys.
switch (keyCode) {
    case KeyEvent.KEYCODE_SYSRQ: {
        if (!down) {
            printScreenSysRq();
        }
        break;
    }
    case KeyEvent.KEYCODE_AVR_POWER: {
        Log.d("##SKYWANG##", "global keycode:"+keyCode);
        if (keyCode == KeyEvent.KEYCODE_AVR_POWER && down==false) {
            // Wifi 按键处理
            WifiManager mWM = (WifiManager)
mContext.getSystemService(Context.WIFI_SERVICE);
            boolean bWifi = mWM.isWifiEnabled();
            mWM.setWifiEnabled(!bWifi);
        }
        break;
    }
    case KeyEvent.KEYCODE_VOLUME_DOWN:
    case KeyEvent.KEYCODE_VOLUME_UP:
    case KeyEvent.KEYCODE_VOLUME_MUTE: {

```

```

if (keyCode == KeyEvent.KEYCODE_VOLUME_DOWN) {
    if (down) {
        if (isScreenOn && !mVolumeDownKeyTriggered
            && (event.getFlags() & KeyEvent.FLAG_FALLBACK) == 0) {
            mVolumeDownKeyTriggered = true;
            mVolumeDownKeyTime = event.getDownTime();
            mVolumeDownKeyConsumedByScreenshotChord = false;
            cancelPendingPowerKeyAction();
            interceptScreenshotChord();
        }
    } else {
        mVolumeDownKeyTriggered = false;
        cancelPendingScreenshotChordAction();
    }
} else if (keyCode == KeyEvent.KEYCODE_VOLUME_UP) {
    if (down) {
        if (isScreenOn && !mVolumeUpKeyTriggered
            && (event.getFlags() & KeyEvent.FLAG_FALLBACK) == 0) {
            mVolumeUpKeyTriggered = true;
            cancelPendingPowerKeyAction();
            cancelPendingScreenshotChordAction();
        }
    } else {
        mVolumeUpKeyTriggered = false;
        cancelPendingScreenshotChordAction();
    }
} else if (keyCode == KeyEvent.KEYCODE_VOLUME_MUTE) {
    // add by skywang
    if (!down)
        handleMuteKey();
}

if (down) {
    ITelephony telephonyService = getTelephonyService();
    if (telephonyService != null) {
        try {
            if (telephonyService.isRinging()) {
                // If an incoming call is ringing, either VOLUME key means
                // "silence ringer". We handle these keys here, rather than
                // in the InCallScreen, to make sure we'll respond to them
                // even if the InCallScreen hasn't come to the foreground yet.
                // Look for the DOWN event here, to agree with the "fallback"
                // behavior in the InCallScreen.
                Log.i(TAG, "interceptKeyBeforeQueueing:"
                    + " VOLUME key-down while ringing: Silence ringer!");

                // Silence the ringer. (It's safe to call this
                // even if the ringer has already been silenced.)
                telephonyService.silenceRinger();
            }
        }
    }
}

```

```

        // And *don't* pass this key thru to the current activity
        // (which is probably the InCallScreen.)
        result &= ~ACTION_PASS_TO_USER;
        break;
    }
    if (telephonyService.isOffhook()
        && (result & ACTION_PASS_TO_USER) == 0) {
        // If we are in call but we decided not to pass the key to
        // the application, handle the volume change here.
        handleVolumeKey(AudioManager.STREAM_VOICE_CALL, keyCode);
        break;
    }
} catch (RemoteException ex) {
    Log.w(TAG, "ITelephony threw RemoteException", ex);
}
}

if (isMusicActive() && (result & ACTION_PASS_TO_USER) == 0) {
    // If music is playing but we decided not to pass the key to the
    // application, handle the volume change here.
    handleVolumeKey(AudioManager.STREAM_MUSIC, keyCode);
    break;
}
}
break;
}

case KeyEvent.KEYCODE_ENDCALL: {
    result &= ~ACTION_PASS_TO_USER;
    if (down) {
        ITelephony telephonyService = getTelephonyService();
        boolean hungUp = false;
        if (telephonyService != null) {
            try {
                hungUp = telephonyService.endCall();
            } catch (RemoteException ex) {
                Log.w(TAG, "ITelephony threw RemoteException", ex);
            }
        }
        interceptPowerKeyDown(!isScreenOn || hungUp);
    } else {
        if (interceptPowerKeyUp(canceled)) {
            if ((mEndcallBehavior
                & Settings.System.END_BUTTON_BEHAVIOR_HOME) != 0) {
                if (goHome()) {
                    break;
                }
            }
        }
    }
}
}

```

```

        if ((mEndcallBehavior
            & Settings.System.END_BUTTON_BEHAVIOR_SLEEP) != 0) {
            result = (result & ~ACTION_POKE_USER_ACTIVITY) |
ACTION_GO_TO_SLEEP;
        }
    }
}
break;
}

case KeyEvent.KEYCODE_POWER: {

if(mHdmiPlugged&&SystemProperties.get("ro.hdmi.power_disable","false").equals("true")){
    Log.d("hdmi","power disable-----");
    result=0;
    break;
}
result &= ~ACTION_PASS_TO_USER;
if (down) {
    if (isScreenOn && !mPowerKeyTriggered
        && (event.getFlags() & KeyEvent.FLAG_FALLBACK) == 0) {
        mPowerKeyTriggered = true;
        mPowerKeyTime = event.getDownTime();
        interceptScreenshotChord();
    }

    ITelephony telephonyService = getTelephonyService();
    boolean hungUp = false;
    if (telephonyService != null) {
        try {
            if (telephonyService.isRinging()) {
                // Pressing Power while there's a ringing incoming
                // call should silence the ringer.
                telephonyService.silenceRinger();
            } else if ((mIncallPowerBehavior
                &
Settings.Secure.INCALL_POWER_BUTTON_BEHAVIOR_HANGUP) != 0
                && telephonyService.isOffhook()) {
                // Otherwise, if "Power button ends call" is enabled,
                // the Power button will hang up any current active call.
                hungUp = telephonyService.endCall();
            }
        } catch (RemoteException ex) {
            Log.w(TAG, "ITelephony threw RemoteException", ex);
        }
    }
    interceptPowerKeyDown(!isScreenOn || hungUp
        || mVolumeDownKeyTriggered || mVolumeUpKeyTriggered);
} else {

```



```

        mPowerKeyTriggered = false;
        cancelPendingScreenshotChordAction();
        if (interceptPowerKeyUp(canceled || mPendingPowerKeyUpCanceled)) {
            result = (result & ~ACTION_POKE_USER_ACTIVITY) | ACTION_GO_TO_SLEEP;
        }
        mPendingPowerKeyUpCanceled = false;
    }
    break;
}

case KeyEvent.KEYCODE_MEDIA_PLAY:
case KeyEvent.KEYCODE_MEDIA_PAUSE:
case KeyEvent.KEYCODE_MEDIA_PLAY_PAUSE:
    if (down) {
        ITelephony telephonyService = getTelephonyService();
        if (telephonyService != null) {
            try {
                if (!telephonyService.isIdle()) {
                    // Suppress PLAY/PAUSE toggle when phone is ringing or in-call
                    // to avoid music playback.
                    break;
                }
            } catch (RemoteException ex) {
                Log.w(TAG, "ITelephony threw RemoteException", ex);
            }
        }
    }

case KeyEvent.KEYCODE_HEADSETHOOK:
case KeyEvent.KEYCODE_MUTE:
case KeyEvent.KEYCODE_MEDIA_STOP:
case KeyEvent.KEYCODE_MEDIA_NEXT:
case KeyEvent.KEYCODE_MEDIA_PREVIOUS:
case KeyEvent.KEYCODE_MEDIA_REWIND:
case KeyEvent.KEYCODE_MEDIA_RECORD:
case KeyEvent.KEYCODE_MEDIA_FAST_FORWARD: {
    if ((result & ACTION_PASS_TO_USER) == 0) {
        // Only do this if we would otherwise not pass it to the user. In that
        // case, the PhoneWindow class will do the same thing, except it will
        // only do it if the showing app doesn't process the key on its own.
        // Note that we need to make a copy of the key event here because the
        // original key event will be recycled when we return.
        mBroadcastWakeLock.acquire();
        Message msg =
mHandler.obtainMessage(MSG_DISPATCH_MEDIA_KEY_WITH_WAKE_LOCK,
                        new KeyEvent(event));
        msg.setAsynchronous(true);
        msg.sendToTarget();
    }
    break;
}

```

```

    }

    case KeyEvent.KEYCODE_CALL: {
        if (down) {
            ITelephony telephonyService = getTelephonyService();
            if (telephonyService != null) {
                try {
                    if (telephonyService.isRinging()) {
                        Log.i(TAG, "interceptKeyBeforeQueueing:"
                            + " CALL key-down while ringing: Answer the call!");
                        telephonyService.answerRingingCall();

                        // And *don't* pass this key thru to the current activity
                        // (which is presumably the InCallScreen.)
                        result &= ~ACTION_PASS_TO_USER;
                    }
                } catch (RemoteException ex) {
                    Log.w(TAG, "ITelephony threw RemoteException", ex);
                }
            }
        }
        break;
    }
}
return result;
}

```



在上面的代码中，我们捕获了 `KeyEvent.KEYCODE_AVR_POWER`，并对其进行处理。

方案结论：方案可行。而且运行效率比“方案二”高，不会造成功耗很大的问题。

最终总结：方案三最好！