

Grandstream Networks, Inc.

潮流网络 Android SDK 使用指南



技术支持

深圳市潮流网络技术有限公司为客户提供全方位的技术支持。您可以与本地代理商或服务提供商联系，也可以与公司总部直接联系。

地址：深圳市南山区科技园本区新西路 16 号彩虹科技大厦 4 楼

邮编：518057

网址：<http://www.grandstream.cn>

客服电话：0755-26014600

客服传真：0755-26014601

技术支持热线：4008755751

技术支持论坛：<http://forums.grandstream.com>

网上问题提交系统：<http://www.grandstream.com/support/submit-a-ticket>

商标注明



和其他潮流网络商标均为潮流网络技术有限公司的商标。本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

注意

由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。



目录

支持产品.....	8
变更记录.....	9
SDK 框架服务版本 1.1.15.....	9
SDK 框架服务版本 1.1.14.....	9
SDK 框架服务版本 1.1.13.....	9
SDK 框架服务版本 1.1.12.....	9
SDK 框架服务版本 1.1.11.....	9
SDK 框架服务版本 1.1.10.....	9
SDK 框架服务版本 1.1.9.....	10
SDK 框架服务版本 1.1.8.....	10
SDK 框架服务版本 1.1.7.....	10
SDK 框架服务版本 1.1.6.....	10
概览.....	11
应用编译.....	12
在 APP 模块目录下, 创建 “gslibs”	12
Build.gradle.....	12
JAR 中主要的 API	13
使用 APIS.....	14
初始化 ApiClient	14
帐号 BaseAccountApi	15
帐号信息.....	15
监控帐号状态	15
启动帐号状态监控	15
关闭帐号状态监控	15
AccountStatusListener 实现帐号监控.....	16
更新帐号.....	16
通话 BaseCallApi.....	16



拨打电话.....	16
结束通话.....	17
监控来电.....	17
启动通话状态监控	17
关闭通话状态监控	18
CallStatusListener 实现通话状态监控	18
监控手柄摘/挂机状态.....	18
启动手柄摘机监控	18
停止手柄摘机监控	19
CallStatusListener 实现摘机状态监控	19
转移通话通话设置 CallSettingApi.....	19
默认 Phone 应用	20
实现定制 Phone 服务.....	20
注册自己的 Phone 服务.....	21
配置默认 Phone 应用	21
短信息 SmsManagerApi	22
发送短消息.....	22
重发消息.....	22
接收短消息.....	23
删除短信.....	23
声音通道 AudioRouteApi	24
切换声音通道.....	24
检查当前声音通道.....	24
EHS 耳机.....	25
总以扬声器振铃.....	25
通话设置 CallSettingApi.....	26
DeviceApi	26
CallToneApi	26
Gs Core SDK.....	28
设备管理.....	29
获取设备连接状态	29
重启设备	29



DND 设置	29
获取设备 MAC 地址	29
联系人	31
数据库 ContactsContract.Data 的变更	31
数据库 ContactsContract.RawContacts 的变更	31
ContactsContract.Contacts#CONTENT_FILTER_URI 字段变更	32
新增联系人	32
更改联系人	33
查询联系人	34
删除联系人	34
通话记录	36
CallLog.Calls 表的变更内容	36
新增通话记录	37
查询通话记录	38
更改通话记录	38
删除通话记录	39
LED	40
GsLightsManager	41
可编程键	42
监控扩展板点击事件	42
启动扩展板点击事件监控	42
关闭扩展板点击事件监控	42
MpkStatusListener 实现扩展板点击事件监控	42
WP820 的特殊接口	43
安全告警接口使用说明	43
安全告警广播定义	43
开启安全告警	43
自定义处理安全告警广播	43



KEY Code 定义	44
ADB 使用说明	45
打开设备开发者模式	45
adb 连接	45
设备允许调试	45
检查 adb 连接状态	45
adb 断开连接	45
API DEMO	46



图目录

图 1: 创建 gslibs 目录.....	12
图 2: 默认 Phone 应用	21

表目录

表 1: 支持产品.....	8
表 2: 包和描述.....	13
表 3: 接口类和描述.....	13
表 4: ContactsContract.Data 变更字段	31
表 5: ContactsContract.RawContacts 变更字段	31
表 6: CallLog.Calls 变化	36
表 7: LED 颜色和频率	40



支持产品

下表列出了本指南中支持 SDK API 的 Grandstream 产品：

表 1：支持产品

产品型号	是否支持	支持固件
GXV3370	支持	1.0.2.6+
GXV3380	支持	1.0.1.9+
WP820	支持	1.0.3.5+
GXV3350	支持	1.0.1.8+



WP820 不支持上述 1.1.7 版本的 SDK 框架服务提供的新功能。



变更记录

本节介绍 GXV3370 / WP820 / GXV3380 / GXV3350 SDK 框架服务指南的历史变更记录。此处只列出主要新功能或主要文档更新，更正或编辑的次要更新则不列出。

SDK 框架服务版本 1.1.15

- 增加 DND 控制功能。[Gs Core SDK]
- 在实现定制 Phone 服务中增加回调接口。[实现定制 Phone 服务]

SDK 框架服务版本 1.1.14

- 新增通话音接口。[CallToneApi]
- 增加获取线路呼叫 id 和本地号码接口。[BaseLine]
- 增加获取帐号注册状态接口。[BaseAccount]

SDK 框架服务版本 1.1.13

- 新增通话转移功能。[转移通话]
- 在 PhoneContext 中增加线路状态。

SDK 框架服务版本 1.1.12

- 支持 GXV3350 通过 GBX20 添加可编程键。[可编程键]
- 新增设备管理和系统信息获取的 DeviceApi。[DeviceApi]

SDK 框架服务版本 1.1.11

- 支持 GXV3350。
- 新增删除短信功能。[短信息 SmsManagerApi]

SDK 框架服务版本 1.1.10

- 新增控制 LED 等的新功能。[LED]
- 新增获取设备 MAC 地址的功能。



SDK 框架服务版本 1.1.9

- 移除系统键值存储 NvramApi 和 NvramManager。

SDK 框架服务版本 1.1.8

- 新增系统键值存储 NvramManager 和 NvramApi。[系统键值处理 NvramApi 和 NvramManager]
- 新增设备管理 GsDevStateManager，包括设备连接状态和设备重启。[Gs Core SDK]
- 更新紧急呼叫的默认 Phone 应用。[默认 Phone 应用]
- 更新 Update BaseCallApi 中的 send DTMF。[通话设置 BaseCallApi]
- 更新其他内容。

SDK 框架服务版本 1.1.7

- 新增“总以扬声器振铃”的新 Apis。[总以扬声器振铃]
- 新增通话设置 CallSettingApi。[通话设置 CallSettingApi]
- 新增声音通道 AudioRouteApi 的更多功能。[声音通道 AudioRouteApi]

SDK 框架服务版本 1.1.6

- 初始版本



概览

GXV3370 / GXV3380 / GXV3350 / WP820 操作系统是基于 Android™ 平台开发的。除了继承 Android 界面功能外，还根据用户需求添加了其他界面。本文档将具体描述如何利用 API 开发应用程序。



在进行 API 演示或测试您自己的应用程序之前，请将 GXV3370 / GXV3380 / GXV3350 / WP820 升级到最新的固件版本。固件版本信息可在以下链接中找到：

<http://www.grandstream.com/support/firmware>



应用编译

用户依赖 GS JARs 使用 Android Studio（或其他 IDE）进行开发，具体请参考以下步骤。

在 APP 模块目录下，创建 “gslibs”

首先，在 APP 模块下创建 “gslibs” 目录。然后，复制 “gslibs” 目录下的 JAR 文件 “gsapi-1.1.6.jar”，如下图所示，以便可以从应用程序文件访问 JAR 文件。

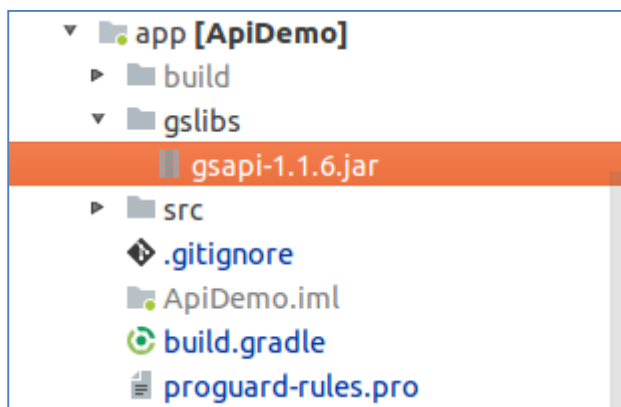


图 1：创建 gslibs 目录

Build.gradle

使用 `compileOnly` 或 `provided` 在 `build.gradle` 文件中添加依赖项，使用参考如下：

```
dependencies {
    .....
    compileOnly files ('gslibs/gsapi-1.1.6.jar')
}
```



不需要将 jar 包打包进 APK。



下表列出了编译应用程序时可以包含的 JAR 包。如果需要使用它们，应添加到 `build.gradle` 文件中。

表 2: 包和描述

包	描述
com.gs.common	支持初始化相关接口 <code>ApiClient</code>
com.gs.account	支持帐号相关接口
com.gs.phone	支持电话相关接口
com.gs.contacts	支持联系人相关接口
com.gs.calllog	支持通话记录相关接口
com.gs.sms	支持短信息相关接口



注意:

所有类、方法和类成员都不是任何受支持的 API 的一部分，并且这些 API 均未在 `gsApiExternal-zh` 中定义。如果编写的代码取决于这些，请自行承担风险。此代码及其内部接口可能会更改或删除，恕不另行通知。

JAR 中主要的 API

以下列举了 JAR 中包含的主要 API 接口类:

表 3: 接口类和描述

API 接口类	描述
AudioRoutetApi	声音通道相关接口，用于切换声音通道和检查声音通道。
BaseAccountApi	帐号相关接口，用于通话相关。
BaseLineApi	线路相关接口，用于打电话或者来电创建的线路通道管理。
BaseCallApi	打电话相关接口，可以拨打电话和接收来电等。
BasePhoneService	默认 Phone 应用开发接口。
ContactsContext	用于联系人增删查改。
CalllogContext	用于通话记录的增删查改。
SmsManagerAPI	可以发送和监听短信息等。
CallToneApi	用于通话音的播放，例如来电铃声、呼叫等待音、忙音、拨号音、DTMF 音、确认音、自动应答提示音、DND 提示音、续订音等



使用 APIs

所有 API 都采用 **XXXApi** 的规则命名，例如：**BaseAccountApi**、**BaseCallApi**、**BaseLineApi**、**SmsManagerApi** 等。

- 在使用 APIs 之前，用户必须先初始化 **ApiClient**
- 用 **XXXApi.function1()** 的方式调用接口方法
- 用 **XXXApi.function2(callback2)** 的方式添加监听

API 接口的详细定义请查看 **gsApiExternal-zh**。

初始化 ApiClient

支持的产品：**GXV3370**、**GXV3380**、**GXV3350**、**WP820**

App 可以使用 **com.gs.com** 上的 **ApiClient**。在 **Application** 块中，使用 **setContext**、**addApi**、**build** 方法可以初始化 **ApiClient**。



注意：

每个应用程序进程只能执行一次初始化，请勿对同一个流程多次初始化。

以下是有关初始化 **ApiClient** 的例子：

```
public class DemoApplication extends Application {
    @Override
    public void onCreate() {
        super.onCreate();

        ApiClient.builder.setContext (getApplicationContext())
            .addApi (BaseAccountApi.API)
            .addApi (BaseCallApi.API)
            .addApi (BaseLineApi.API)
            .addApi (SmsManagerApi.API)
            .addApi (AudioRouteApi.API)
            .build();
    }
}
```



帐号 BaseAccountApi

支持的产品: GXV3370、GXV3380、GXV3350、WP820

帐号 API 可用于获取帐号信息、修改帐号信息和监听帐号变化等。

帐号信息

使用 **BaseAccountApi.getAllSipAccounts** 可以获取所有帐号的信息。

以下是有关如何获取帐号信息的例子:

```
public void getAllAccount() {  
    List<SipAccount> sipAccounts = BaseAccountApi.getAllSipAccounts();  
}
```

监控帐号状态

用户可以使用 **BaseAccountApi.addStatusListener** 和 **BaseAccountApi.removeStatusListener** 来监控帐号状态变化。

以下是有关如何监控帐号状态的例子:

启动帐号状态监控

```
private void startMonitorAccountChange() {  
    mAccountStatusListener = new AccountStatusListener();  
    BaseAccountApi.addStatusListener ("MonitorAccount",  
        mAccountStatusListener.callback,  
        AccountContext.ListenType.SIP_ACCOUNT_STATUS, false);  
}
```

关闭帐号状态监控

```
private void stopMonitorAccountChange() {  
    BaseAccountApi.removeStatusListener (mAccountStatusListener.callback);  
    mAccountStatusListener.callback.destroy();  
    mAccountStatusListener.callback = null;  
}
```



AccountStatusListener 实现帐号监控

```
private class MyAccountStatusListener extends AccountStatusListener {  
    @Override  
    public void onSipAccountStatusChanged(List<SipAccount> list, SipAccount  
sipAccount) {  
    }  
};  
  
AccountStatusListener mAccountStatusListener = null;
```

更新帐号

用户可以使用 **BaseAccountApi.updateSipAccount** 来更新帐号信息。

以下是有关如何更新帐号信息的例子：

```
public void updateAccount() {  
  
    BaseAccount account= BaseAccountApi.getAccountById(0);  
    Log.d(TAG, "updateAccount, original:"+account);  
    if(account != null && account instanceof SipAccount){  
        account.setAccountName("Test Account");  
        boolean ret = BaseAccountApi.updateSipAccount((SipAccount) account);  
        if(ret){  
            Log.d(TAG, "updateAccount, changed to:"+account);  
        }  
    }  
}
```

通话 BaseCallApi

通话接口可以用于拨打电话、结束通话和监控来电。

拨打电话

支持的产品：GXV3370、GXV3380、GXV3350、WP820

用户可以使用 **BaseCallApi.makeCalls** 拨打电话。

以下是有关如何拨打电话的例子：

```
public void call(String num) {  
    List<DialingInfo> dialingInfos = new ArrayList<DialingInfo>();
```



```
DialingInfo dialingInfo = new DialingInfo();
dialingInfo.setAccountID(BaseAccountApi.getDefaultAccount().getAccountID());
dialingInfo.setOriginNumber(num);
dialingInfos.add(dialingInfo);
DialResults dialResults = BaseCallApi.makeCalls(dialingInfos);
int result = dialResults.getDialResult(dialingInfo);
}
```

结束通话

支持的产品: GXV3370、GXV3380、GXV3350、WP820

用户可以使用 **BaseCallApi.endCall** 结束通话。

以下是有关如何结束通话的例子:

```
public void endCall(View view) {
    DemoApplication app = (DemoApplication) getApplication();
    int lineId = app.getCurLineId();
    BaseCallApi.endCall(lineId);
}
```

监控来电

支持的产品: GXV3370、GXV3380、GXV3350、WP820

如果要接听来电,用户需要先对来电进行监听。

以下是有关如何监听来电的例子:

启动通话状态监控

要启动监听线路状态,需要使用带有 **LINE_STATUS** 的 **BaseCallApi.addStatusListener** 接口,并监控具有 **LINE_ID** 的呼叫线路。

```
private void startMonitorCallLines() {
    mCallStatusListener = new MyCallStatusListener();
    BaseCallApi.addStatusListener ("MonitorCall",
        mCallStatusListener.callback,
        PhoneContext.ListenType.LINE_STATUS,
        false);
}
```



关闭通话状态监控

```
private void stopMonitorCallLines() {
    BaseCallApi.removeStatusListener (mCallStatusListener.callback);
    mCallStatusListener.callback.destroy();
    mCallStatusListener.callback = null;
}
```

CallStatusListener 实现通话状态监控

```
private class MyCallStatusListener extends CallStatusListener {
    @Override
    public void onLineStatusChanged(int notifyType, BaseLine baseline,
    List<Baseline> list) {
    }

    @Override
    public void onLineIdChanged(int oldLineId, int newLineId) {
    }
}

private CallStatusListener mCallStatusListener = null;
```

监控手柄摘/挂机状态

支持的产品: GXV3370、GXV3380、GXV3350

与监听来电相同，手柄监听也使用 **add/removeStatusListener**。

以下是有关如何监控手柄摘机事件状态的例子：

启动手柄摘机监控

用户可以使用带有 **HOOK_EVENT_STATUS** 的 **BaseCallApi.addStatusListener** 接口来监控手柄摘机事件状态。通话设置 **CallSettingApi**

```
private void startMonitorHandsetChange() {
    mHookStatusListener = new MyHookStatusListener();
    BaseCallApi.addStatusListener ("MonitorHandset",
    mHookStatusListener.callback,
    PhoneContext.ListenType.HOOK_EVENT_STATUS, false);
}
```



停止手柄摘机监控

```
private void stopMonitorHandsetChange() {  
    BaseCallApi.removeStatusListener (mHookStatusListener.callback);  
    mHookStatusListener.callback.destroy();  
    mHookStatusListener.callback = null; 通话设置 CallSettingApi  
}
```

CallStatusListener 实现摘机状态监控

```
private class MyHookStatusListener extends CallStatusListener {  
    @Override  
    public void onHookEventChanged(int device, boolean isOffHook) {  
    }  
}  
  
private CallStatusListener mHookStatusListener = null;
```

转移通话 通话设置 CallSettingApi

支持的产品: GXV3370、GXV3380、GXV3350、WP820

用户可以通过 **transferBlind/transferAttended** 进行盲转移和指定转移, 转移前需要先保持当前通话。

盲转移后当前通话直接结束, 被转移方直接呼叫转移目标。通话设置 CallSettingApi

支持的产品: GXV3370、GXV3380、GXV3350、WP820

指定转移后当前通话不会被结束, 由当前设备新呼叫转移目标, 在转移目标未接听时可以取消转移/立即完成转移; 当转移目标接听后可以立即完成转移/分离。

相关接口如下:

```
BaseCallApi.transferBlind(int lineId, String number) ; //盲转接  
BaseCallApi.transferAttended(int lineId, String number) ; //指定转接  
BaseCallApi.transferAttendedCancel(int lineId) ; //指定转移过程中转移目标未接听时取消指定转移  
BaseCallApi.endCall(int lineId) ; //指定转移过程中转移目标未接听时结束当前通话以完成指定转移  
BaseCallApi.transferAttendedEnd(int lineId) ; //当转移目标接通后结束当前通话以完成指定转移  
BaseCallApi.transferSplit() ; //指定转移过程中当转移目标接听之后用于分离两路通话
```



默认 Phone 应用

支持的产品: GXV3370、GXV3380、GXV3350

一般来说, 系统的 Phone 是默认的, 并且可以满足大多数情况的需求。这里, 我们仍然提供定制化 Phone 的接口, 让你自己的 Phone 应用成为默认的 Phone 应用。

基于接口可以开发自定义 Phone 应用。通过将自定义 Phone 设置成默认的 Phone 应用就可以成为唯一收到摘机、来电、线路变化事件的应用。应用需要实现指定的 PhoneService 以及相关的业务逻辑。当然, 即使将自定义 Phone 应用设置成默认, 系统的 Phone 仍然可以用于拨号。

实现定制 Phone 服务

自定义自己的电话服务, 需要扩展 **BasePhoneService**, 并处理摘/挂机事件。

```
public class DemoService extends BasePhoneService {  
    @Override  
    public void onHookEvent(boolean OffHook) {  
        super.onHookEvent(offHook);  
        //TODO:do something  
    }  
  
    public void onEhsHookEvent(boolean OffHook) {  
        super.onEhsHookEvent(offHook);  
        //TODO:do something  
    }  
  
    public void onLineStateChanged(int lineId, int status) {  
        super.onLineStateChanged(lineId, status);  
        //TODO:do something  
    }  
  
    /**  
     * return true means this click event was cost by this service.  
     * else this event will call the system emergency dialer.  
     */  
    public boolean onEmergencyCallButtonClicked(int lineId, int status) {  
        super.onLineStateChanged(lineId, status);  
        //TODO:do something  
        return true;  
    }  
}
```



```

/**
 * 通话时点击顶部状态栏"点击返回通话"时该方法区域会被调用,可以再这里启动通话界面
 */
@Override
    public void onShowCallingView() {
        super.onShowCallingView();
    }
}

```

注册自己的 Phone 服务

```

<service android:name=".DemoService" >
    <intent-filter>
        <action android:name="com.gs.phone.service"/>
    </intent-filter>
</service>

```

配置默认 Phone 应用

1. 安装自定义 Phone 应用。
2. 打开系统设置，进入应用→默认应用程序→电话应用。
3. 选择自定义安装的 Phone 应用作为默认 Phone 应用。

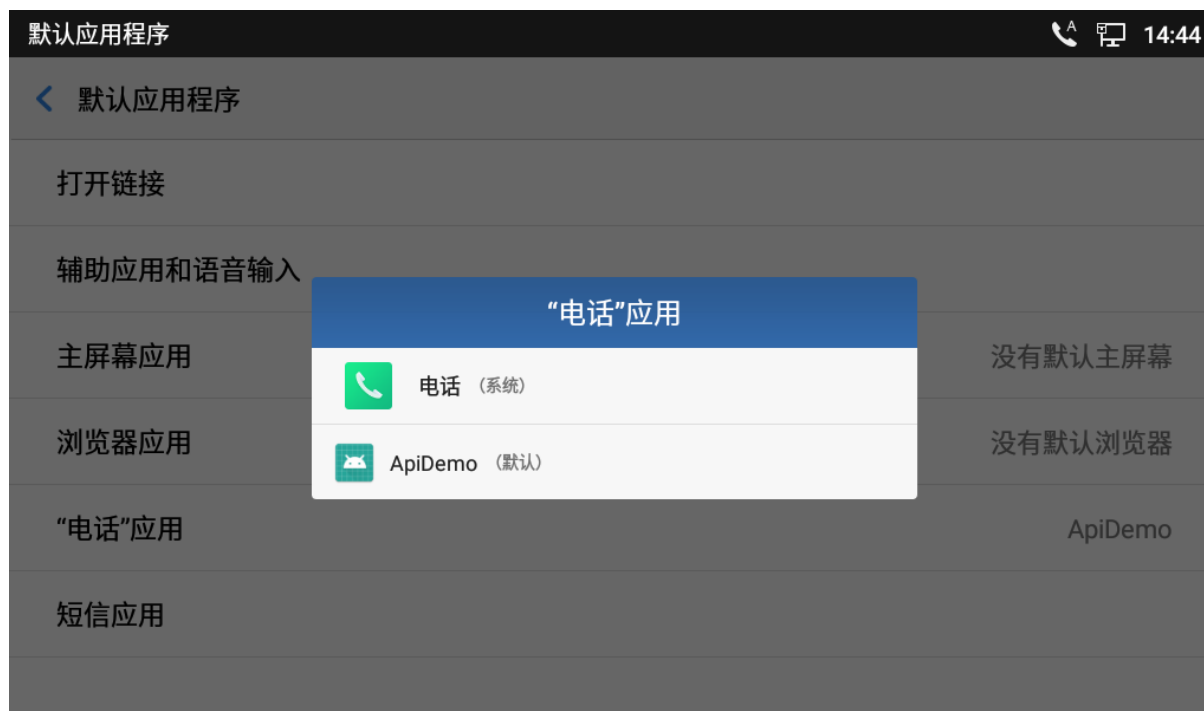


图 2: 默认 Phone 应用



短信息 SmsManagerApi

支持的产品: GXV3370、GXV3380、GXV3350、WP820

SMS 接口允许用户发送和接收短信息，并且可以继续操作发送失败的短信息。

发送短消息

用户可以使用 **SmsManagerApi.sendSmsAfterSaved** 接口向单个联系人或者一组联系人发送短信息。

```
int accountId = 0;
String numberStr = "36324";
String content = "Test SMS to single one.";
long msgId = SmsManagerApi.sendSmsAfterSaved(accountId, numberStr, content);
if(msgId > -1){
    Log.d(TAG, "Sending sms (" + msgId + ") ..");
}

String group = numberStr + "_," + "36325";
content = "Test SMS to a group of contacts.";
msgId = SmsManagerApi.sendSmsAfterSaved(accountId, group, numberStr, content);
if(msgId > -1){
    Log.d(TAG, "Sending sms (" + msgId + ") ..");
}
```

用户可以使用 **SmsManagerApi.addSmsListener** 接口监听短信息发送结果。

```
SmsManagerApi.addSmsListener(new SmsListener() {
    @Override
    public void onSend(long msgId, boolean sendOk) {
        Log.d(TAG, "This is main thread.");
        Log.d(TAG, "sms (" + msgId + ") is send " + (sendOk ? "success" : "fail"));
    }
});
```

重发消息

用户可以使用 **SmsManagerApi.resendFailedSms** 接口重发失败的短信息。

```
boolean ret = SmsManagerApi.resendFailedSms(msgId);
if(ret){
    Log.d(TAG, "ReSending sms (" + msgId + ") ..");
}else{
```



```
Log.e(TAG, "ReSending sms(" + msgId + ") fail.");  
}
```

接收短消息

用户可以使用 **SmsContext.SMS_RECEIVED_ACTION** 注册 **BroadcastReceiver** 来接收短消息，并使用定义在 **SmsContext.ReceiveSms** 中的常量来读取短消息的内容。示例如下：

```
private SmsReceiver smsReceiver;  
private void registerSmsReceiver(){  
    IntentFilter filter = new IntentFilter();  
    filter.addAction(SmsContext.SMS_RECEIVED_ACTION);  
    smsReceiver = new SmsReceiver();  
    registerReceiver(smsReceiver, filter);  
}  
private void unregisterSmsReceiver(){  
    if(smsReceiver != null){  
        unregisterReceiver(smsReceiver);  
    }  
}  
class SmsReceiver extends BroadcastReceiver{  
    @Override  
    public void onReceive(Context context, Intent intent) {  
        if(SmsContext.SMS_RECEIVED_ACTION.equals(intent.getAction())){  
            long id =  
intent.getLongExtra(SmsContext.ReceiveSms.ID, -1);  
            String number =  
intent.getStringExtra(SmsContext.ReceiveSms.NUMBER);  
            int accountId =  
intent.getIntExtra(SmsContext.ReceiveSms.ACCOUNT_ID, -1);  
            String content =  
intent.getStringExtra(SmsContext.ReceiveSms.CONTENT);  
        }  
    }  
}
```

删除短信

用户可以使用 **SmsManagerApi.removeSmsById** 和 **SmsManagerApi.removeSmsByType** 接口删除短信。接口定义如下：

```
public static int removeSmsById(long smsId);
```



根据 **smsId** 删除短信;

参数: **smsId** 短信 ID;

返回值: **int** 小于 0: 出错; 大于等于 0: 删除个数;

返回为 0 表示数据库中无此短信,因此大于等于 0 均可以认为删除成功。

```
public static int removeSmsByType(int removeType);
```

根据 **removeType** 删除短信;

参数 **removeType**: 0 - 删除接收的信息, 1 - 删除发送的信息, 2 - 删除草稿箱的信息;

返回值: **int** 小于 0: 出错, 大于等于 0: 删除的个数;

返回为 0 表示数据库中无此类短信, 因此大于等于 0 均可认为删除成功。

声音通道 **AudioRouteApi**

支持的产品: **GXV3370**、**GXV3380**、**GXV3350**、**WP820**

使用 **AudioRouteApi**, 应用程序可以切换或检查声音通道。

切换声音通道

使用接口 **AudioRouteApi.switchVoiceToXXX** 可以切换声音通道。请确保切换到通道是产品支持的。否则, 它将无法切换。

```
AudioRouteApi.switchVoiceToSpeaker();  
AudioRouteApi.switchVoiceToHandset();  
AudioRouteApi.switchVoiceToRJ9Headset();  
AudioRouteApi.switchVoiceToBlueToothHeadset();  
AudioRouteApi.switchVoiceToHdmi();  
AudioRouteApi.switchVoiceToEarphone();  
AudioRouteApi.switchVoiceToUsbHeadset();
```

检查当前声音通道

使用接口 **AudioRouteApi.isVoiceOnXXX** 可以检查当前是否有打开的声音通道。

```
AudioRouteApi.isVoiceOnSpeaker();  
AudioRouteApi.isVoiceOnHandset();  
AudioRouteApi.isVoiceOnRJ9Headset();  
AudioRouteApi.isVoiceOnBlueToothHeadset();  
AudioRouteApi.isVoiceOnHdmi();  
AudioRouteApi.isVoiceOnEarphone();  
AudioRouteApi.isVoiceOnUsbHeadset();
```



EHS 耳机

- **setEhsHookStatus:** 控制 EHS 耳机摘/挂机状态。
- **isEhsOffHook:** 检查 EHS 耳机是否摘机。
- **isEhsHeadsetConnected:** 检查 EHS 耳机是否连接。



注意:

1. 当 EHS 耳机处于挂机状态，即使声音通道在 EHS 耳机，EHS 耳机也不会播放声音。
2. 调用 **setEhsHookStatus** 时，必须保证声音通道在 EHS 耳机，即 **isVoiceOnRJ9Headset()** 的结果必须为 **true**，请在调用 **setEhsHookStatus** 后 500ms 内不要切换声音通道。

总以扬声器振铃

如果在 Web 上配置 WebUI→电话设置→通话设置→“总以扬声器振铃”，即表示无论当前音频在哪个通道，来电时，扬声器都会播放来电铃声。

为了实现上述功能，第三方自定义 Phone 应用需要通过接口 **switchCurrentVoiceWithSpeaker** 来实现，同时结束铃声的时候需要调用接口 **switchCurrentVoiceWithoutSpeaker**。

以下是有关收到一个来电后的简单例子：

```
private class MyCallStatusListener extends CallStatusListener {
    @Override
    public void onLineStatusChanged(int notifyType, BaseLine baseLine,
    List<BaseLine> list) {
        final BaseLine line = baseLine;
        if (line.getStatus() == 2) { // 2 means ringing
            if (CallSettingApi.isAlwaysRingSpeaker()) {
                AudioRouteApi.switchCurrentVoiceWithSpeaker();
                startTone(); // start ringing
            }
        } else {
            if (mPrevStatus == 2) {
                stopTone(); // stop ringing
                AudioRouteApi.switchCurrentVoiceWithoutSpeaker();
            }
        }
        mPrevStatus = line.getStatus();
    }
}
```



```
}
```

通话设置 **CallSettingApi**

支持的产品: GXV3370、GXV3380、GXV3350、WP820

应用通过使用 **CallSettingApi** 的接口, 可以获取到一些通话相关的设置。

DeviceApi

支持的产品: GXV3370、GXV3380、GXV3350、WP820

DeviceApi 提供设备管理和信息相关的 SDK。

获取系统信息的接口如下:

```
SystemInfo systemInfo = DeviceApi.getSystemInfo();  
String productBaseName = systemInfo.getProductBase();  
String systemVersion = systemInfo.getSystemVersion();
```

CallToneApi

支持的产品: GXV3370、GXV3380、GXV3350、WP820

CallToneApi 提供铃声播放的相关接口

```
// 检查是否有 tone 音在播放  
public static boolean isAnyToneInPlay();  
  
// 检查铃声是否在播放  
public static boolean isRingToneInPlay();  
  
// 检查拨号音是否在播放  
public static boolean isDialToneInPlay();  
  
// 检查 DTMF 音是否在播放  
public static boolean isDtmfToneInPlay();  
  
// 停止播放所有 tone 音  
public static void stopAllTone();  
  
// 停止播放所有铃声  
public static void stopRingtone();
```



```
// 开始播放 DTMF 音
public static void startDtmfTone(int keyValue);

// 停止播放 DTMF 音
public static void stopDtmfTone(int keyValue);

// 开始播放呼叫等待音
public static void startCallWaitingTone();

// 停止播放呼叫等待音
public static void stopCallWaitingTone();

// 开始播放回铃音
public static void startRingbackTone();

// 停止播放回铃音
public static void stopRingbackTone();

// 开始播放忙音
public static void startBusyTone();

// 停止播放忙音
public static void stopBusyTone();

// 开始播放拨号音
public static void startDialTone();

// 停止播放拨号音
public static void stopDialTone();

// 开始播放备选拨号音
public static void startSecondDialTone();

// 停止播放备选拨号音
public static void stopSecondDialTone();

// 开始播放确认铃音
public static void startConfirmTone();
```



```
// 停止播放确认铃声
public static void stopConfirmTone();

// 开始播放续订音
public static void startReorderTone();

// 停止播放续订音
public static void stopReorderTone();

// 根据 lineId 播放匹配的铃声
public static void startRingtone(int lineId);

// 开始播放系统铃声
public static void playSystemRingtone();

// DND 模式下开始播放 DND 音
public static void playDndTone();

// 有未读语音邮件时开始播放拨号音
public static void startVMDialTone();

// 有未读语音邮件时停止播放拨号音
public static void stopVMDialTone();

// 开始播放自动应答音
public static void startAutoAnswerTone();

// 停止播放自动应答音
public static void stopAutoAnswerTone();

// 获取当前正在播放的 tone 音类型
public static int getCurToneType();
```

Gs Core SDK

这些 SDK 可以不需要初始化 **ApiClient** 就能使用。这些 SDK 一般命名成 **XXXManager**。例如：**GsDevStateManager**。



设备管理

支持的产品: GXV3370、GXV3380、GXV3350、WP820

获取设备连接状态

```
// 3.5mm 耳机是否连着
GsDevStateManager.instance().isDeviceConnected(GsDevStateManager.NAME_EARPHONE);

// ehs 耳机设备是否连着
GsDevStateManager.instance().isDeviceConnected(GsDevStateManager.NAME_EHS);

// hdmi 设备是否连着, 一般指的是 hdmi 输出设备
GsDevStateManager.instance().isDeviceConnected(GsDevStateManager.NAME_HDMI);

// hdmi 输入设备是否连着
GsDevStateManager.instance().isDeviceConnected(GsDevStateManager.NAME_HDMI_IN);

// usb 耳机设备是否连着
GsDevStateManager.instance().isDeviceConnected(GsDevStateManager.NAME_USB_HEADSET);
```

重启设备

```
GsDevStateManager.instance().reboot();
```

DND 设置

```
DndManager.instance().setDndOn(); //开启全局 DND 功能
DndManager.instance().setDndOff(); //关闭全局 DND 功能
DndManager.instance().isDndOn(); //查询当前 DND 在功能是否为开启状态
```

获取设备 MAC 地址

```
GsDevStateManager.instance().getDeviceMac();
```





联系人

支持的产品: GXV3370、GXV3380、GXV3350、WP820

源生的 **android.content.ContentResolver** 已经提供了联系人的增删查改 API。有关这些 API 信息, 请参阅以下链接:

<https://developer.android.google.cn/reference/android/content/ContentProvider>

有关联系人数据库的更多详细信息, 请参阅:

<https://developer.android.google.cn/reference/android/provider/ContactsContract>

使用 **android.content.ContentResolver** 进行联系人增删查改, 可以满足大多数需求。

在增删查改联系人之前, 请务必阅读以下内容, 了解适用于 Grandstream 支持产品的更改。数据库表中使用了一些新列和一些保留列来存储重要信息。这些数据库列名和值定义在

com.gs.contacts.context.ContactsContext 中。有关更多详细信息, 请参阅 **gsApiExternal-zh**。

数据库 ContactsContract.Data 的变更

联系人数据库表 ContactsContract.Data 更改如下:

表 4: ContactsContract.Data 变更字段

Key	Type	Description
ContactsContext.ContactsItem#ITEM_ACCOUNT_ID	整数(long)	特殊用途

数据库 ContactsContract.RawContacts 的变更

联系人数据库表 ContactsContract.RawContacts 更改如下:

表 5: ContactsContract.RawContacts 变更字段

列字段	类型	Description
ContactsContext.ContactsItem#ITEM_SORT_KEY_T9	文本	新增列
ContactsContext.ContactsItem#ITEM_SORT_KEY_T9_FL	文本	新增列



ContactsContract.Contacts#CONTENT_FILTER_URI 字段变更

我们在源生的基础上添加了模糊搜索。通过模糊搜索，用户可以通过诸如'number=123'之类的条件进行查询，以获得包括在诸如'123456'或'0123'之类的数字中的 123 而不仅仅是'123'的所有匹配结果。用户可以通过设置 **ContactsContext.SearchSnippets.FUZZY_KEY** 开启模糊搜索，如下所示：

```
Uri.Builder builder = ContactsContract.Contacts.CONTENT_FILTER_URI.buildUpon();
builder.appendPath(query);
builder.appendQueryParameter(ContactsContract.DIRECTORY_PARAM_KEY,
String.valueOf(directoryId));
builder.appendQueryParameter(ContactsContract.SearchSnippets.DEFERRED_SNIPPETING_KEY, "1");
builder.appendQueryParameter(com.gs.contacts.context.ContactsContext.SearchSnippets.FUZZY_KEY, "1");
Uri uri = builder.build();
Cursor cursor = mContext.getContentResolver().query(uri, null, null, null, null);
```

新增联系人

以下是有关如何添加联系人的例子：

```
public void addContact(String name, String number, long accountId) {
    <ContentProviderOperation> ops

        = new ArrayList<ContentProviderOperation>();

    int rawContactInsertIndex = ops.size();
    ops.add(ContentProviderOperation
        .newInsert(ContactsContract.RawContacts.CONTENT_URI)
        .withValue(ContactsContract.RawContacts.ACCOUNT_TYPE, null)
        .withValue(ContactsContract.RawContacts.ACCOUNT_NAME, null)
        .withYieldAllowed(true).build());
    ops.add(ContentProviderOperation
        .newInsert(ContactsContract.Data.CONTENT_URI)
        .withValue(ContactsContext.ContactsItem.ITEM_ACCOUNT_ID, accountId)
        .withValueBackReference(ContactsContract.Data.RAW_CONTACT_ID,
            rawContactInsertIndex)
        .withValue(ContactsContract.Data.MIMETYPE,
ContactsContract.CommonDataKinds.StructuredName.CONTENT_ITEM_TYPE)
        .withValue(ContactsContract.CommonDataKinds.StructuredName
            .DISPLAY_NAME, name)
```



```
        .withYieldAllowed(true).build());
ops.add(ContentProviderOperation
        .newInsert(android.provider.ContactsContract.Data.CONTENT_URI)
        .withValueBackReference(ContactsContract.CommonDataKinds.Phone
            .RAW_CONTACT_ID, rawContactInsertIndex)
        .withValue(ContactsContract.CommonDataKinds.Phone.MIMETYPE,
            ContactsContract.CommonDataKinds.Phone.CONTENT_ITEM_TYPE)
        .withValue(ContactsContract.CommonDataKinds.Phone.TYPE,
            ContactsContract.CommonDataKinds.Phone.TYPE_MOBILE)
        .withValue(ContactsContract.CommonDataKinds.Phone.NUMBER, number)
        .withYieldAllowed(true).build());
try {
    contentResolver.applyBatch(ContactsContract.AUTHORITY, ops);
} catch (Exception e) {
    e.printStackTrace();
}
}
```

更改联系人

以下是有关如何修改联系人的例子：

```
public void updateContact(long rawContactId, String name) {
    ArrayList<ContentProviderOperation> ops
        = new ArrayList<ContentProviderOperation>();
ops.add(ContentProviderOperation
        .newUpdate(ContactsContract.Data.CONTENT_URI)
        .withSelection(ContactsContract.Data.RAW_CONTACT_ID+"=?",
            new String[]{rawContactId+""})
        .withValue(ContactsContract.CommonDataKinds.StructuredName
            .DISPLAY_NAME, name)
        .build());
try {
    contentResolver.applyBatch(ContactsContract.AUTHORITY, ops);
} catch (Exception e) {
    e.printStackTrace();
}
}
```



查询联系人

以下是有关如何查询联系人的例子：

```
public void getContacts() {
    String[] projection = new String[]{
        ContactsContext.ContactsItem.ITEM_CONTACT_ID_IN_DATA,
        ContactsContext.ContactsItem.ITEM_ACCOUNT_ID,
        ContactsContext.ContactsItem.ITEM_PHONE_NUMBER,
        ContactsContext.ContactsItem.ITEM_DISPLAY_NAME
    };
    Cursor cursor = contentResolver
        .query(ContactsContract.CommonDataKinds.Phone.CONTENT_URI,
            projection, null, null, null);
    if (cursor != null) {
        while (cursor.moveToNext()) {
            long contactId = cursor.getLong(
                cursor.getColumnIndex(ContactsContext.ContactsItem
                    .ITEM_CONTACT_ID_IN_DATA));
            long accountId = cursor.getInt(cursor.getColumnIndex(
                ContactsContext.ContactsItem.ITEM_ACCOUNT_ID));
            String number = cursor.getString(cursor.getColumnIndex(
                ContactsContext.ContactsItem.ITEM_PHONE_NUMBER));
            String name = cursor.getString(cursor.getColumnIndex(
                ContactsContext.ContactsItem.ITEM_DISPLAY_NAME));
        }
        cursor.close();
    }
}
```

删除联系人

以下是有关如何删除联系人的例子：

```
public void deleteContactById(long contactId) {
    ArrayList<ContentProviderOperation> ops
        = new ArrayList<ContentProviderOperation>();
    ops.add(ContentProviderOperation
        .newDelete(ContactsContract.RawContacts.CONTENT_URI)
```



```
        .withSelection(ContactsContract.RawContacts.CONTACT_ID
            + "=" + contactId, null)
        .build());
ops.add(ContentProviderOperation
    .newDelete(ContactsContract.Data.CONTENT_URI)
    .withSelection(ContactsContract.Data.CONTACT_ID
        + "=" + contactId, null)
    .build());
try {
    contentResolver.applyBatch(ContactsContract.AUTHORITY, ops);
} catch (Exception e) {
    e.printStackTrace();
}
}
```



通话记录

支持的产品: GXV3370、GXV3380、GXV3350、WP820

源生的 **android.content.ContentResolver** 已经提供通话记录的增删查改 API。有关这些 API 信息, 请参阅以下链接:

<https://developer.android.google.cn/reference/android/content/ContentProvider>

有关通话记录数据库的更多详细信息, 请参阅:

<https://developer.android.google.cn/reference/android/provider/CallLog>

大多数情况下, 在使用 BaseCallApi 进行拨号之后, 通话记录会自动添加或变化。Gs JARS 不提供额外的 API 单独用于通话记录功能。使用 **android.content.ContentResolver** 来进行增删查改可以满足大多数需求了。

在增删查改通话记录之前, 请务必阅读以下内容, 了解 Grandstream 支持产品的更改。数据库表中使用了一些新列和一些保留列来存储重要信息。这些数据库列名和值定义在

com.gs.calllog.context.CallLogContext 中。有关更多详细信息, 请参阅 **gsApiExternal-zh**。

CallLog.Calls 表的变更内容

通话记录数据库表 CallLog.Calls 变化如下:

表 6: CallLog.Calls 变化

Key	Type	Description
CallLogContext.CallLogItem#ITEM_ACCOUNT	整数(long)	新增列
CallLogContext.CallLogItem#ITEM_CALL_MODE	整数(int)	新增列
CallLogContext.CallLogItem#ITEM_MEDIA_TYPE	整数(int)	新增列
CallLogContext.CallLogItem#ITEM_NUMBER_ORIGINAL	文本	新增列
CallLogContext.CallLogItem#ITEM_CONTACT_CACHE_NAME	文本	新增列
CallLogContext.CallLogItem#ITEM_END_TIME	文本	新增列
CallLogContext.CallLogItem#ITEM_IS_IN_CONFERENCE	整数(int)	新增列
CallLogContext.CallLogItem#ITEM_GROUP_ID	整数(long)	新增列



**注意:**

不推荐使用定义在 `CallLog.Calls` 的 `public static` 方法，否则可能会得到非预期的结果。

新增通话记录

由于通话记录与通话密切相关，我们强烈建议您使用 **BaseCallApi** 拨打电话并默认保存通话记录。用户可以使用 **ContentResolver** 接口来添加通话记录。

以下是有关如何添加通话记录的例子：

```
public void addCalllog(String name,String number,long accountId) {
    ArrayList<ContentProviderOperation> ops
        = new ArrayList<ContentProviderOperation>();
    ops.add(ContentProviderOperation.newInsert(CallLogContext.CALL_LOG_URI)
        .withValue(CallLogContext.CallLogItem.ITEM_ACCOUNT, accountId)
        .withValue(CallLogContext.CallLogItem.ITEM_CACHE_NAME, name)
        .withValue(CallLogContext.CallLogItem.ITEM_CALL_MODE,
            PhoneContext.CallMode.SIP_CALL)
        .withValue(CallLogContext.CallLogItem.ITEM_START_TIME,
            System.currentTimeMillis())
        .withValue(CallLogContext.CallLogItem.ITEM_IS_IN_CONFERENCE, false)
        .withValue(CallLogContext.CallLogItem.ITEM_NUMBER_ORIGINAL, number)
        .withValue(CallLogContext.CallLogItem.ITEM_NUMBER, number)
        .withValue(CallLogContext.CallLogItem.ITEM_CALL_TYPE,
            CallLogContext.CallLogType.TYPE_MISSED)
        .withValue(CallLogContext.CallLogItem.ITEM_DURATION, 0)
        .withValue(CallLogContext.CallLogItem.ITEM_MEDIA_TYPE,
            CallLogContext.MediaType.AUDIO)
        .build());
    try {
        contentResolver.applyBatch(CallLog.AUTHORITY, ops);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```



查询通话记录

用户可以使用 **ContentResolver** 接口来查询通话记录。

以下是有关如何查询通话记录的例子：

```
public void getCallLogs() {
    Cursor cursor = contentResolver.query(CallLogContext.CALL_LOG_URI,
        null,null,null,null);
    StringBuilder res = new StringBuilder("");
    if(cursor!= null){
        if(cursor.moveToFirst()){
            do{
                long calllogId = cursor.getLong(
                    cursor.getColumnIndex(CallLogContext.CallLogItem.ITEM_ID));
                String cacheName = cursor.getString(cursor.
                    getColumnIndex(CallLogContext.CallLogItem.ITEM_CACHE_NAME));
                int mediaType = cursor.getInt(cursor.
                    getColumnIndex(CallLogContext.CallLogItem.ITEM_MEDIA_TYPE));
            }while (cursor.moveToNext());
        }
        cursor.close();
    }
}
```

更改通话记录

用户可以使用 **ContentResolver** 接口来更改通话记录。

以下是有关如何更改通话记录的例子：

```
public void updateCalllogById(long callId) {
    ArrayList<ContentProviderOperation> ops
        = new ArrayList<ContentProviderOperation>();
    ops.add(ContentProviderOperation
        .newUpdate(CallLogContext.CALL_LOG_URI)
        .withSelection(CallLogContext.CallLogItem.ITEM_ID+"="+callId, null)
        .withValue(CallLogContext.CallLogItem.ITEM_DURATION,1000)
```



```
        .build());  
    try {  
        contentResolver.applyBatch(CallLog.AUTHORITY, ops);  
    } catch (Exception e) {  
        e.printStackTrace();  
    }  
}
```

删除通话记录

用户可以使用 **ContentResolver** 接口来删除通话记录。

以下是有关如何删除通话记录的例子：

```
public void deleteCalllogById(long callId) {  
    ArrayList<ContentProviderOperation> ops  
        = new ArrayList<ContentProviderOperation>();  
    ops.add(ContentProviderOperation  
        .newDelete(CallLogContext.CALL_LOG_URI)  
        .withSelection(CallLogContext.CallLogItem.ITEM_ID+"="+callId, null)  
        .build());  
    try {  
        contentResolver.applyBatch(CallLog.AUTHORITY, ops);  
    } catch (Exception e) {  
        e.printStackTrace();  
    }  
}
```



LED

支持的产品: GXV3370、GXV3380、GXV3350

在 Android 标准接口中, 通知是唯一使用 LED 的方法。

以下是有关如何使用 LED 通知的例子:

```
int color = 0xFF00FF00;
int onMs = 1000;
int offMs = 1000;
NotificationManager manager = (NotificationManager)
getSystemService(NOTIFICATION_SERVICE);
Notification notification = null;
notification = new Notification.Builder(this)
    .setContentTitle("This is content title")
    .setContentText("This is content text")
    .setWhen(System.currentTimeMillis())
    .setSmallIcon(R.mipmap.ic_launcher)
    .setLargeIcon(BitmapFactory.decodeResource(getResources(),
R.mipmap.ic_launcher))
    .setLights(color, onMs, offMs)
    .build();
manager.notify(1, notification);
```

请在 GXV3370 上找到支持的 LED 颜色和频率列表。如果设备不支持颜色或频率设置, 设备将自动选择默认颜色或最近频率。下表列出了 GXV3370 / GXV3380 / GXV3350 LED 支持的颜色和频率。

表 7: LED 颜色和频率

Product	GXV3370	GXV3380	GXV3350
Red LED	On/Off	On/Off	On/Off
Green LED	On/Off	On/Off	On/Off
Blue LED	N/A	N/A	N/A
常亮频率	onMs: 1000, offMs: 0	onMs: 1000, offMs: 0	onMs: 1000, offMs: 0
慢速闪烁频率	onMs: 1000, offMs: 3000	onMs: 1000, offMs: 3000	onMs: 1000, offMs: 3000
快速闪烁频率	onMs: 1000, offMs: 1000	onMs: 1000, offMs: 1000	onMs: 1000, offMs: 1000



GsLightsManager

除了通过 Notification 控制 LED 之外，还可以通过 GsLightsManager 对 LED 进行自由控制。相关接口定义如下：

```
public void openCustom(int color, int shineType);  
public void closeCustom();
```

其中，参数解释如下：

shineType: 用于控制闪烁类型。支持快闪、慢闪、常亮。

color: 打开的 LED 颜色。根据设备不同，可以支持红色、绿色、蓝色。



可编程键

支持的产品: GXV3350

可编程键 MpkApi 用于监听扩展板点击事件等。

监控扩展板点击事件

用户可以使用 **MpkApi.addStatusListener** 和 **MpkApi.removeStatusListener** 来监听扩展板点击事件。

以下是有关如何监控扩展板点击事件的例子:

启动扩展板点击事件监控

```
private void initExtClickListener() {  
    mMpkStatusListener = new MpkStatusListener();  
    MpkApi.addStatusListener ("ExtEvent",  
        mMpkStatusListener.callback,  
        MpkContext.ListenType.EXT_EVENT);  
}
```

关闭扩展板点击事件监控

```
private void removeExtClickListener() {  
    MpkApi.removeStatusListener (mMpkStatusListener.callback);  
    mMpkStatusListener.destroy();  
    mMpkStatusListener = null;  
}
```

MpkStatusListener 实现扩展板点击事件监控

```
private class MyMpkStatusListener extends MpkStatusListener {  
    @Override  
    public void onExtItemClick(MpkEntity entity) {  
    }  
};  
  
MpkStatusListener mMpkStatusListener = null;
```



WP820 的特殊接口

安全告警和 KEY Code Definitions 是 WP820 产品支持的特殊接口。

安全告警接口使用说明

安全告警主要通过 Android 广播的形式发送给应用。

安全告警广播定义

安全告警广播 Action 定义

```
public static final String SAFE_MONITORING_STATE_CHANGED_ACTION =
"android.sensor.monitoring.STATE_CHANGED";
```

Action Extra key 定义

```
public static final String EXTRA_STATE = "android.intent.extra.STATE";
```

安全告警检测类型：奔跑

```
public static final int SENSOR_MONITORING_TYPE_RUN = 1;
```

安全告警检测类型：倾斜

```
public static final int SENSOR_MONITORING_TYPE_TILT = 2;
```

安全告警检测类型：静止

```
public static final int SENSOR_MONITORING_TYPE_NOMOVE = 3;
```

安全告警检测类型：多功能按键手动触发

```
public static final int SENSOR_MONITORING_TYPE_PANIC = 4;
```

开启安全告警

从 WP820 LCD 菜单上进入设置→高级设置→安全/告警→安全监控设置，开启“监控”以及需要检测的项。只有开启“监控”和相关项之后，告警广播才能发送。

自定义处理安全告警广播

以下是有关安全告警广播的例子：

(1) 注册广播

```
<receiver android:name=".SafeMonitorReceiver">
    <intent-filter>
        <action android:name="android.sensor.monitoring.STATE_CHANGED"/>
    </intent-filter>
</receiver>
```

(2) 接收广播

```
public class SafeMonitorReceiver extends BroadcastReceiver {
```



```
private static final String TAG= "SafeMonitorReceiver";
@Override
public void onReceive(Context context, Intent intent) {
    String action = intent.getAction();
    Log.i(TAG, "onReceive:"+action);
    if("android.sensor.monitoring.STATE_CHANGED".equals(action)){
        int type = intent.getIntExtra("sensor_monitoring", 0);
        if(type==1){
            Log.i(TAG, "type run");
        }else if(type==2){
            Log.i(TAG, "type tilt");
        }else if(type==3){
            Log.i(TAG, "type not move");
        }else if(type==4){
            Log.i(TAG, "type panic");
        }
    }
}
```

KEY Code 定义

左边按键

```
public static final int KEY_CODE_LEFT = 1803;
```

中间按键

```
public static final int KEY_CODE_MID = 1804;
```

右边按键

```
public static final int KEY_CODE_RIGHT = 1805;
```



ADB 使用说明

使用 Android Studio 或其他 IDE 进行开发应用程序时，请看这份使用手册。

不同于通过 USB 连接到 Android 设备进行开发，在 Grandstream Android 设备上开发应用程序需将设备连接到网络。

打开设备开发者模式

进入话机 LCD 菜单→设置→系统安全，打开开发者模式。

adb 连接

使用命令 'adb connect IP' 连接设备。例如：

```
adb connect 192.168.100.1
```

设备允许调试

成功'adb connect'连接后，设备会显示一个对话框以运行调试。选择“确定”以继续。建议选中“始终允许来自此计算机”，以便下次将设备连接到同一台计算机时不再显示此对话框。

检查 adb 连接状态

使用命令 'adb devices' 检查 adb 连接状态。如果显示以下内容，则表示连接成功：

```
List of devices attached
```

```
192.168.100.1:5555    device
```

adb 断开连接

使用命令 'adb disconnect' 断开与所有设备的连接。



一台设备同时只能连接一台电脑。



API Demo

在 SDK 包中，您可以使用名为 **ApiDemo** 的 SDK API 找到开发的应用程序。

