2025

AJVoIP

Android Java VoIP SDK

The Mizu Android Java VoIP SDK (AJVoIP) is a compact SIP client library for Android compatible with all common VoIP devices, servers and softphones.

-
Mizutech
6/3/2025

Contents

About

The Android Java VoIP SDK (AJVoIP) is a SIP client library for the Android platform. Since it is based on the open standard Session Initiation Protocol, it can inter-operate with any other SIP-based device (servers and clients).

AJVoIP is an easy to use compact Android SIP client library consisting of a single library file which have to be added to your project (one single aar or jar file) containing a simple high-level API (one single class).

With the Android SIP SDK you have a compact but full featured SIP/media stack, easy to integrate with any Android application.
It can be used to develop a custom Android SIP client application or to add VoIP call capabilities into any existing Android application.

Package content

The followings can be found in the AJVoIP package:

- The library itself:
  - **AJVoIP.aar** (Android Archive file to be used with Android Studio or in any other IDE/tool with .aar library support)
  - AJVoIP.jar (optional jar file to be used in any other development environment which doesn't support .aar files, such as old Eclipse or Delphi)
  - ajvoip-sources.jar: the library API interface source/javadoc for better help in your IDE (optional)
- Documentation:
  - SDK description: software home-page
  - Full documentation: this document
  - JavaDoc: download
- Sample project:
  - Test project (a working example) to be downloaded from here
- License:
  - AJVoIP comes with life-time license allowing commercial usage for your company (limited features and number of users with the Basic license, al features and unlimited users and calls with the Advanced and Gold license)
- Support:
  - Support is also included with your AJVoIP license for no extra cost. Contact us with any question/issue/bug report and we are here to help.
  - Upgrades: AJVoIP is actively maintained with usually one new internal stable release per month and a major new version per year including improvements, bug fixes and new features in each new release. Upgrades are always backward and forward compatible with no code changes required in your project.

Requirements

- **Programming language:** Java, Kotlin, Scala or any other which can consume a aar/jar and can emit android apk such as Xamarin, Corona, Phonegap/Cordova, C++, Python, Groovy, Flutter, Xamarin, React, Ionic and others.
- **IDE**: For development you can use any environment which can produce android executables. The most popular options are Android Studio and Eclipse. You can use any operating system to work on your project: Windows, MAC, Linux.
- **SIP**: a SIP account at any VoIP service provider or your own IP-PBX/Softswitch/SIP sever. Can be also used without registration for peer to peer SIP calls.
- **Network**: All networks are supported with connection speed above 12 kbits including Wi-Fi and mobile GPRS, EDGE, 3G, 4G, LTE, 5G. VPN connections are also supported.
- **Size**: below 3 MB
- **RAM**: around 35 MB while in call. (This means that you can run the SIP engine also on low-end devices with 256 MB RAM or less)
- **CPU architecture**: any. Most of the library is pure Java. *To improve performance, some audio processing and codec related functionalities are implemented also as native ABI modules for all the popular platforms (arm, x86 and mips, including 64bit versions), but these modules are optional (will be loaded only on compatible platforms although this means that it will work on most devices)*
- **CPU speed**: any (The library will auto detect the CPU performance and it might disable some features on old/slow CPU's, such as G.729, wideband and AEC which means that you can run it even on low-end 100 MHz ARM processors)
- **Android versions**: Android 4 (released in 2011) – Android 16+ (2025). Compatible with all devices since Android 4 and forward compatible with upcoming Android releases (new releases might add further optimizations for new Android versions)
- **API level**: All Android API/SDK versions are supported (AJVoIP is optimized to old API levels but also takes full advantage of the features found in latest versions if you are running it on a compatible OS)
  - Minimum (minSdkVersion): 14 (released in 2011 with Android 4.0 Ice Cream Sandwich), which means compatibility also with ancient phones covering 100% of the market.
    Note: On request, we can ship also build with minSdkVersion 9 (this might be necessary only if you wish to target some specific/outdated/ancient hardware)
  - Target (targetSdkVersion): 14 or higher (you should set this as high as possible for your project environment. Current recommended is 35)
    Note: due to Google Play regulations you must always target latest android versions as old API levels cannot be submitted.
  - Compile: The compileSdk should be set to 35 or higher (you should set it to highest possible by your project environment)
  - Maximum: 36+. API level 36 introduced with Android 16 in 2025. The SDK is continuously tested and optimized to latest API levels as released by Google while also keeping full backward compatibility with old API levels. You can expect to work it also in all future versions with no modifications required and we also add optimizations to latest API levels when new versions are released if there is any new Android feature that can be exploited by the VoIP SDK.
- **Required permissions:**

You can set these in your AndroidManifest.xml as uses-permission elements:
Example: <uses-permission android:name="android.permission.RECORD_AUDIO"/>

Here are the required and optional permissions:

- Dangerous/Mandatory (your app must have permission for this):
  - RECORD_AUDIO (used for the audio input stream)
- Dangerous/Optional (permission have to be asked only if related features are needed):
  - CAMERA (used only if you need video calls)
  - POST_NOTIFICATIONS (not required for AJVoIP, but might be needed for your app if you wish to post notifications)
  - READ_CONTACTS and WRITE_CONTACTS (not required for AJVoIP at all, but might be needed for your app if you wish to manage contacts)
  - READ_PHONE_STATE (only if the rejectonphonebusy parameter is set to 0 or 2 or if you need more accurate call state detection)
- Normal (permissions are granted automatically for these, just add them into your manifest.xml):
  - INTERNET (mandatory)
  - ACCESS_NETWORK_STATE (optional but highly recommended)
  - CHANGE_NETWORK_STATE (optional)
  - MODIFY_AUDIO_SETTINGS (optional)
  - ACCESS_WIFI_STATE (optional)
  - CHANGE_WIFI_STATE (optional)
  - BLUETOOTH (optional for API level < 31)
  - BLUETOOTH_CONNECT (optional for API level >= 31)
  - DISABLE_KEYGUARD (optional)
  - VIBRATE (optional)
  - FOREGROUND_SERVICE (optional since API level 27 if you wish to launch the sipstack from a foreground service)
  - REQUEST_IGNORE_BATTERY_OPTIMIZATIONS (optional for API level 26 or above running the sipstack as a service)
  - WAKE_LOCK (optional but highly recommended)
  - SCHEDULE_EXACT_ALARM (optional for API level < 32; might help to keep the app service running on some devices)
  - RECEIVE_BOOT_COMPLETED (needed only if you wish to run it as a service and auto-start)

Note: there is no need to ask for permissions at runtime if your application target API level 22 or below outside the Play Store.
More details about permissions can be found here.

- **Required android features:**

You can set these in your AndroidManifest.xml as uses-feature elements:
The only important feature is the android.hardware.microphone (but this is also not enforced by default by the library because you might use it for other purposes when no media recording is required for calls).
For a usual SIP client application you should add the following line to your manifest:

```
<uses-feature android:name="android.hardware.microphone" android:required="true" />
```

The RECORD_AUDIO permission must be also asked at runtime as described here. AJVoIP will try to ask for this on first audio device access, but this might fail outside your activity and it is much better if you ask it explicitly from your app at a more appropriate time (for example at startup or before the first call).
Other features that might be used by the library (thus might be required by your application) are the followings:

- android.hardware.audio.low_latency
- android.hardware.audio.output
- android.hardware.audio.pro
- android.hardware.telephony
- android.hardware.bluetooth
- android.hardware.bluetooth_le
- android.hardware.wifi
- android.hardware.sensor.proximity
- android.hardware.camera
- android.hardware.camera.front
- android.hardware.camera.any
- android.hardware.camera.autofocus
- android.software.sip
- android.software.sip.voip
- android.software.webview

None of these are crucial so you might add them to your manifest with required flag set to false or don't add at all if not mandatory for your use-case.
Example: <uses-feature android:name="android.hardware.audio.low_latency" android:required="false" />

Usage

You can use the AJVoIP library in any Android project. Just add the library to your project and call its public API functions.

Instructions

Step-by-step instruction for using AJVoIP:

1. Download AJVoIP from here (this is the demo version) or from the link provided by Mizutech (paid versions) and unzip.
   You might load the sample project first and check its MainActivity.java file for a simple but working usage example.

2. Add the SIP library to your project
   You can use either the AJVoIP.aar file or the AJVoIP.jar file. If you are using Android Studio then we recommend the aar file. Otherwise the jar file.

- For Android Studio:
  1) Copy the AJVoIP copy the AJVoIP.aar into your \app\libs\ folder
  2) Modify the libs folder import in your build.gradle to include also .aar files (not only .jar files like the default)

```
dependencies {
        implementation fileTree(dir: 'libs', include: ['*.jar','*.aar'])
        ...
```

  3) Optionally copy also the ajvoip-sources.jar file for better inline documentation in your IDE. It might need to be selected from "Choose Sources" if not recognized automatically.
- For Android Studio you can also add it explicitly to your project this way:
  1) Click File > New > New Module.
  2) Click Import .JAR/.AAR Package then click Next.
  3) Enter the location of the AJVoIP.aar file then click Finish.
  4) Make sure the library is listed at the top of your settings.gradle file:

```
include ':app', ':AJVoIP'
```

  5) Open the app module's build.gradle file and add a new line to the dependencies block:

```
dependencies {
  ...
  implementation project(":AJVoIP")
}
```

  6) Click Sync Project with Gradle Files.
- For Android Studio you can also import AJVoIP in the following way:
  1) File -> Project Structure -> Dependencies -> select your app module and add to declared dependencies with the + button
  2) More details here
- If you are using Eclipse then follow these steps:
  1) Create a folder called libs in your project's root folder
  2) Copy the AJVoIP.jar file to the libs folder
  3) Now right click on the AJVoIP.jar file and then select Build Path > Add to Build Path, which will create a folder called 'Referenced Libraries' within your project
  4) You can read more details here
- If you are using other IDE or environment: Just import the AJVoIP.aar (or the AJVoIP.jar of your IDE doesn't support .aar libraries) into your project and use its public class.

- More details about the project configuration can be found here

3. Instantiate a SipStack object

Add the following code to the location from where do you wish to start the SIP client:

```
import com.mizuvoip.jvoip.*; //import the SipStack class from AJVoIP
SipStack mysipclient = null; //declare the SIP stack object variable somewhere in your application
mysipclient = new SipStack(); //create the SIP Stack instance.
mysipclient.Init(context);  //initialize the SIP stack
//The context parameter is your application Context which can be obtained from any activity or as described here
```

4. Call the SetParameter (or the SetParameters) function to pass any settings

```
mysipclient.SetParameter("loglevel","5");  //set loglevel
mysipclient.SetParameter("serveraddress","voip.mizu-voip.com");  //set your voip server domain or IP:port
mysipclient.SetParameter("username","sdktest2");  //set SIP username
mysipclient.SetParameter("password","sdktest2"); //set SIP password
//you might set other parameters here such as proxyaddress, autoaccept and many others. See the parameters chapter for the full list of possible parameters.
```

5. Call the Start() to start the sipstack

```
mysipclient.Start();  //this will start the SIP stack internal main thread. It might also auto register, depending on the “register” parameter
//mysipclient.Register();  //if you have set the “register” parameter to 0, then you might need to register explicitly
```

6. Call any of the API functions. This is where the actual work is done. Example:

```
mysipclient.Call(-1, “testivr3”);  //init call to “testivr3”. Possibly behind a “Call” button.
mysipclient.SendChat(-1, “john”, “hi”);  //send “hi” text message to the john SIP user or SIP extension. Possibly behind a “Chat” button.
```

Use any other API (for example to send DTMF, set presence, call transfer, conference or other functions)

7. Handle the notifications.

```
//create a SIPNotification subclass and override the functions (notification events) in which you are interested in:
class MySIPNotificationListener extends SIPNotificationListener
{
  public void onAll(SIPNotification e) {
    Log.v("AJVoIP","Notification received: " + e.toString());
  }
  public void onStatus (SIPNotification.Status e) {
    Log.v("AJVoIP","STATUS notification received: " + e. getStatusText());
```

```
    }
}
//subscribe to notifications (this can be done when you initialize the SipStack):
mysipclient.SetNotificationListener(new MySIPNotificationListener());
```

Adjust your state machine and GUI according the notifications received (the STATUS notification is the most relevant)
See the Notifications chapter and the Javadoc or the ajvoip-source.jar for the details.

8. Fine-tune your application by adjusting any additional parameters that might be needed for your use-case.
Scroll trough the FAQ to find the answer for the most relevant issues and potential features such as call recording or call transfer usage.
Contact us if you run into any issue or with any question

Project configuration

There are no any specific requirements regarding your Android project, IDE or environment. You just need to add the AJVoIP.aar into your Android Studio project (or the AJVoIP.jar if you are using some other IDE) and you are ready to go by importing the API class (import com.mizuvoip.jvoip.*;)

SDK Version

The followings are to be considered for your gradle and manifest file.

The basic requirements are the followings (SDK / api level settings for your app):

- Set the minSdkVersion to 14 or higher (the minimal SDK version your app is prepared to handle. It is recommended to set this to a value to cover most of the phones on the market. Let us know if for some reason you need to go below 14)
- Set the compileSdk to the maximum possible in your development environment (Depending on the SDK version you wish to use to compile your project. AJVoIP should work fine with any SDK)
- Set the targetSdkVersion to at least to the minimum API level allowed in Google Play when your publish your app. (Google Play doesn't allow targeting too old API levels)

The AJVoIP library itself have been built with the following SDK/API settings (at the time when we updated this documentation):

- min SDK: 14 (this gives you a lot of flexibility, allowing your app to target any SDK versions you wish starting with the already ancient v.14)
- target SDK: 35
- compile SDK: 35
- the library might load API level 14+ (including v.35) features at run-time with class-loader or support libraries when it is running on a new devices, taking full advantage of the latest Android platform features.

Dependencies

If your minSdkVersion is 20 or below (prior to Android 5.0), then you should enable multidex as described here.
The library depends also on android support library (this is to allow your project to compile and target any SDK using any API level since 14).
You might need to add it into your gradle config as well:

```
dependencies {
    ...
    implementation 'androidx.legacy:legacy-support-v4:1.0.0'
}
```

Permissions

The only "dangerous" permission required by default is audio recording. For this you should add the following lines to your AndroidManifest.xml:

```
<uses-permission android:name="android.permission.RECORD_AUDIO"></uses-permission>
<uses-feature android:name="android.hardware.microphone" android:required="true" />
```

Above API level 23 the permission request should be handled at runtime as described here and here.
Permissions can be requested from your own code, however the AJVoIP will also check (and ask for permission if needed) for RECORD_AUDIO (for calls) and CAMERA (on video calls).
You must also add the following non-dangerous permission to your AndroidManifest.xml (allowed automatically, no need for explicit permission request):

```
<uses-permission android:name="android.permission.INTERNET"/>
```

All other permissions and features are optional and handled gratefully by the library if not granted. For a full list see the "Required Permissions" section here.
You need to take care of these only if your app requires a specific feature depending on one of these permissions.

More details about permissions can be found here.

Build

For a basic gradle config file, see the build.gradle file in the sample application (inside the "app" subfolder).
You might also check the gradle file of our MizuDroid softphone (this is a full featured SIP softphone based on the AJVoIP library with a long list of features).

For a basic manifest file, see the AndroidManifest.xml in the sample application.

You might also check the manifest file of our MizuDroid softphone used in production.

You might need to slightly modify your project config to match the requirements, however in overall AJVoIP doesn't impose any hard limitations on the required configurations and you should be able to use the library in any environment with any settings if you respect the minimal requirements (min/target/compile SDK versions as described above).

ProGuard

AJVoIP is already minified/optimized and you don't need to do anything extra in your project R8/ProGuard related settings to handle the AJVoIP library.

Old notes (you can ignore this section with the latest AJVoIP versions):

Since AJVoIP is already minified/optimized, you can exclude it from ProGuard, by adding this line in your project proguard-rules.pro file:

*-keep com.mizuvoip.jvoip.** { *; }*

The latest versions (since May 2025) of the library is already shipped with this rule built-in (set by the consumerProguardFiles), so there is no need to do anything in your project.

You should be able to use any proguard rules for your project as this will should not affect AJVoIP usage (the SIP library itself is already highly optimized, but with its interface class kept untouched).

Here is an example proguard-rules.pro file (remove the .txt extension).

The following settings might be used with old IDE/gradle versions (otherwise R8 is the default in the new versions):

Enable the newer R8 engine. Set this in your gradle settings: minifyEnabled true

For a more aggressive optimization, set the following in your gradle.properties file: android.enableR8.fullMode=true

Example code

```
//error handling removed for simplicity
package yourpackagename;
import com.mizuvoip.jvoip.*;  //import AJVoIP

//create SipStack class object instance:
SipStack mysipclient = new SipStack();
//initialize the SDK:
mysipclient.Init((Context)this);

//subscribe to notification events
MyNotificationListener listener = new MyNotificationListener();
mysipclient.SetNotificationListener(listener);


//set parameters (replace uppercase words):
mysipclient.SetParameter("serveraddress", "VOIP_SERVER_IP_OR_DOMAIN");
mysipclient.SetParameter("username", "SIP_USERNAME");
mysipclient.SetParameter("password", "SIP_PASSWORD");
mysipclient.SetParameter("loglevel", "5");
//you might set other parameters here

//start the sipstack:
mysipclient.Start();

//register to your SIP server if needed (usually not needed since the above Start can also auto-register, depending on the "register" parameter):
//mysipclient.Register();

//make a call to a user/extension/phone number/SIP URI (note: 1-2 seconds might be needed between Start/Register and Call for the sipstack to initialize)
mysipclient.Call(-1, "DESTINATION");
//the following lines should be used from another function
//during the call you might call call divert functions by your app logic or on user interaction, for example Hold()

//call hangup to end the call (possibly triggered by a "Hangup" button pressed by the user):
mysipclient.Hangup(-1);
//stop AJVoIP when you don't need it anymore
mysipclient.Unregister();
sipnotifications.Stop();

//handle the notifications:
class MyNotificationListener extends SIPNotificationListener
{
  //see the Notifications chapter and the javadoc or the ajvoip-sources.jar file for the details.

  public void onAll(SIPNotification e) {
    Log.i("SIP", "Notification received: " + e.toString());
  }

  //override any other members here after your needs.
```

}

See the API and the parameters sections below for the details about the usage.

Sample project

You can download a working example from here.

Usage:

1. Download and unzip the AJVoIPTest.zip file
2. Open the project in Android Studio
3. Add AJVoIP.aar to your project
   Copy the AJVoIP.aar into the \app\libs\ folder! (a demo version can be downloaded from here or use the package sent to you by Mizutech)
   Optionally copy also the ajvoip-sources.jar file for better inline documentation in your IDE. It might need to be selected from "Choose Sources" if not recognized automatically.
   More help here.
4. The project have been generated on Windows 10, using Android Studio Ladybug with Gradle 8.9 (AGP 8.7.3). You might need slight modifications for your environment (for different operating systems or build tools). You might also need to adjust the SDK version and location in the configuration files (local.properties, \app\build.gradle, etc).
5. From Android Studio select Rebuild Project from the Build menu. Once the rebuild is ready, you should be able to Run it (on your device or in emulator)

The relevant code can be found in the \AJVoIPTest\app\src\main\java\com\ajvoiptest\MainActivity.java file.
This is an Android Studio example project. If you are using Eclipse or other environment, then just inspect the above file or convert the project to your environment.

For a full demonstration of the SDK capabilities you might check the MizuDroid Android softphone which is also based on this SDK.

Features

AJVoIP is a full featured SIP library for Android with all the usual telephony features implemented and many other extra features.

- Standard SIP client for calls (in/out), chat, conference and others
- **SIP** and RTP stack compatible with any VoIP server or client (Cisco, Asterisk, gateways, ATA, softphones, IP Phones, X-Lite and many others)
- Compatible with all Android devices (phones, tablets, TV's and others). SDK compatibility with 100% of the market share (min SDK: 9) and always optimized for latest Android versions while keeping full backward compatibility even with ancient devices
- Protocols: SIP/SIPS, RTP/SRTP. Transport: UDP, TCP, TLS, TCP tunnel, SOCKS proxy traversal, HTTP proxy traversal, HTTP, VPN tunneling
- NAT/Firewall support: stable SIP and RTP ports ,keep-alive, UPnP, rport support, fast ICE/fast STUN protocols and auto configuration
- Encryption: **TLS/SRTP**, tunneling and peer to peer encrypted media (if direct routing is not disable by your SIP server SDP negotiation)
- RFC's: 2543, 3261, 2976, 3892, 2778, 2779, 3428, 3265, 3515, 3311, 3911, 3581, 3842, 1889, 2327, 3550, 3960, 4028, 3824, 3966, 2663, 3022 and others
- Supported methods: REGISTER, INVITE, reINVITE, ACK, PRACK, BYE, CANCEL, UPDATE, MESSAGE, INFO, OPTIONS, SUBSCRIBE, NOTIFY, REFER
- Audio codec: PCMU, PCMA, **G.729**, GSM, iLBC, SPEEX, **OPUS**
- RTC Video codec: H264, VP8 (optional as-is)
- HD Audio: Wideband, **ultra-wideband** and full-band codecs (speex, opus)
- Audio enhancements: PLC (packet loss concealment), **AEC** (acoustic echo canceller), Noise suppression, Silence suppression, AGC (automatic gain control), VAD (voice activity detection), audio focus and auto QoS (quality of service)
- **Conference** calls (built-in RTP mixer)
- **Voice recording** (SIPREC, local, FTP or HTTP upload in wav, gsm or ogg format), SIPREC, custom audio streaming (to external app or service)
- DTMF (SIP INFO method in signaling, RFC 2833, In-Band)
- **IM/Chat** (RFC 3428), offline chat support (caching messages when peer is not online), group chat
- IMS/3GPP (basic compatibility and features such as USSD and 3GPP SMS)
- **Presence** capability and **BLF** (busy lamp field)
- Redial, call **hold**, MOH (music on hold), mute, **forward** and **transfer** (attended and unattended)
- Call park and pickup, barge-in
- Additional call features: call fork, re-INVITE, 3PCC, ED-137, early media, local ring-back, PRACK and 100rel, replaces
- Balance display, call timer, inbound/outbound calls, Caller-ID display, **voicemail** (MWI)
- Support for credit (balance) and call rating display
- Custom ringtone
- Contact management
- Auto handle phone power state, idle state, charging and airplane mode
- Call optimizations such as proximity sensor, WiFi lock/reconnect and optimal wake-looks (CPU/keyboard/screen)
- Native call integration support (including ConnectionService support; optional for custom builds since it requires more permissions)
- Unlimited lines, multiple accounts
- Support for run as service with minimal battery usage (including bypassing background service limitations)
- Support for **push notifications**

- A long list of other features (see the parameter list and the API for the details)
- Flexibility (all parameters/behavior can be changed/controlled by settings and/or the API)
- **Auto adapt** to hardware: auto turn on/off features based on CPU type and RAM size, so it will run also on low-end hardware
- **Reliable incoming calls** with all-time availability using improved service mode and/or push to handle situations where your device or app is closed, **standby** or the device is in **doze** or **sleeping** state
- Auto adapt to networking circumstances: different codec parameters and prioritization based on network type (WiFi/3G/LTE/others) and quality (bandwidth/packet loss/jitter)
- Stable high-level API: Always backward and forward compatible including the parameters and the API (you don't need to change your code when upgrading to new versions)

Licensing

The Mizu Android Java VoIP SDK (AJVoIP) is sold with life-time unlimited client license (Advanced and Gold) or restricted number of licenses (Basic). You can use it with any VoIP server(s) which belongs to you or your company. Your VoIP server(s) address (IP or domain name) will be hardcoded into the software to protect the licensing. You can find the licensing possibilities on the Android SIP SDK page. After successful tests please ask for your final version at info@mizu-voip.com. Mizutech will deliver the AJVoIP build (your licensed copy) within one workday on your payment.

Release versions don't have any of the demo limitations and can be fully customized with your branding with "mizu", "mizutech" words, links and all reference to Mizutech removed. Your final build must be used only for you company needs (including your direct sip endusers and VoIP clients).

The library doesn't have any external dependencies (everything it requires is within the .aar / .jar file).

Title, ownership rights, and intellectual property rights in the Software shall remain with MizuTech and/or its suppliers.
The agreement and the license granted hereunder will terminate automatically if you fail to comply with the limitations described herein. Upon termination, you must destroy all copies of the Software. The software is provided "as is" without any warranty of any kind.

You may:

Use AJVoIP on any number of devices or as permitted by your license
Use AJVoIP as an SDK embedded in your Android project
Use AJVoIP with VoIP servers for which you have license for (after the agreement with Mizutech). All the VoIP servers must be owned by you or your company. Otherwise please contact our support to check the possibilities (the Gold license allows 10+ or freely configured SIP server address)

You may not:

Resell AJVoIP
Use AJVoIP with VoIP servers not communicated with Mizutech, except Gold license
Reverse engineer, decompile, disassemble or modify the software in any way (except modifying the parameters and using it via the public API)

Demo version:

We are providing a demo version which you can try and test before any payment. The demo version has all features enabled but with some restrictions to prevent commercial usage. The limitations are the followings:

- maximum 10 simultaneous AJVoIP instances in the same time
- will expire after several months of usage (usually 2-8 months, depending on the last release date)
- maximum ~100 sec call duration restriction
- maximum 10 calls / session limitation (after 10 calls you will have to restart)
- will work only maximum 20 minutes (after that you have to restart the AJVoIP library or your application)
- verifications against the mizu license service

Note: for the first few calls there might be fewer limitations than described above.
All the above limitations will be removed from your licensed copy.
On your request we can also send a trial version (will expire after some time but doesn't have the above demo limitations).

API

You can use AJVoIP as an SDK by embedding into your Android application and calling its public API functions.

Add AJVoIP.aar or AJVoIP.jar to your project and use the following import statement in your code:

```
import com.mizuvoip.jvoip.*;
```

The whole API is defined in the SipStack class for ease of use.
For maximum flexibility, the Android SIP SDK exposes numerous functions, however this doesn't mean that the usage is difficult. You can ignore most of them and use only those few relevant for your needs also outlined in the usage example.
For example if you just need to make simple calls, then the following five functions will cover all your needs: Init, SetParameters, Start, Call, Hangup.

General considerations

All API calls are **thread safe** and will **not throw exceptions** (on exception or error they will send "ERROR" notifications and/or return false/-1 depending on the context).
API calls never returns null (in case of strings you will always receive an empty string return value when needed instead of null).

Most of the functions **return a boolean** value. True when the operation was completed or initiated successfully, otherwise false.
Most of the functions are executed **asynchronous**ly (non-blocking) such as the Call and Register. This means that it can return a true value immediately and fail later. For example for the Call function the return value means only that the call was initiated successfully. At this point we don't know if the call will be successful (connected) or not (rejected or failed). You can get the call status by parsing the notification messages or you can periodically poll AJVoIP status with the GetStatus function.

The **line** parameter is part of most of the functions and it means the channel number to be used if you wish to select a specific session.
For simple use cases usually you just have to set it to **-1**. For the details, see the "Line parameter" and the "Multiple lines" FAQ points.

Functions

The list of public functions exported by the SipStack class are described below:

SipStack()

Constructor. Create new SIP stack instance by creating a new SipStack object:

```
SipStack mysipclient = new SipStack();
```

boolean Init(Context context_in)

Initialize the SIP engine. This is the very first function you must call to initialize the internal data structures.
The context_in parameter is your application Context which can be obtained from any activity or as described here.

boolean SetParameter(String param, String value)

Configure the Android SIP library by passing any settings.
The full list of available parameters are listed here.

Example: `mysipclient.SetParameter("loglevel", 5);`

Notes:
Some basic parameters should be set before the Start function have been called (such as "register"), however most parameters can be also changed at run-time.
Some advanced parameters can be set also by line using the SetLineParameter function (for advanced users only; use the other API functions instead to control the individual lines).

boolean SetParameters(String parameters)

You can pass a set of parameters with this function in value=key lines separated by CRLF (\r\n).

It has the same effect like the above SetParameter function, but uses a list of parameters at once instead of setting them individually.
See the full list of supported parameters here.
Example: `mysipclient.SetParameters("loglevel=5\r\ndtmfmode=1\r\n");`

boolean SetCredentials(String server, String username, String password, String authname, String displayname)

Will set the SIP server address (ip:port or domain:port) the SIP username and the password. These values can also be preset by parameters. Parameters with empty strings will be omitted. For example if you would like to change only the username and the password, you can write SetCredentials("","newusername", "newpassord")
If authname is empty, then the username will be used for authentications. The displayname is usually empty (no special displayname will be presented for peers). If other parameters are empty, then they can be specified by user input (If the Android SIP SDK has a visible user interface).
Note: you can also use the SetParameter(s) to initialize the SDK with the credentials.

boolean SetCredentialsMD5(String server, String username, String md5, String realm)

Instead of passing the password directly you can use MD5 checksum.
In this case the md5 parameter must be the md5 checksum for username:realm:password

The realm parameter is optional (can be set as an empty string) but it is recommended for easy error detection. If present and the server realm don't match with this one, an error message will be displayed by AJVoIP.

boolean Start()

Will start the SIP engine (starting the internal main thread).

boolean StartStack()

This function call is optional to start the sip stack on demand.
If not called, then the sip stack is started anyway if the "startsipstack" parameter is set, otherwise will start at first registration or outgoing call attempt.

boolean Register()

Will connect to the SIP server. This can be also done automatically by parameter ("register"). This function have to be called only once at the startup. Further re-registrations are done automatically based on the "registerinterval" parameter. When called subsequently, the old registrar endpoint is deleted, a new one will be created with a new call-id and AJVoIP will reregister.
Even if you wish to force re-registration, you should not call this more frequently than 40 seconds (because up to 40 seconds might be needed for a slow registration attempt)

boolean Register(String server, String username, String password, String authname, String displayname)

Same as the above but with parameters. Otherwise you can set the credentials with the SetParameter(s) function before to call Register().
See the Accounts chapter if you wish to use multiple SIP accounts.

boolean Unregister()

Will stop all endpoints (hangup current calls if any and unregister)

boolean CheckVoicemail(int line)

Will (re)subscribe for voicemail notifications. No need to call this function if the "voicemail" parameter is set to 2.
The line parameter should be set to -1.

boolean SetLine(int line)

Will set the current channel. (Use only if you present line selection for the users. Otherwise you don't have to take care about the lines).
Note: Instead of using each API call with the line parameter, you can just use this function when you wish to change the active line and use all the other API calls with -1 for the line parameter.

int GetLine()

Will return the current active line. This should be the line which you have set previously except after incoming and outgoing calls (the SIP client will automatically switch the active line to a new free line for these if the current active line is already occupied by a call)

int GetLineStatus(int line) or string GetLineStatusText(int line)

Query the status of the line.
Note: this is rarely needed since you receive the status also by event notifications

String GetLineDetails(int line)

Get details about a line. The line parameter can be -1 (will return the "best" line status).
Will return the following string:
LINEDETAILS,line,state,callid,remoteusername,localusername,type,localaddress,serveraddress,mute,hold,remotefullname

Line line number (might be useful if you pass -1 as line parameter
State same as in STATUS notification
CallID: SIP session id (SIP call-id)
Remoteusername is the other party username (if any)
Localusername is the local user name (or username).
Type is 1 from client endpoints and 2 from server endpoints.
Localaddress: local IP:port

Serveraddress: remote IP:port
Mute: is muted status. 0=no,1=undefined,2=hold,3=other party hold,4=both in hold (or if you wish to simplify: 0 means no, others means yes)
Hold: is on hold status: 0=no,1= undefined,2=hold,3=other party hold,4=both in hold (or if you wish to simplify: 0 means no, others means yes)
Remotefullname is the other party display name if any

boolean Call(int line, String peer)

Initiate call to a number or sip username.
If the peer parameter is empty, then will redial the last number.

- *Example:*
  - *mysipclient.Call(-1, "004401234567"); //call to phone number on the default line*
  - *mysipclient.Call(2, "bob"); //call to sip username on the second line*
- *It is recommended to pass only the peer number, extension or username to this function and not the full SIP URI (the SIP stack will construct the full SIP URI with the serveraddress already set)*
- *This function is for normal audio calls. For video use the VideoCall API.*

boolean Hangup(int line, String reasontext)

Disconnect current call(s). If you set -2 for the line parameter, then all calls will be disconnected (in case if there are multiple calls in progress). The "reasontext" parameter is optional.

boolean Accept(int line)

Connect incoming call.

boolean Reject(int line)

Disconnect incoming call. (Hangup will also work)

boolean Ignore(int line)

Ignore incoming call.

boolean Forward(int line, String peer)

Forward incoming call to peer (with 302 Moved Temporarily disconnect code)

boolean Transfer(int line, String peer)

Transfer current call to peer which is usually a phone number or a SIP username. (Will use the REFER method after SIP standards).
You can set the mode of the transfer with the "transfertype" parameter.
If the peer parameter is empty than will interconnect the currently running calls (should be used only if you have 2 simultaneous calls)

boolean SetSpeakerMode(boolean loud)

Switch between speakerphone and loudspeaker.
Set the loud parameter true if you wish to set to speakerphone/loudspeaker. Otherwise false for normal earspeaker/earpiece.
Note: You can also preset it using the aspeakermode parameter. More details here.

boolean Mute(int line, boolean mute, int direction)

Mute current call.
The line parameter is the endpoint. -2 for all or -1 for current line.
Set the mute parameter to true for mute or false to un-mute.
The direction can have the following values:

- 0: mute in and out
- 1: mute out (speakers)
- 2: mute in (microphone)
- 3: mute in and out (same as 0)
- 4: mute default (set by the "defmute" parameter, which is "mute microphone only" by default)

int IsMuted(int line)

Return if the selected line is muted or not.
Return values:

- -1: unknown
- 0: not muted
- 1: both muted (in/out)
- 2: out muted (speaker)
- 3: in muted (microphone)
- 4: both muted (in/out; same as 1)

boolean Hold(int line, boolean hold)

Hold current call. This will issue an UPDATE or a reINVITE.
Set the second parameter to true for hold and false to reload.

int IsOnHold(int line)

Query if the selected line is on hold or not
Return values:

- -1: unknown
- 0: no
- 1: not used
- 2: on hold
- 3: other party held
- 4: both in hold

Note: pass -2 for the line to find if any endpoint is in hold

boolean Conf(String peer)

Add people to conference.
If peer is empty than will mix the currently running calls (if there is more than one call)
Otherwise it will call the new peer (usually a phone number or a SIP user name) and once connected will join with the current session.

boolean ConfEx(int line, String peer, boolean add)

Add/remove people or line to conference.
If peer is empty than:
-if add is true:
-if line is -2 then it will mix all the currently running calls (if there is more than one call)
-if line is not -2, then it will add the channel to the conference
-if add is false:
-if line is -2 then it will destroy the conference (but will keep the calls on individual lines)
-if line is not -2 then it will remove the selected line from the conference
Otherwise it will call the new peer (usually a phone number or a SIP user name) and once connected will join with the current session.

boolean Dtmf(int line, String dtmf)

Send DTMF message by SIP INFO, In-Band or RFC2833 method (depending on the "dtmfmode" parameter). Please note that the dtmf parameter is a string. This means that multiple dtmf characters can be passed at once and the Android SIP library will streamline them properly. Use the space char to insert delays between the digits. The dtmf messages are sent with the protocol specified with the "dtmfmode" parameter.
Example: Dtmf(-2," 12 345 #");
Note: dtmf messages can be also sent by adding it to the called number after a comma. For example if you make a call to 123,456 then it will call 123 and then it will send dtmf 456 once the call is connected.

boolean Info(int line, String msg)

Send any SIP INFO message as defined in RFC 2976.
Use this API to send any custom INFO messages to the server or to the connected peer (in case if your server will forward it to other peer and not discard it).
The msg string will be sent in the INFO body.
The Content-Type can be specified with the infocontenttype parameter. For example you might set the infocontenttype to "application/mydata". If the infocontenttype is not set then it will be auto guessed as application/json, application/xml or application/octet-stream"
Feedback about the message delivery can be received with the INFO notifications.

Example: Info(-2,"MyMessage");

For more details see the Custom INFO messages FAQ point.

boolean SendUSSD(int line, String method, String ussd)

Send an USSD message after the IMS 3GPP TS 24.390 standard.
The method parameter can be set to "INVITE", "INFO" or "BYE".
If INVITE, then a new session will be created. Otherwise will use the session suggested by the line parameter.
The ussd parameter have to be set to the USSD string (for example "*135#") or whole XML ("<?xml version … ").

Will return true if send initialized successfully or false on failure.
Further feedback about the actual delivery or incoming ussd messages can be obtained from the USSD notifications.
More details here.

boolean SendChat(int line, String peer, String message)

Send a chat message. (SIP MESSAGE method after RFC 3428)
Peer can be a phone number or SIP username/extension number.

Parse the CHAT notification to check if delivery succeeded or failed.
On successful delivery you will also receive a log like: EVENT, chat sent successfully
On failed delivery you will also receive a log like: WARNING, chat message not delivered
You can also send typing notifications with the SendChatIsComposing API.

boolean SendSMS(int line, String peer, String message)

Send a SMS message if softswitch has SMS delivery capabilities (Otherwise might try to deliver as IM).
The message is delivered as a 3GPP SMS, SMS HTTP API request or a standard SIP MESSAGE with X-Sms: Yes header (in this case the server is responsible to convert it to SMS).
Note: incoming SMS messages are reported as CHAT notifications.

boolean VideoCall(String destination, int fragmentResId, FragmentActivity fragmentinstance)

Initiate RTC video call to destinationnumber which can be a number, username or SIP URI.
The fragmentResId is the Integer resource ID of the FrameLayout in which the video fragment will be loaded.
The fragmentinstance is the instance of the Activity in which the video module will be displayed. The video module consists of an android Fragment and the FragmentActivity will be its parent.

Place the <FrameLayout> in your Activity's content XML.

```
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
android:id="@+id/fragment_container"
android:layout_width="fill_parent"
android:layout_height="350dp" />
```

In the above example the resource ID will be: R.id.fragment_container

RTC capabilities must be available for the video to work. More details here.

boolean AcceptVideo(int fragmentResId, FragmentActivity fragmentinstance)

Accept RTC video call.
The fragmentResId and fragmentinstance is as described at the VideoCall API above.
If video interface is not loaded yet, then will initiate it now and call back. If incoming call is not with video, then it might result in simple audio call.

boolean RemoveVideo()

Will remove the video fragment (will also disconnect pending video call if any).
Instead of this function, you should just hide/destroy your video container element from the user interface.
More details can be found here.

boolean SetVideoDisplaySize(int type, int width, int height)

Use this function to control the video display size (both for remote and local video).

Accepted parameters:

- type: 1=for remote video container, 2=for local video container
- width: width in pixels
- height: height in pixels; this parameter can be 0, and the height will be set depending on the video's aspect ratio

boolean VoiceRecord(int startstop, int now, String filename)

Will start/stop a voice recording session.

- Startstop: 0 to stop, 1 to start locally, 2 to start remote ftp, 3 start to record both locally and to remote ftp, 4 start to record as it is set by the "voicerecording" parameter
- Now: used if the startstop is set to 0. 0 means that the recorded file will be saved and/or uploaded at the end of the conversation only. 2 means that the file will be saved immediately
- Filename: file name used for storing the recorded voice (if empty string, than will use a default file name)

This function should be used only if you would like to control the recording duration.
If all conversations have to be recorded, then just set the "voicerecording" parameter after your needs.
The last recorded call can be played by calling the PlaySound with the file set to "lastvoicerecord".
More details here.

boolean PlaySound(int start, String file, boolean islocal, boolean toremotepeer, int line)

Play any sound file.
At least wave files are supported in the following format: PCM SIGNED 8000.0 Hz (8 kHz) 16 bit mono (2 bytes/frame) in little-endian (128 kbits).
This function can be used to play audio locally, but also for remote streaming.
For remote playback, make sure to force a narrowband codec if your file is narrowband and wideband codec (speex, opus) if your file is wideband (16 kHz, 16 bit mono in little-endian).
The file must be found locally.

- start: 1 for start or 0 to stop the playback, -1 to pre-cache
- file: file name
- islocal: true if the file have to be read from the device file system. False if remote file (for example if the file is on the webserver)
- toremotepeer: stream the playback to the connected peer
- line: used with toremotepeer if there are multiple calls in progress to specify the call (usually set to -1 for the current call if any)

For remote playback you can pass any file path which can be passed for FileInputStream as our code uses FileInputStream to load the file like this:

```
FileInputStream fileinputstream = new FileInputStream(file);
```

Examples:

-playback a file locally (mysound.wav must):

```
PlaySound(1, "mysound.wav",  false, false, -1)
```

-playback a file to the connected remote peer (mysound.wav must exist):

```
PlaySound(1, "mysound.wav",  false, true, -1)
```

-stop the playback:

```
PlaySound(0, "", false, false, -1)
```

In case if you just want to play a file locally, then you can do it also directly from your code like this:

```
MediaPlayer mp = MediaPlayer.create(appcontext, Uri.parse(file));
mp.start();
```

boolean StreamSoundBuff(int start, int line, byte[] buff, int len)

Stream from raw wave audio PCM (Linear PCM) buffer or from RTP packets.
This function will stream the supplied audio buffer to the peer endpoint, transcoding if necessary.

Parameters:

- start: 1 for start or 0 to stop the playback
- line: specify the channel in case if there are multiple calls in progress (usually set to -1 for the current call if any)
- buff: audio buffer (raw PCM data or RTP packet)
- len: length of the buff (ideally it should be multiple of 360 if buffer is raw PCM, otherwise the size of the RTP packet).
  If len is 0 or negative and start is 1, then len will be set from buff.length.

Notes:

- The buff should not contain any file header if you are using raw PCM (only raw linear PCM audio data) and it should contain a full RTP packet (with the RTP header) if you are passing RTP data. Set the streamsoundisrtp parameter to -1 to auto-guess raw PCM vs RTP (this is the default value), set to 0 if you supply raw PCM buffer or set to 1 if you supply RTP packets.
  The data will be automatically converted to the desired format (depending on the VoIP codec actually used for your call) and most format conversions are supported.

- If the buff is raw PCM, then you can set its format with the following parameters:
  - audioinput_bitdepth: bits in each sample (for example 8 or 16). Default and preferable is 16
  - audioinput_channels: 1 for mono, 2 for stereo. Default and preferable is 1 (mono).
  - audioinput_samplerate: samples per second in Hz. Preferable is 8000 or 16000.

  If you wish to avoid conversions, then use 16 bit (2 bytes/frame) little endian mono in 8000 Hz (if you are using narrowband codec such as G.729/GSM/G.711/PCMA/PCMU) or in 16000 Hz (if you are using wideband codec such as opus or speex).
- You might set the useaudiodevicerecord parameter to false if you need streaming but don't have an audio recorder device installed.
- If previously supplied buffer(s) are not completed yet, then the new one will be queued.
- If you are streaming in real-time (feeding the API with short RTP/audio packets) then you might set the realtimeplayback parameter to 1.
- Alternatively you can use the StreamSoundStream function to pass an InputStream instead of byte buffers.
- In case if you wish to receive the audio stream from the remote peer endpoint, see this FAQ point instead.

See more details here.

byte[] GetMedia(int timeout)

Retrieve the next media (audio or video) packet as a byte buffer from the AJVoIP internal queue.
The buffer format (prefix bytes) can be specified with the related sendmedia_... parameters as described in the streaming FAQ.

This is a blocking function call, so you should call it from a separate thread.
The timeout can be specified in milliseconds. If it is higher then 0, then AJVoIP will use an additional waiting lock. If 0 then will issue a simple/fast blocking read.
The internal queue size is 900 and on overflow it will remove all queued packets (if you are not calling this function or not fast enough).

The internal media packets queueing will start only if the sendmedia_mode is set to 2 or 3 or at first call to this function.
(The first call for this function will also set the sendmedia_mode parameter to 2 or 3 if it was not set before)
If you don't need media streaming anymore, then you might call the GetMediaEnd() function to stop queuing any more media packets, otherwise it will stop automatically after some time of no usage or overflow.

The receive byte buffer might contain some header bytes as specified by the sendmedia_ parameters.
For more details see the streaming FAQ.

boolean SetVolume(int dev,int volume, int line)

Set volume (0-100%) for the selected device.
The dev parameter can have the following values:
0 for the recording (microphone) audio device
1 for the playback (speaker) audio device
2 for the ringback (speaker) audio device
The line is an optional parameter to be used only if you need different volume for separate calls.
Note: The ringer volume might not be always honored and might not change during ring playback.

int GetVolume(int dev, int line)

Return the volume (0-100%) for the selected device.
The dev parameter can have the following values:
0 for the recording (microphone) audio device
1 for the playback (speaker) audio device
2 for the ringback (speaker) audio device
The line is an optional parameter to query the volume of a specific call line in case if you set the volume per call. Otherwise don't use or set to -2.

boolean SetIdle(boolean idle)

You might use this API to further extend the battery life (above the usual phone power state changes).
The sipstack internally has an idle state which can be useful to decrease CPU and RAM usage by using less resources, slowing down all operations, running threads in low priority and increases internal timer intervals. This means that the app can enter in idle state even if running in foreground on a non-sleeping phone state (but with no user interaction and no important task running such as an ongoing call)
You should call this function with the idle parameter set to true if the screen is dimmed/off, there is no call and no user interaction with the application for a while. Call with the idle parameter set to false when the user begins any interaction with the application. The sipstack will also automatically disable the idle state when there is a new incoming call, chat or other important event that might require user interaction or CPU resources.

String VAD()

Returns voice activity statistics. See the VAD notification for more details.

For this to work you should set the vad parameter to at least 2 (4 for full report).
Note: if you call this function, VAD notifications will not be sent automatically anymore. (So you will need to continue to poll for the details).
More details here

String RTPStat()

Returns media statistics. See the RTPSTAT notification for more details.
For this to work you should set the vad parameter to at least 2 (4 for full report).
Note:
RTP statistics can be sent also automatically if you set the rtpstat parameter.
if you call this function, RTPSTAT notifications will not be sent automatically anymore
More details here.

string GetVersion()

Return the program version number.

String GetStatus(int line, int strict)

Returns line status or global status if you pass -2 as line parameter. The possible returned texts are the same like for notifications (listed below).
If the strict variable is set to 1, then it will return "Unknown" if no such line is activated. If the strict variable is set to 0, then it will return the default active line if the line doesn't exists.
You should use the notifications described below to get the actual status of AJVoIP instead of continuously polling it with this function call.

boolean SetNotificationListener(SIPNotificationListener listener)

Subscribe to notification events.
Create a subclass of the AJVoIP SIPNotificationListener first, then pass this object as a listener.
You will receive the SIPNotification objects overriding the member functions.
More details:

- Notifications
- Javadoc
- Code template
- Full/working example code

String PollNotificationStrings ()

Should be used only if you wish to receive the notifications as strings.
Use the SetNotificationListener() instead to receive SIPNotification objects instead of strings.

Return the notification strings. You should poll for the notifications periodically from a separate thread. It will return accumulated events since the last function call (notifications separated by \r\n -CRLF).
More details here.

Note:

- *Once a notification string have been read then it will be cleared from the internal list, so it is guaranteed that you will never receive duplicates.*
- *The old function name as GetNotifications which was renamed for more clarity (the old way will still be kept)*

String GetNotificationStrings ()

Should be used only if you wish to receive the notifications as strings.
Use the SetNotificationListener() instead to receive SIPNotification objects instead of strings.

Same as above PollNotificationStrings(), but will blocking wait for data (more efficient) and it should be used from a separate thread.
The old function name as GetNotificationsSync which was renamed for more clarity (the old way will still be kept)

SIPNotification PollNotification()

Use the SetNotificationListener() instead to receive the notifications as objects instead of strings.

Same as above PollNotificationStrings(), but will return a SIPNotification object instead of string.

SIPNotification ParseNotification (String notificationstring)

Convert a notification string to a SIPNotification object.

Useful only if you receive the notifications as strings (instead of SIPNotification objects which is recommended).
More details here.

boolean SetPushNotifications(int pushnotifications, String clientid, String packagename, String gateway)

If you wish to use VoIP push notifications then you might use this function to have it assisted by the SIP SDK (otherwise you can implement it also separately in your code).

Parameters:

- pushnotifications: 0=disable,1=auto,2=direct,3=via gateway
- clientid: your app token ID (FCM registration token returned by registering to the notification service)
- packagename: the package name of your application to be used for push notification. If empty then it will be read from your app context. If using mizu push gateway then you might leave it empty or set to "com.mizuvoip.mizudroid.app"
- gateway: if your SIP server doesn't support push notifications then you might use a gateway (the SIP stack will register also via this gateway)

To activate push notifications, first you will need a Firebase project and register to FCM from your own code as described here.
Then once you call this function with the acquired client token id, the SIP stack will begin to send push bind request (extra SIP headers or tags in the REGISTER requests using the protocol specified by pushtype).
This is required for the server side to learn your token and be able to send push notifications on incoming calls or chat.
The purpose of these notifications should be only to wake up your application if in background/sleeping mode, so it can start/register and process the incoming INVITE as normally. For chat message it should be enough if you just display the message and eventually store it (the user can launch your app by tapping on the displayed notification).

Start example: SetPushNotifications(1, "TOKEN", "", "");
Stop example: SetPushNotifications(0, "", "", "");

If you set the pushnotifications parameter to 1 (auto), then will try push with your server first (as described in RFC 8599 so your server must answer to REGISTER with Feature-Caps: *;+sip.pns="fcm") and if fails then it will try via push gateway.

For more details read here.

int GetPushNotifications()

Returns the push notification subscription state from the SIP stack. This should be used only if you let the SIP stack to handle the push notifications for you by calling the above SetPushNotifications API first.
Possible return values:

- 0: disabled (push notification disabled)
- 1: enabled unknown (might work)
- 2=success (push notification working)
- 3=failed (push notification subscription failed)

Note: you can assume that push notifications are working if you were capable to subscribe to FCM and get a valid token AND the GetPushNotificatons function returns 2.

Accounts

This chapter is about SIP account management, in case if you wish to use multiple accounts.
In case if you are using the SIP SDK only with one SIP account, then you can skip this chapter and configured the account with the serveraddress/username/password main parameters or with the Register() function.

Concepts

The VoIP SDK is capable to register multiple SIP accounts at the same time. This can be useful to be able to make and receive calls with using multiple SIP accounts on the same or on different SIP servers. You should read trough this chapter only if you wish to use multiple accounts from AJVoIP.

In AJVoIP there can be one main account and multiple extra/secondary accounts.
The main/primary account can be configured directly with the serveraddress/username/password main parameters or with the Register() function.
The extra/secondary account(s) configuration is described in this chapter.
Main and secondary accounts works in similar ways, but the main account has the followings privileges:

1. AJVoIP will try to register the main account first when possible and the secondary accounts register will start on main account success or failure
2. The AJVoIP main state (STATUS) will reflect the main account status
3. Outgoing calls will be initiated with the main account (so you might need to switch the main account if you wish to make a call with another/secondary account)

- Multiple accounts can be configured at once, using the extraregisteraccounts parameter or the RegisterEx function passing a string with the accounts fields in the required format. See the Extra accounts string format FAQ for more details.

- Otherwise, you might use the AddAccount / GetAccount / DelAccount / ListAccounts / GetMainAccount / SetMainAccount functions instead for account management.

Notes:
-Internally the SIP stack identifies the separate accounts by their base URI: user@domain (without the port part, which means that it is impossible to create accounts same username, same server but different server port; in this case AJVoIP will merge the account instead with the old/existing one)
-Configured accounts will be automatically saved/remembered, so it is not necessary to set the accounts at every startup (unless you delete/clear the settings or if you configured AJVoIP to not store the settings). However adding the accounts again will also work (AJVoIP will merge the setting or will skip if already exists and all parameters are the exact same again)
-There should be always a main account. AJVoIP will auto-select a main account if you haven't set it or if you delete the main account.
-The main account might be changed automatically by AJVoIP depending on the autoswitchmainacc parameter. On main account change you can receive ACCOUNT notifications.
-The registrations will be kept automatically by AJVoIP (will automatically re-register when needed; you don't need to explicitly register the accounts again)
-You can watch for the REGISTER notifications to find out the register state of the separate accounts
-STATUS about registrations are not reported from secondary accounts unless you set the secondarystatus parameter to 1 (but otherwise you will receive call state STATUS notification while in call as normally)
-If you will connect to more then 10 servers via TCP/TLS then you might need to increase the value of the maxsigstreams parameter (default is 10). You might also set the multiplesigstreams parameter to 2.
-The line parameter for the call has nothing to do with the accounts. That only means call lines, useful if there are multiple simultaneous calls.
See the Multiple lines section if you wish to perform multiple simultaneous calls on these accounts
-You can unregister all accounts at once or individually using the Unregister() function
-All accounts will be automatically unregistered when AJVoIP terminates (except if the needunregister parameter is set to false)
-By default all the old extra accounts might be unregistered (and registered again) with the Register()

If you need a more strict/separated control on the accounts then you can also just launch AJVoIP multiple times (multiple concurrent instances) with different parameters (different SIP account configuration).
Multiple AJVoIP/webphone instances per account requires a bit more system resources –some more RAM for each instance- but it might be more convenient to work with if you have to do different things with the accounts (such as different servers/transport protocols/different handling).
Using this feature (multi-accounts in one instance) can be useful if you wish to handle the accounts in a similar way or if you need many accounts (to optimize system resource utilizations).

boolean RegisterEx(String accounts)

This is the old-style API to set all accounts at once. You might use only the below functions instead (but it is possible to use both this and the below functions combined).
You can use this function to set one ore more secondary accounts (up to 99) on the same or different servers.
Multiple accounts can be passed at once separated by semicolons (;).
An account must be specified as a SIP URI or as the following parameters separated by comma (,) :

1. Server address: SIP server address: domain or IP:port
2. Username: SIP account user ID / extension ID
3. Password: SIP account password
4. Register interval: register refresh timer in seconds (default is 3600)
5. SIP proxy: outbound SIP proxy if any (required only if it is different then the above server address
6. SIP realm: only if a different realm must be used
7. Auth user name: if separate extension id and authorization username have to be used
8. Displayname: optional displayname / caller-id
9. Transport protocol (-1/default/empty, 0/UDP, 1/TCP, 2/TLS. Default is -1)
10. Main (-1: auto, 0: no/secondary, 1: yes/main/primary. Default is -1)
11. FCM (ignore. Set to empty or 0)
12. Enabled (false: disabled, true: enabled. Default is true.)

The Username parameter is mandatory, all the others are optional. AJVoIP will use the defaults for the empty parameters.
All accounts have to be passed in a single line which should look like this: server,usr,pwd,ival;server2,usr2,pwd2,ival2;etc…

Examples:
```
mysipclient.RegisterEx("user:pwd@domain:port"); //add a single extra account as a SIP URI
mysipclient.RegisterEx("sip:john:secret@myserver.com:5060"); //add a single extra account as a SIP URI
mysipclient.RegisterEx("\"John Smith\" <john:secret@myserver.com:5060>"); //add a single extra account as a SIP URI specifying also the displayname
mysipclient.RegisterEx("myserver.com,john,secret"); //add a single extra account as parameters
mysipclient.RegisterEx("myserver.com:5061,john,secret,180, , , , ,TLS"); //add a single extra account with 180 as the register interval and TLS transport
mysipclient.RegisterEx("john:jsecret@192.168.1.50:5060;kate:ksecret@192.168.1.50:5060"); //add multiple account as SIP URI's
mysipclient.RegisterEx("92.168.1.50:5060,john,jsecret,180;92.168.1.50:5060,kate,ksecret;92.168.1.50:5060,mary,msecret,600"); //add multiple account as parameters with/without register interval
mysipclient.RegisterEx("null"); //clear old settings
```

Note:
When you call the RegisterEx function, these accounts will be remembered (like you would set them with the extraregisteraccounts parameter) and will be re-used also at next startup. You can clear the old accounts by passing "null" as the accounts parameter.

You can combine RegisterEx with the other account related API's (such as add them at all at start, then manipulate them separately).
See the Extra accounts string format FAQ for more details.

boolean AddAccount(String uri)

Add/register (secondary) account.
The uri parameter is a SIP URI in user@domain format. It is also capable to include the password and port like usr:password@domain:port.
If the account represented by the uri already exists, then it will skip or merge (for example if the password or the port is different).

The uri accepts also parameters separated by comma, as described at the extraregisteraccounts parameter.
Use the below extended AddAccount function instead if you need to specify other settings such as the transport protocol or the proxy address.
If no user or domain part is provided, then AJVoIP will use the main/global username/serveraddress.
Returns true on success and false on failure (such as invalid uri input).

boolean AddAccount(String server, String proxy, String username, String sipusername, String displayname, String password, int transport, long registerinterval, int ismain)

Add/register account.
Parameters:

- server: SIP server address: domain or IP:port
- proxy: outbound SIP proxy if any (required only if it is different then the above server address
- username: SIP account user ID / extension ID
- sipusername: SIP auth username for authentication (if different then the above username)
- displayname: optional displayname / caller-id
- password: SIP account password
- transport: -1: auto, 0: UDP, 1: TCP, 2: TLS
- registerinterval: register refresh timer in seconds (default is 3600)
- ismain: -1: auto, 0: secondary account, 1: main account

If the account represented by the URI (username@server) already exists, then it will skip or merge.
If no username or server address is provided, then AJVoIP will use the main/global username/serveraddress.

boolean SetAccount(String params)

Add/register (secondary) account.
The params parameter is the comma separated accounts parameters as described at extraregisteraccounts.
If the account represented by the URI already exists, then it will skip or merge (for example if the password or the port is different).

boolean SetAccount(String server, String proxy, String username, String sipusername, String displayname, String password, int transport, long registerinterval, int ismain)

Change an existing account.
Parameters: same as for AddAccount.
If the account represented by the URI (username@server) doesn't exists, then it will add as new account.

String GetAccount(String uri)

Get account details.
The returned string will be formatted as described at the extraregisteraccounts parameter.

There is an extended version where you can set the second strict parameter to true if you wish to lookup with exact match only. Default is false.
boolean GetAccount(String uri, boolean strict);

To simplify the API surface, there is no any "Account" class exposed, so you will need a little string parsing to get the account parameters (Parameters are separated by comma as described here)

boolean DelAccount(String uri)

Delete SIP account and unregister.
The uri parameter is a SIP URI in user@domain format.
If you pass "all" as the URI, then it will delete all accounts. (You might use Unregister instead if you wish to unregister all accounts).
If you delete the main account, then AJVoIP might automatically switch to another main account if any.

There is an extended version where you can set the second unregister parameter to false if you don't wish the account to unregister:

boolean DelAccount(String uri, boolean unregister);

String ListAccounts()

Will return the list of the account URI's separated by ;.

There is an extended version with extra parameters:
String ListAccounts(int format, String markmain, boolean withstatus, String separator);
Parameters:

- *format: return the following (instead of URI):*
  - 1: full representation as described at the extraregisteraccounts parameter
  - *2: SIP URI*
  - *3: compact (The most compact format possible. Might be useful to display an account select list for the user. For example if all accounts are on the same SIP server, then it will list only the usernames).*
  - *4: username only*
  - *5: domain only*

  Default is 2
- *markmain: any string to prefix the main account with (for example "*" or "[MAIN]". Default is empty.*
- *withstatus: append account connection/registration state*
- *separator: string to separate the accounts. Default is semicolon ;*

Examples:

- *Display the accounts (for example to allow the user to select one): ListAccounts(3, "*", true, "\r\n");*
- *List the account URI's: ListAccounts(2, "", false, "\r\n");*
- *List the accounts with all their parameters: ListAccounts(1, "", true, "\r\n"); or just ListAccounts();*

To simplify the API surface, there is no any "Account" class exposed, so you will need a little string parsing to get the account parameters with the format set to 1 (Accounts separated by ; or by separator. Individual account parameters are separated by comma. Details here)

String GetMainAccount ()

Get the SIP URI of the main/primary account (if any; otherwise empty string).

There is an extended version with a strict parameter: String GetMainAccount(boolean strict, int format);
If the strict parameter is set to true, then it will return the main global account even if no accounts have been configured. Otherwise, if the strict parameter is false, then it will return the main account only if multiple accounts have been configured. Default is true.
Format is the same like for ListAccounts.

boolean SetMainAccount(String uri)

Change/switch main/primary account.
If the specified account doesn't exists yet, then it will create it.
The main account will be used for the next outgoing call.

String GetAccountForLine(int line)

Get the account used for a call.
Line is the call line number.
Returns the account URI if any.

There is an extended version with a strict parameter: String GetAccountURI(int line, boolean strict, int format);
If the strict parameter is false, then it will return local endpoint URI even if account doesn't exists. If true then it will return only exact account match. Default is false.
The format parameter is like at ListAccounts().

int GetAccountRegState(String uri)

Query account register state. Useful if you are using multiple accounts.
With the uri parameter pass the account URI in user@server format (or only the user or the server part if there are no other similar accounts)
Returns: -4: not initialized or error, -3: invalid account input, -2: no such account, -1: not registering, 0: working, 1: success, 2: failed, 3: unregistered

Alternative function:
int GetAccountRegState(String domain, String proxy, String username, String sipusername);
Parameters domain, proxy, username, sipusername will be used to match the account.

String GetAccountRegStateString(String uri)

Query account register state. Useful if you are using multiple accounts.

With the uri parameter pass the account URI in user@server format (or only the user or the server part if there are no other similar accounts)
Return status text (for example "Registered" if the account is registered to the server).

Alternative function:
String GetAccountRegStateString(String domain, String proxy, String username, String sipusername);
Parameters domain, proxy, username, sipusername will be used to match the account.

Example

```
mysipclient.AddAccount("2222:pwd2@sip.myserver.com"); //register user 2222 with password pwd2 to server sip.myserver.com
mysipclient.AddAccount("sip.myserver.com", "", "3333", "", "", "pwd3", 2, -1, -1); //register user 3333 with password pwd3 to server sip.myserver.com using TLS for the signaling
mysipclient.AddAccount("4444:pwd4@sip2.myserver.com:5070"); //register user 4444 with password pwd4 to server sip2.myserver.com:5070
mysipclient.AddAccount("4444@sip2.myserver.com"); //delete and unregister the 4444 account
mysipclient.SetMainAccount("3333@sip.myserver.com"); //set the 3333 as the main account
Log.i("SIP", mysipclient.GetMainAccount()); //get the main account URI (should return 3333@sip.myserver.com)
Log.i("SIP", mysipclient.ListAccounts(2,"",false, "\r\n")); //list all account URI's
Log.i("SIP", mysipclient.GetAccount("2222@sip.myserver.com")); //get details about the 2222 secondary account
Log.i("SIP", mysipclient.GetAccountRegStateString("2222@sip.myserver.com")); //check the 2222 secondary account status
```

Contacts

Contact management has little to do with the SIP stack and should be handled entirely by your application (except presence status). You can use the above AJVoIP API to add VoIP functionality to your address book or contact list and based on the user presence you can display the different buttons with your design. Near each contact you can display a call/chat button which will launch a voip endpoint instance preconfigured with the actual contact ("callto" parameter). For the contact listing you can just use the native system phone-book or a separate contact list for your app if needed (or a mix of these).

For your convenience as an extra functionality the SDK also provides a simple contact management API which can be also used if for some reason you find it more suited for your needs, but in general we recommend to handle the contacts entirely in your own app since this task is more about user interface related development and less to do with the SIP specific API.

Contact parameters are stored as comma delimited strings with the following parameters:
imstatus,name,sip,phone,phone2,phone3,othernumbers,sipcontacturi,email,web,address,speeddial,extra,internalextra

boolean SetContacts(String contacts)
Set all contacts (contact parameters separated by new line)

String GetContacts()
Will return all contacts (in separated lines the parameters described above)

boolean DelContact(String name)
Delete contact.

boolean AddContact(String params)
Add a contact. Example: `Add_Contact(',John Smith,jsmith,');  //here we set only the name and the SIP fields`

boolean SetContact(String name, String params)
Change contact.

String GetContact(String name)
Will return a single contact in the format described above.

Helper functions to set/get individual fields:

boolean SetContactName(String name, String param)
Set contact name.

boolean SetContactSIP(String name, String param)
Set contact sip uri.

boolean SetContactPhone(String name, String param)
Set contact phone number.

boolean SetContactSpeedDial(String name, String param)
Set contact speed dial number (short number).

String GetContactName(String name)

Get contact name.

String GetContactSIP(String name)
Get contact sip uri.

String GetContactPhone(String name)
Get contact phone number.

String GetContactSpeedDial(String name)
Get contact speed dial number (short number).

Example code for contact management can be found here.

Presence

Presence is based on SIP SIMPLE SUBSCRIBE/NOTIFY mechanism and it is used to detect the online status of the contacts.
There is no need to manage the contacts within SIP endpoint (as described above) to have presence functionality (so you can manage the contacts externally in your application).
The following steps are required:

1. Related settings:
   - enablepresence: 0/1
   - email = email address sent with contact info
   - presenceexpire = 3600
   - autoacceptpresencrequests = -1; //-1: not set (1), 0=auto reject all,1=ask for new users,2=yes, autoaccept new unknown users
2. First you should call PushContactlist to pass all the usernames and phonenumbers from your external contact list if any. This is necessary, because for existing contacts AJVoIP can accept the requests automatically, while for other it might ask for user permission
3. On first start you might call NumExists. If using the Mizu VoIP server, then it will return all existing contacts with SERVERCONTACTS,userlist notification where userlist are populated with the valid users and their online status.
4. Call CheckPresence(userlist). To save softswitch resources, you should carefully select the contacts. (Send only the contacts which are actually used and called numbers. We recommend up to 50 contacts. If the user select a contact, then you can call this function later with that single contact to request its status). *This function will start to send SUBSCRIBE requests.*
5. Use the SetPresenceStatus(statustring) function call to change the user online status with one of the followings strings: Online, Away, DND, Invisible , Offline (case sensitive) . *This function will start to send NOTIFY requests to subscribed parties.*
6. Once these are done, the following notifications can be received from java sip stack:
   NEWUSER,peerusername,displayname,email,URI
   PRESENCE,peerusername,status,displayname,email
   (displayname and email can be empty)

   On newuser, you should ask the user if wish to accept it. If accepted, call the NewUser function. The same function should be called when the user adds a new contact to its contactlist.
   For presence the following status strings are defined (be prepared to receive any of these and handle it with case insensitive by displaying red/green/gray/other icons):
   - Open/Online/Reachable/Available/Call Me/Registered [GREEN]
   - DND (Do not disturb; halt popups and sounds) [RED]
   - Busy/Speaking (can be auto set) [ORANGE/YELLOW]
   - Pending/Forwarding [ORANGE/YELLOW]
   - Away/Idle [ORANGE/YELLOW]
   - Close/Unreachable/Offline/Unregistered [GREY/WHITE]
   - Unknown/Not Set [GREY/WHITE/NOCOLOR]
   - Invisible (no status notifications will be sent) [GREY/WHITE/NOCOLOR]

   Other suggestion for colors:
   - red: busy/dnd
   - bright green: online
   - pale green: away/forwarding/pending
   - white: user exists but unknown status or invisible
   - no color: user doesn't exists / no presence feature

7. You might use the UnSubscribe() API to unsubscribe all endpoints (this includes presence, voicemail and BLF subscribes).

BLF

BLF (Busy Lamp Field) can be used to monitor the state of an extension and it is implemented as SUBSCRIBE/NOTIFY with dialog event package as described in RFC 3265 and RFC 4235.
Set the enableblf parameter to enable/disable BLF. The following values are defined:

0: disable BLF
1: auto (from here it will auto switch to 0 or 2 regarding the circumstances –whether BLF was initiated and succeed/failed)
2: enable BLF
3: force always (if you set to 3 then it can't be switched off later and will use BLF even after failure)

Default value is 1.

To subscribe to other extensions call state, you can set the blfuserlist parameter or use the CheckBLF(userlist) to subscribe to other extension(s) state changes (users separated by comma). To remove an extension, just modify the blfuserlist parameter or call the DisableBLF(userlist) API.

Make sure that the other extension also has BLF support (so it can respond with NOTIFY for the BLF SUBSCRIBE).
Once the state of the remote extension(s) are changed, you will receive BLF notifications in this format:

BLF,peerusername,direction,state,callid

Fields:

- BLF: state header string
- peersusername: extension username
- direction
  - undefined (not for calls or no announced by the peer)
  - initiator (outgoing call)
  - receiver (incoming call)
- state
  - trying (call connect initiated)
  - proceeding (call connecting)
  - early (ringing or session progress)
  - confirmed (call connected)
  - terminated (call disconnected)
  - unknown (unrecognized call state received)
  - failed (BLF subscribe or notify failed)
- callid: optional value if reported from the remote extension (the SIP call-id of the call)

Miscellaneous

Some other not so important API calls are listed below (helper functions):

boolean Test()
You might use this function to check the API availability. Should return true.

boolean ServerInit(String address) -deprecated
Call this function before to start any communication with this address (usually an IP number). This is required to release the Java security restrictions. Wait 1-2 second before calling the next function like Register or Call. This function is deprecated since v.3.8 (no need to call this, just call Register or others directly)

boolean MightStop()
It is a good practice to call this function when your app might be stopped or killed soon. For example your app/activity is pausing/closing/exiting and from there the OS might kill your process. This will just make sure that the internal SIP stack settings are properly saved and the sipstack is prepared for an unexpected termination.

boolean Stop()
Will stop all endpoints. This function call is optional when you unload AJVoIP from external app.

boolean Exit()
Will stop all endpoints and terminates the java sip application. This function call is optional when you unload the AJVoIP java module or wish to issue a forced termination. Its behavior can be controlled by the “exitmethod” parameter.

boolean WaitFor()
Use WaitFor after Stop if you wish to wait a bit for proper shutdown, for example if you wish to make sure that everything was cleaned up properly before to create a new SIP stack.

boolean ReStart()
Restart the SIP stack.
Avoid using this API. A better restart can be done by stopping the old instance (Stop, WaitFor) and creating a new SIPStack object instead.

void CheckConnection()
You might call this function to quickly recover from connection failures on events not know by AJVoIP such as app switched to foreground or activity launch (otherwise AJVoIP should auto-recover from network failures)

boolean CapabilityRequest(String server, String username)
Will send an OPTION request to the server. Usually you should not use this function.
The server parameter can be empty if you already set it with other API calls or by parameter.
The username parameter can be empty (in this case the “From” address will be set to “unknown”)

String LineToCallID(int line)

Get the SIP Call-ID for a line number.

int CallIDToLine(String callid)
Get the line number for a SIP Call-ID.

boolean SetLineParameter(int line, String param, String value, int permanent)
Set parameter for the current/next/all/specific line. Only some of the parameters are supported by this function (listed below).
These are advanced settings and this function might be used only if you need some specific/different behavior for a specific line only.
Otherwise, the behavior of the different lines are loaded from the global configuration (parameters), from the circumstances
or can be influenced by the other API functions (try to use the other API's instead of this when possible).
Parameters:

Line: channel number.
If -2, then it will be set as global config like the SetParameter.
If -1, then for the active line.
If 0, then it will be applied for all next/new channel(s).
Otherwise it will be applied for the specific line.
Param: settings key name
Value: value as string (or "NULL" to clear any old preset)
Permanent
If 1 then the setting will be kept for all future lines with the same line number
If 0 then the setting will be applied only for the current or very next channel with the same line

The following parameters are supported by this function:
peer (called number), address (will set both the serveraddress and proxyaddress), serveraddress, proxyaddress, username, authusername, password, displayname, voicerecording, muted, holded, maxsipmessagesize, codecretry, disccode, registerinterval, maxmsgresend, srtp_suite, strictsrtp, dtmfmode, rtp_timestamp, rtp_seq, rtp_ssrc, defpayload, setfinalcodec, presenceexpiresec, notifyexpiresec, sipproto, volumein, volumeout, volumering, videocalltype, videodirection, keepvideosdponhold, defvideopayload, serverdomainandport, serveraddr, serverport, serverip, proxyport, proxyip, realm, p_referred_identity, p_asserted_identity, remote_party_id, privacy, customsipheader, customsdpfield, customsdpmediafield, sip_uui,sessionid, discreason, usesdpip, transport, line, callid, state, mediaencryption.

All preset settings for all lines can be removed by passing 0 for the line and empty or NULL for both the key and the value.

Examples:

```
mysipclient.SetLineParameter(1,"username", "mia",0); //set the username to mia for the next call on line 1
mysipclient.SetLineParameter(2,"username", "mia",1); //set the username to mia for all upcoming calls on line 2
mysipclient.SetLineParameter(2,"username", "NULL",1); //clear the permanently preset username from line 2
mysipclient.SetLineParameter(-1,"muted", "0",0); //unmute the current call (use the Mute API instead)
mysipclient.SetLineParameter(0,"muted", "4",0); //set the default muted state both side mute for the next call
mysipclient.SetLineParameter(0,"NULL", "NULL",1); //clear all the permanent settings from all lines
```

boolean SetSIPHeader(int line, String hdr)
Set a custom sip header (a line in the SIP signaling) that will be sent with all messages. Can be used for various integration purposes to send auxiliary meta-data via the SIP signaling (for example for sending the http session id).
Example: SetSIPHeader(-1, "X-MyData: VALUE");
Multiple headers can be separated by CRLF (\r\n).
You can also set this with the customsipheader parameter.

String GetSIPHeader(int line, String hdr)
Return a sip header value received by AJVoIP. If not found it will return a string beginning with "ERROR:" such as "ERROR: no such line".

boolean SetUUI(int line, String value, int fortarget)
Set a custom UUI (User-to-User Call Control Information as described in RFC 7433).
Value is the UUI data (with or without parameters)
Fortarget specifies if the UUI have to be sent to target party with call transfer or redirect escaped. 0: no (to peer with the User-to-User header), 1: yes (to target in Contact or Refer-To URI), 2: both.

Example:
SetURI(-1, "anydata",-2);
SetURI(-1, "DATA;encoding=hex;purpose=foo;content=bar",-2);
You can also set this with the sip_uui parameter.

boolean SetSDPField (int line, String field, int type)
Set a custom SDP field (a line in the SDP body) that will be sent with all messages.
Set the type parameter to 0 for global values (before the m= line) or 1 for media values (after the m= line).
Example: SetSDPField(-1, "a=3ge2ae:requested",1);
Multiple headers can be separated by CRLF (\r\n). You can also set this with the customsdpfield and customsdpmediafield parameters.
Line -2 means all lines, -1 means current active line or use 1+ means for an existing line in call.
You can set a SIP header for the next call by using line number -3 or clear the old header(s) by using line number 0.

boolean RTPHeaderExtension (int line, int profile, String extension)
Set RTP header extension for the RTP packets.
The profile will be set as the first two bytes in the extension header (might be used as a custom identifier or parameter).
The extension can be one or multiple numbers separated by semicolon (;). These numbers will be sent as 32 bit words with the RTP header extension payload.
Example: RTPHeaderExtension (-1, 1, "987"); //will set the profile number to 1 and the rtp extension word to 987 for the current call
Line -2 means all lines, -1 means current active line or use 1+ means for an existing line in call.
You can set a SIP header for the next call by using line number -3 or clear the old header(s) by using line number 0.
Set the profile to 0 and the extension string to empty to clear it.

boolean ED137PTT (int line, int ptt, int pttid)
ED-137 Push to talk.
Parameters:
- line: channel number
- ptt: 0: off (not speaking), 1: on (Normal PTT ON), 2: Coupling PTT ON, 3: Priority PTT ON, 4: Emergency PTT ON
- pttid: optional PTT-ID (otherwise loaded from global config)

Will return true if initiated successfully or false on failure.
The ed137 parameter should be set to 1 before using this function. See the ED_137 guide for more details

boolean SendSIPMessage(int line, String msg, String body, String account, String target)
Send a custom SIP signaling message (for example OPTIONS, NOTIFY, etc).
Parameters:
- line: endpoint where the message will be sent. -3: new endpoint, -2: all endpoints, -1: current active, 0: register endpoint, 1+: call channel number
- msg: SIP message to send. It can be just the method name (OPTIONS,MESSAGE,NOTIFY,INVITE,etc), the full message header or the full message including also a body (such as SDP)
- body: optional SIP message body (such as SDP, xml payload or chat text, etc). Set to "null" if body must not be sent.
- account: optional SIP account details if you wish to use other parameters then the configured one.
  same format like the extraregisteraccounts parameter
- target: optional target username, number or SIP URI if not specified in the msg

Examples:
- Send a custom REGISTER request: mysipclient.SendSIPMessage(0, "fullregistermessagehere", "", "");
- Send UPDATE on line 1: mysipclient.SendSIPMessage(1, "UPDATE", "", "");
- Send IM on line 2: mysipclient.SendSIPMessage(2, "MESSAGE", "hi", "", "evelin");
- Send OPTIONS to another server: mysipclient. SendSIPMessage(-3, "OPTIONS", "", "username:password@otherserveraddress.com");
- Send PUBLISH with a custom body: mysipclient.SendSIPMessage(-3, "PUBLISH", "anymessagehere", "");

The Content-Type for INFO, MESSAGE and NOTIFY requests can be specified with the contenttype parameter. Otherwise the content type will be auto guessed.

String GetSIPMessage(int line, int dir, int type)
Return the last received or sent SIP signaling message as raw text.
Dir:
0: in (incoming/received message)
1: out (outgoing/sent message)
Type:
0: any
1: SIP request (such as INVITE, REGISTER, BYE)
2: SIP answer (such as 200 OK, 401 Unauthorized and other response codes)
3: INVITE (the last INVITE received or sent)
4: the last 200 OK (call connect, ok for register or other)

String GetLastRecInvite()
Return the last received INVITE message.

String GetLastRecSIPMessage(String line)
Get the last received SIP message as clear text. Line is the line number or the SIP call id.

String GetLastRecFileName (String line)
Returns the last recorded file name.

String SendChatIsComposing (String line, String number)
Send typing notification.

String GetAddress()
Return the local SIP listener address (IP:port)

boolean IsOnline()
Return true if network is present

boolean IsRegistered()
Return true if AJVoIP is registered (“connected”) to the SIP server.

int IsRegisteredEx()
Returns extended registration state: 0=unknown,1=not needed,2=yes,3=working yes,4=working unknown,5=working no,6=unregistered,7=no (failed)

int IsInCall()
Return whether the sip stack is in call: 0=not in call,1=connecting/setup/ringing,2=connected/speaking

boolean IsIncomingVideo()
Return true if the current incoming call has video offer or false if it has only audio offer

Example code:

```
if(mysipclient.IsIncomingVideo()) //check incoming call type
{
        //auto accept video call
        mysipclient.AcceptVideo(fragmentResId, fragmentinstance);
}
else
{
        //auto accept audio call
        mysipclient.Accept(-1);
}
```

int GetCurrentConnectedCallCount()
Get number of current connected calls

String GetRegFailReason(boolean extended)
Will return a text about the reason of the last failed registration. Set the extended parameter to true to get more details

String GetDiscReasonText()
Return the disconnect reason of the last disconnected call.

int GetAccountRegState(String domain, String proxy, String username, String sipusername, boolean pushnotify, boolean strict)
Query account register state. Useful if you are using multiple accounts.
Returns -1: no such account register ep, 0: working, 1: success, 2: failed, 3: unregistered.
Parameters domain, proxy, username, sipusername will be used to match the account.
If the pushnotify is true, then it will look after push register endpoints (useful if push via gateway).
If the strict parameter is false, then will check the best matching account after the passed parameters.
If the strict parameter is true, then will enforce exact parameter match when checking after the endpoints (server domain/username/etc).

String GetAccountRegStateString(String domain, String proxy, String username, String sipusername, boolean pushnotify, boolean strict)
Query account register state. Useful if you are using multiple accounts.
Return status text (for example "Registered" if the account is registered to the server).
Parameters domain, proxy, username, sipusername will be used to match the account.
If the pushnotify is true, then it will look after push register endpoints (useful if push via gateway).
If the strict parameter is false, then will check the best matching account after the passed parameters.
If the strict parameter is true, then will enforce exact parameter match when checking after the endpoints (server domain/username/etc).

String GetCallerID(int line)
Will return the remote party name

String GetIncomingDisplay(int line)
Get incoming caller id (might return two lines: caller id \n caller name)

String GetLastCallDetails()
Get details about the last finished call.

String GetMySIPURI(boolean all)
Will return the SIP URI on which the current endpoint can be reached such as username@server.com or username@localip:port.
If all is true, then it will return all possible URI’s separated by comma.

void AddLog(String msg)
Add a message to AJVoIP log.

boolean HoldChange(int line)
Same as Hold, but without the second parameter. This call will always invert the hold status for an endpoint (If the call was active, then it will switch to held status and if the call was in hold, then it will reactivate it).

String VAD()
Returns voice activity statistics. See the VAD notification for more details.
Note: if you call this function, VAD will not be sent automatically anymore. (So you will need to continue to poll for the details).

String HTTPGet(String url)
Send a HTTP GET request. Check if return string begins with "ERROR".

Note:
You should never call http requests from your main GUI thread, otherwise it will fail with a NetworkOnMainThreadException exception (This exception will occur only internally, not thrown to you).
For development, you might remove the restriction:
```
StrictMode.ThreadPolicy policy = new StrictMode.ThreadPolicy.Builder().permitAll().build();
StrictMode.setThreadPolicy(policy);
```

boolean HTTPPost(String url, String data)
Send a HTTP POST request.
You should not call this function from your main GUI thread.

String HTTPReq(String url, String data)
Send a HTTP POST or GET request. (If data is empty, then GET will be sent). Can be tunneled. Can block for up to 20 seconds.
You should not call this function from your main GUI thread.

boolean HTTPReqAsync(String url, String data)
Send a HTTP POST or GET request. (If data is empty, then GET will be sent). Can be tunneled. The result will be returned in notifications with "ANSWER" header.
It is safe to call this function also from the main GUI thread.

boolean SaveFile(String filename, String content)
Will save the text file to local disk AJVoIP working directory in encrypted format (use SaveFileRaw to save as-is)

String LoadFile(String filename)
Load file from local disk.

boolean SaveFileRemote(String filename, String content)
Save file to remote storage (preconfigured ftp or http server)

boolean LoadFileRemote(String filename)
Load file from remote storage. The download process is performed asynchronously. You need to call this function only once and then a few seconds later call the LoadFile function with the same file name. It will contain "ERROR: reason" text if the download failed.

String LoadFileRemoteSync(String filename)
Will download the specified file from remote storage synchronously (will block until done or fails).

String GetBlacklist()
Get blacklist (blocklist)

boolean SetBlacklist(String str)
Set whole blacklist (users/numbers separated by comma)

boolean AddToBlacklist(String str)
Add to blacklist

boolean ClearCredentials()
Clear existing user account details.

boolean DelSettings (int level)
Delete settings and data. Levels: 0: nothing, 1: settings, 2: everything

int IsEncrypted ()
Returns whether the connection is encrypted. -1: unknown, 0=no,1=partially/weak yes,2=yes,3=always strong

String GetLogs()
Can be used to get the logs (which are otherwise written also to standard output and to Logcat).
The logpolling parameter must be set to 1 for this to work.
Will store up to 20000 log entries (so you might call this periodically if you don't wish to loose past logs)

String GetBindir()
Returns the application path (folder with the app binaries or app home folder)

String GetWorkdir()
Returns the application working directory (data folder).
Based on:

appcontext.getFilesDir().getPath();
Environment.getDataDirectory().getPath();

String GetAltWorkdir()
Returns the application alternative working directory (such as folder on external SD card).

String GetLogPath()
Will return the log file path.
By default the log file path will be set automatically, usually inside the work directory. It can be modified with the logpath parameter.
Logs are written to file only if the canlogtofile parameter is set to 1 or 2.
The log file might be in use while AJVoIP is running and you will be able to access it only in shared read mode.

int ShouldReset()
Check if the sipstack should be restarted. Usually this is required only on local network change (such as IP change or changing from mobile data to wifi).
Possible return values: 0=no,1=not registered for a while,2=network changed

boolean ShouldResetBeforeCall()
This function might be called before calls and you should quickly restart the sipstack if returns true (then continue to make the call).

boolean RecFiles_Del()
Delete local recorded files.

boolean GetDeviceID()
Returns an unique device ID.

String GetVersion()
Return the program version number.

Most of the API functions were covered above in this guide, however you might find also some additional helper functions exposed. You should be able to cover all your needs using the above functions and most of the additional API's are redundant or needed only in very special circumstances.
See the Javadoc for a complete listing of all public functions.

Notifications

Notifications means events received from the SIP library when something is happening/happened.
To be able to get events from AJVoIP you will need to extend the SIPNotificationListener class and subscribe for the notification events with the SetNotificationListener function call. Then in your NotificatioListener subclass you will receive the notifications as SIPNotification objects or its subclasses.

Example:

```
class MySIPNotificationListener extends SIPNotificationListener
{
  public void onAll(SIPNotification e) {
    Log.v("AJVoIP","Notification received: " + e.toString());
  }
  public void onStatus (SIPNotification.Status e) {
    Log.v("AJVoIP","STATUS notification received: " + e. getStatusText());
  }
}

mysipclient.SetNotificationListener(new MySIPNotificationListener());
```

See the javadoc for a more structured help about the notification objects and their fields.
The source/doc for the API/interface files are also shipped (ajvoip-sources.jar) which you might import in your IDE for better help functionality.
Here is a simple working example code.

For maximum flexibility, the AJVoIP SIP library implements also receiving notifications as strings (instead of SIPNotification objects). In case if somehow you are interested in these, see the details here.

The SIPNotification base class has a toString() function which can be used to convert the event to string. (The string format is better explained here)
Many notifications has a line field to be queried with the getline() function returning the SIP endpoint channel number as described here.
You don't necessarily have to handle all notification events in your SIPNotificationListener subclass. Override only the members you are interested at.

The easiest way to get started is to just log out all messages at first and from there the usage should be obvious.

Below we describe most notifications as strings and field names. See the javadoc for the actual SIPNotification class functions and/or use the auto-complete functionality from your IDE.

The following notifications are defined:

Status

You will receive STATUS notifications when the SIP session state is changed (SIP session state machine changes) or periodically even if there was no any change. Usually you will have to display the call state for the user, and when a call arrives you might have to display an accept/reject button.

Template string: STATUS,line,statustext,peername,localname,endpointtype
A typical status as string looks like this:
STATUS,line,statustext,peername,localname,endpointtype,peerdisplayname,[callid],online,registered,incall,mute,hold,encrypted,video,group,rtpsent,rtprec,rtploss,rtplosspercet,videohold,videosent,videorec,serverstats

Note: not all fields/functions are meaningful for all notifications. For example getRtpsent() should be queried only for endpoints in calls (status between STATUS_CALL_SETUP/ STATUS_CALL_CONNECT and STATUS_CALL_FINISHED) and it is meaningless for STATUS_REGISTER.

SIPNotification.Status methods:
getLine(): The line parameter can be -1 for global status or a positive value for the different lines.
Global status means the state of the "best" endpoint. For example if one line is disconnected and another is in Speaking, then the global state will be Speaking.
You can receive the same STATUS notification multiple times: for the particular endpoint, for the global status and repeatedly if the state remains the same.
You might decide to parse only general status messages (where the line is -1), messages for specific line (where line is a positive number) or both.
getStatus(): The global or endpoint state. One of the following constants: STATUS_UNKNOWN, STATUS_NOTREADY, STATUS_INITIALIZING, STATUS_READY, STATUS_OUTBAND, STATUS_REGISTER, STATUS_REGISTERED, STATUS_REGISTERFAILED, STATUS_UNREGISTER, STATUS_SUBSCRIBE, STATUS_MESSAGE, STATUS_CALL_SETUP, STATUS_CALL_ROUTED, STATUS_CALL_PROGRESS, STATUS_CALL_SESSIONPROGRESS, STATUS_CALL_RINGING, STATUS_CALL_CONNECT, STATUS_CALL_SPEAKING, STATUS_CALL_MIDCALL, STATUS_CALL_MUTE, STATUS_CALL_HOLD, STATUS_CALL_FINISHING, STATUS_CALL_FINISHED, STATUS_DELETABLE, STATUS_ERROR
getStatusText(): the status as string as described here
getPeer(): the other party username (if any)
getLocalname(): the local user name (or username).
getEndpointType(): check if it is a server or client endpoint. Will return one of the following constants:
DIRECTION_UNKNOWN
DIRECTION_OUT: for sessions initiated by AJVoIP such as outbound calls where AJVoIP acts as a client
DIRECTION_IN: for inbound sessions such as incoming calls when AJVoIP acts as a server endpoint
getEndpointTypeText(): endpoint state as string
getPeerDisplayname(): the other party display name if any
getCallID(): SIP session id (SIP call-id)
getOnline(): network state. NETWORK_STATE_UNKNOWN, NETWORK_STATE_OFFLINE (no network or internet connection), NETWORK_STATE_ONLINE (connectivity detected)
getOnlineText(): online state as string
getRegistered(): registration state. REGISTER_STATE_UNKNOWN, REGISTER_STATE_NOTNEEDED, REGISTER_STATE_YES, REGISTER_STATE_WORKING_YES, REGISTER_STATE_WORKING_UNKNOWN, REGISTER_STATE_WORKING_NO, REGISTER_STATE_UNREGISTERED, REGISTER_STATE_FAILED
getRegisteredText(): registration state as string
getIncall(): phone/line is in call. CALL_STATE_UNKNOWN, CALL_STATE_NO, CALL_STATE_CONNECTING, CALL_STATE_CONNECTED
getIncallText(): call state as string
getMute(): is muted state. MUTE_STATE_NO, MUTE_STATE_UNKNOWN, MUTE_STATE_PLAY, MUTE_STATE_REC, MUTE_STATE_BOTH
getMuteText(): mute state as string
getHold(): is on hold state: HOLD_STATE_NO, HOLD_STATE_UNKNOWN, HOLD_STATE_SENDONLY, HOLD_STATE_RECVONLY, HOLD_STATE_BOTH
getHoldText(): hold state as string
getEncrypted(): encryption state: ENCRYPTION_STATE_UNKNOWN, ENCRYPTION_STATE_NO, ENCRYPTION_STATE_YES_WEAK, ENCRYPTION_STATE_YES, ENCRYPTION_STATE_YES_STRONG
getEncryptedText(): encryption state as string
getVideo(): video state: VIDEO_STATE_UNKNOWN, VIDEO_STATE_NO, VIDEO_STATE_INITIALIZED, VIDEO_STATE_OFFERED, VIDEO_STATE_RTPREADY, VIDEO_STATE_STARTED, VIDEO_STATE_STREAMING
getVideodText(): video state as string
getGroup(): group string for group chat and conference calls (members separated by |)
getRtpsent(): number of sent RTP packets (only if endpoint is in call)
getRtprec(): number of received RTP packets (only if endpoint is in call)
getRtploss(): number of lost RTP packets (only if endpoint is in call)
getRtplosspercent(): percent of the lost RTP packets (only if endpoint is in call)
getVideoHold(): video on hold state: HOLD_STATE_NO, HOLD_STATE_UNKNOWN, HOLD_STATE_SENDONLY, HOLD_STATE_RECVONLY, HOLD_STATE_BOTH
getVideoHoldText(): video hold state as string
getVideoRtpsent(): number of sent video RTP packets (only if endpoint is in video call)
getVideoRtprec(): number of received video RTP packets (only if endpoint is in video call)
getServerstats(): RTP statistics received from the server, if any (only if endpoint is in call)

See the SIPNotification.Status in the javadoc for the SIPNotification object details.

STATUS string examples:

The following status means that there is an incoming call ringing from 2222 on the first line:

STATUS,1,Ringing,2222,1111,**2**,Katie,[callid]

The following status means an outgoing call in progress to 2222 on the second line:

STATUS,2,Speaking,2222,1111,**1**,,[callid]

The following status means that the call was disconnected:

STATUS,2,Finished,2222,1111,1,,[callid]

Note:

- *To verify the connection/registration state you might use the below REGISTER notification instead or as described here.*
- *If the "stats" parameter is on (set to a value higher than 0) then you will receive periodic status messages during calls (might be useful if you are interested in RTP statistics during a call).*

Register

This notification is received for register state changes from registrar endpoints.
Registration means connection and authentication on your SIP server.
Template string: REGISTER,line,state,text,main,fcm,user,reason

SIPNotification.Register methods:

getLine(): channel number (should be always 0 for register)
getStatus(): registration state: STATUS_INPROGRESS, STATUS_SUCCESS, STATUS_FAILED, STATUS_UNREGISTERED
getText(): register state as string
getIsMain(): true for primary account, false for secondary registrations (if you are using multiple accounts)
getFcm(): not used (ignore)
getUser(): local username (useful if you are using multiple accounts)
getReason(): failure reason text if any

See the SIPNotification.Register in the javadoc and the How to register FAQ point for more details.

Note: This notification can be disabled by setting the sendregisternotifications parameter to 0 (for compatibility reasons with old versions).

Presence

This notification is received for presence changes (peers online state).
Template string: PRESENCE,peername,state,details,displayname,email

SIPNotification.Presence methods:

getPeer(): username of the peer
getStatus(): presence state. One of the following constants: PRESENCE_UNKNOWN, PRESENCE_OFFLINE, PRESENCE_DND, PRESENCE_AWAY, PRESENCE_BUSY, PRESENCE_ONLINE, PRESENCE_PENDING
getStatusText(): presence state as string
getDetails(): presence details if any
getPeerDisplayname(): peer full name if received (it can be empty)
getEmail(): peer email address if received (it can be empty)

See the SIPNotification.Presence in the javadoc for the SIPNotification object details.
More details about presence handling can be found here.

Notes for the state and detail strings:
The state and the details strings are usually the followings:
CallMe,Available,Open,Pending,Other,CallForward,CallSetup,Speaking,Busy,Idle,DoNotDisturb,DND,Unknown,Away,Offline,Closed,Close,Unreacheable, Unregistered,Invisible,Exists,NotExists,Unknown,Not Set or as reported by the peer
One of these fields might be empty in some circumstances and might not be a string in the above list (especially the details).
The details field will provide a more exact description (for example "Unreachable") while the state field will provide a more exact one (for example "Close"). For this reason if you have a presence control to be changed and you are looking for the strings instead of using the getstatus function, then check the details string first and if you can't recognize its content, then check the state string. For displaying the state as text, you should display the details field (and display the state field only if the details string is empty).

DTMF

Incoming DTMF notification.
Msg is a string parameter representing the incoming DTMF digit. Usually one of the followings: 0123456789*#ABCDF

Supported receive dtmf methods are SIP INFO in SIP signaling and NTE (RFC 2833 in RTP).
Template string: DTMF,line,msg
Example: DTMF,1,7 (dtmf digit"7" received on first line)

SIPNotification.DTMF members:

getLine(): returns the channel number (the endpoint which received the DTMF message)
getMsg(): returns the DTMF message (one or more DTMF digits)

See the SIPNotification.DTMF in the javadoc for the SIPNotification object details.

INFO

Notifications about incoming/outgoing INFO or DTMF messages.

Note: for incoming DTMF you should watch for the above described DTMF notification instead of this! DTMF notification will be triggered even if the message was received with SIP INFO.

Template string: INFO,type,line,peername,text

SIPNotification.INFO members:
getType(): will return one of the following constants:

- INFO_UNKNOWN: this should never be emitted
- INFO_ERROR: Outgoing DTMF (Dtmf()) or INFO (Info()) message failed
- INFO_WARNING: Outgoing DTMF failover from SIP INFO to RFC 2833 or In-Band on answer timeout or error response
- INFO_OK: Outgoing DTMF (Dtmf()) or INFO (Info()) was sent successfully
- INFO_REC: Incoming INFO message received. Note that if the incoming message is a dtmf digit, then a DTMF notification will be triggered instead of INFO

getTypeText(): type as string
getLine(): Phone line
getPeer(): Other party username
getText(): INFO message body when type is REC or any additional info if type us ERROR, WARNING or OK.

See the SIPNotification.INFO in the javadoc for the SIPNotification object details.

Examples:

INFO,ERROR,1,1111,timeout (on Dtmf() or Info() message send failed to 1111 on line 1)
INFO,WARNING,1,1111,failback to rfc2833 (on Dtmf() failback from INFO to rfc2833 or in-band)
INFO,OK,1,1111 (on Dtmf() or Info() message sent successfully to 1111 on line 1)
INFO,REC,1,1111,hello (on SIP INFO message received from 1111 which is not DTMF)

Note:
When sending DTMF in RTP (inband or rfc2833) then it is not possible to get feedback whether the message was actually delivered. In this case you will receive an INFO,OK when the message send was initiated successfully. Otherwise if SIP INFO is used, then you will receive INFO,OK only on 2xx answer from the server or the other peer.

BLF

This notification is received for incoming BLF messages.
Template string: BLF,peerusername,direction,state,callid
Described at the BLF section above.

USSD

This notification is received about incoming USSD messages or report about success/failure about the outgoing USSD messages sent by SendUSSD.
Template string: USSD,line,status,text

SIPNotification.USSD members:

getLine(): the session line
getStatus():

- USSD_UNKNOWN: should never be emitted
- USSD_FAILED: send failed
- USSD_SENT: send succeeded
- USSD_RECEIVED: received

getStatusText(): the status as string
getText(): text is the received USDD string (if status is 2) otherwise it might contain an error (if status is 0)

See the SIPNotification.USSD in the javadoc for the SIPNotification object details.
More details about USSD handling can be found here.

Chat

This notification is received for incoming chat messages.
Template string: CHAT,line,peername,text

SIPNotification.Chat members:

getLine(): used phone line
getPeer(): username of the peer
getText(): the message body as-is
getMsg(): chat text
getGroup(): group name (only for multi-user/group messages, otherwise will return empty string)
getMultiparCurrent(): multipart current fragment (only for 3gpp multipart messages)
getMultiparTotal(): multipart total fragments (only for 3gpp multipart messages)

See the SIPNotification.Chat in the javadoc for the SIPNotification object details.

ChatReport

Chat transmission status report so you can process if outgoing message sent with the SendChat was delivered successfully or failed.
Template string: CHATREPORT,line,peername,status,statustext,group,md5,id

SIPNotification.ChatReport members:

getLine(): used phone line
getPeer(): username of the peer
getStatus(): one of the following constants:
- STATUS_UNKNOWN: should never be emitted
- STATUS_INIT: means chat send initialized
- STATUS_SENDING: means sending in progress
- STATUS_SENT: means successfully sent
- STATUS_FAILED: means failed
- STATUS_QUEUED: means queued for later send

getStatusText(): status text if any (such as delivery error reason if status is 3)
getGroup(): optional parameter for group chat (member names separated by |)
getMD5(): the MD5 checksum of the message text
getID(): message ID (user supplied msgid, 3gpp message reference or generated id)

See the SIPNotification.ChatReport in the javadoc for the SIPNotification object details.

Note:

This notification is sent also for SMS messages.
If the offline messaging is not disabled (offlinechat parameter is not set to 0) then AJVoIP can retry later (on next start and/or when any SIP request is received from the target user) reporting status 4 instead of 3 on failed delivery. The MD5 checksum in this case will be calculated from concatenated pending message text and not for the last message.
A STATUS_QUEUED is reported also if your SIP server responds with 202 (Accepted) instead of 200 (OK). This (the 202 response code) should indicate that the request has been accepted but the processing has not been completed, however some servers (such as Asterisk) might always (incorrectly) answer with 202 for the MESSAGE requests instead of 200. You might set the chatacceptasok parameter to 1 to report status STATUS_SENT for these instead of status STATUS_QUEUED.

ChatComposing

Chat “is composing” notifications are usually emitted when the other party starts or stops typing.
Template string: CHATCOMPOSING,line,peername,status

SIPNotification.ChatComposing members:

getLine(): used phone line
getPeer(): username of the peer
getStatus(): STATUS_TYPING means other party is typing, STATUS_IDLE means other party is idle or stopped typing
getStatusText(): status as string

See the SIPNotification.ChatComposing in the javadoc for the SIPNotification object details.

CDR

After each call, you will receive a CDR (call detail record) with the details about the call.
Template string: CDR,line,peername,caller,called,peeraddress,connecttime,duration,discparty,reasontext,callid,dir,disccode,peerdisplayname

SIPNotification.CDR members:

getLine(): the phone/endpoint line
getPeer(): the other/remote party username, phone number or SIP URI
getCaller(): the caller party name (our username in case when we are initiated the call, otherwise the remote username)
getCalled(): called party name (our username in case when we are receiving the call, otherwise the remote username)
getPeerUsername(): remote party username
getPeerDisplayname(): remote party displayname if any, otherwise the remote party username
getPeerAddress(): remote address (usually the VoIP server IP or domain name)
getConnectTime(): milliseconds elapsed between call initiation and call connect (or until reject/hangup if the call was not connected)
getDuration(): milliseconds elapsed between call connect and hangup (0 for not connected calls. Divide by 1000 to obtain seconds.)
getEndpointType(): check if it is a server or client endpoint. Will return one of the following constants:
- DIRECTION_UNKNOWN
- DIRECTION_OUT: outgoing (client endpoint: the call was initiated by AJVoIP and AJVoIP acts as a UAC)
- DIRECTION_IN: incoming (server endpoint: incoming call when AJVoIP acts as a UAS)

getDiscParty(): the party which was initiated the disconnect:
- DISCBY_NOTSET: not set
- DISCBY_LOCAL: local AJVoIP
- DISCBY_REMOTE: peer
- DISCBY_UNKNOWN: undefined/timeout

getDiscReason(): a text about the reason of the call disconnect (SIP disconnect code, CANCEL, BYE or some other error text)
getReasonCode(): the SIP disconnect reason code received from the peer (-1 if not received; useful for failed calls)

See the SIPNotification.CDR in the javadoc for the SIPNotification object details.

MWI

Voicemail notifications.
Messages are received on new voicemail notifications if you have enabled voicemail and there are pending new messages.
This notification might be set by the SIP server without asking or depending on the voicemail parameter or requested by CheckVoicemail.
Template string: MWI,hasvoicemail,voicemailnumber,to,count,message
Example string: MWI,yes,5001,1111,3,3/0

SIPNotification.MWI methods:

- **getHasMessage()**: “yes” or “no” (Messages-Waiting indicator, whether if there are message(s) pending)
- **getVMNumber()**: the voicemail access number (as configured or as received in Message-Account. see voicemailnum)
- **getTo()**: username from the SIP To header (Message-Account; usually the local account username, useful if you have multiple accounts)
- **getCount()**: number of new pending messages (Messages-Waiting)
- **getMessage()**: message text if sent by the server (Voice-Message)

See the SIPNotification.MWI in the javadoc for the SIPNotification object details.

PlayReady

Audio streaming finished (initiated by PlaySound() to remote peer).
Template string: PLAYREADY,line,callid

SIPNotification.PlayReady members:

getLine(): endpoint channel number
getCallID(): SIP Call-ID

See the SIPNotification.PlayReady in the javadoc for the SIPNotification object details.

Note: this notification is not sent with call disconnect (call disconnect will always terminate the audio streaming).

Vrec

Voice upload status (for voice recording / call recording).
Template string: VREC,line,stage,type,path,reason,source

SIPNotification.Vrec members:

getLine(): channel number *(note: with stage 3 and 4 it will always report -1 or -2 means default/not specified)*
getStatus(): call recording state:

- STAGE_DISABLED: no such state should be reported
- STAGE_INIT: call record begin
- STAGE_SAVING: save begin
- STAGE_SAVED: save success
- STAGE_FAILED: save fail
- STAGE_UNKNWON: no such state should be reported

getStatusText(): state as string
getType(): upload method:

- TYPE_UNKNOWN: unknown
- TYPE_FILE: local file
- TYPE_FTP: ftp
- TYPE_HTTP: http
- TYPE_SERVER: server
- TYPE_SIPREC: siprec

getTypeText(): the type as string
getPath(): upload URL or file path *(note: if stage is 1 then type and path is not reported yet)*
getReason(): failure reason *(if stage is 4)*
getSource(): who initiated the recording: SOURCE_USER, SOURCE_SIPREC, SOURCE_AUTO, SOURCE_UNKNOWN
getSourceText(): the source as text (USER, SIPREC or AUTO)

See the SIPNotification.Vrec in the javadoc for the SIPNotification object and the call recording FAQ for more details.

SIP

Notifications for received/sent SIP signaling messages.
Can be enabled by the sipmsgnotifications parameter (disabled/not sent by default)
Template string: SIP,direction,address,message

SIPNotification.SIP members:

getDirection(): DIRECTION_UNKNOWN, DIRECTION_OUT, DIRECTION_IN
getDirectionText(): “in” (for outgoing messages sent by AJVoIP) or “out” (for incoming messages received by AJVoIP)
getAddress(): peer IP:port
getMessage(): the whole SIP signaling message text

See the SIPNotification.SIP in the javadoc for the SIPNotification object details.

Note: if the message already belongs to a session, then you can get the endpoint line by extracting the Call-ID value and passing it to the CallIDToLine() function.

Account

SIP account related changes, which might be sent depending on the accountnotification parameter. Possible values for the accountnotification parameter:

- 0: never send
- 1: auto (yes, if there are multiple accounts configured and the autoswitchmainacc parameter is not 0, otherwise not sent)
- 2: always send

Template string: ACCOUNT,line,change,uri,display

SIPNotification.Account members:

getLine(): -1 for main account change, 1+ for call endpoint accounts
getChange(): returns what change happened. ACCOUNT_CHANGE_MAIN (This is the only change type supported for now. We might add others later if it will be required)
getAccount(): get the account URI
getDisplay(): account shortest unique display string (might return only the username or only the domain part if there are no other accounts set with the same username/domain)

See the Accounts chapter for more details about SIP accounts handling.

Block

This notification is emitted when AJVoIP blocks a signaling message only if the sendblocknotifications parameter is set to 1.
Template string: BLOCK,type,message

SIPNotification.Block members:

getType(): TYPE_UNKNOWN, TYPE_BLACKLIST, TYPE_WHITELIST
getTypeText(): the type as string
getMessage(): the SIP message being blocked / ignored

See the SIPNotification.Block in the javadoc for the SIPNotification object details.

VAD

Voice Activity Detection.
This is sent in around every 3000 milliseconds (3 seconds) by default (configurable with the vadstat_ival parameter in milliseconds) if you set the "vadstat" parameter to 3 or 4 it can be requested with the VAD() function. Also make sure that the "vad" parameter is set to at least "2".
This notification can be used to detect speaking/silence or to display a visual voice activity indicator.

Template string: VAD,parameters

String format:
VAD,local_vad: ON local_avg: 0 local_max: 0 local_speaking: no remote_vad: ON remote_avg: 0 remote_max: 0 remote_speaking: no

If the vadbyline parameter is set to 1 then individual lines will report in the following format:
VAD,line: X remote_vad: ON remote_avg: 0 remote_max: 0 remote_speaking: no

SIPNotification.VAD members:
getLine(): endpoint channel number
getParameters(): returns all parameters

getLocalValid(): whether VAD is measured for microphone: ON or OFF
getLocalAvg(): average signal level from microphone
getLocalMax(): maximum signal level from microphone
getLocalSpeaking(): local user speak detected: yes or no

getRemoteValid(): whether VAD is measured from peer to speaker out: ON or OFF
getRemoteAvg(): average signal level from peer to speaker out
getRemoteMax(): maximum signal level from peer to speaker out
getRemoteSpeaking(): peer user speak detected: yes or no

See the SIPNotification.VAD in the javadoc for the SIPNotification object details.

More details about VAD handling can be found here.

RTPE

Reports about RTP extra header changes.
This is sent only if there RTP header extensions are received and the rtpextraheadernotify parameter is set to 1.
Template string: RTPE,line,profile,extension
Example: RTPE,1,1,98;76543

SIPNotification.RTPE members:
getLine(): the session endpoint line number
getProfile(): identifier or parameter extracted from the first two byte of the extension header
getExtension(): 32 bit words as int converted to string. If there are more then one then separated by semicolon (;)

See the SIPNotification.RTPE in the javadoc for the SIPNotification object details.

RTPStat

Media quality reports.
Can be enabled by the rtpstat parameter.
The parameters are calculated since the last RTPSTAT notification (for the past x seconds).
Template string: RTPSTAT,quality,sent,rec,issues,loss

SIPNotification.RTPStat members:

- **getQuality()**: quality points between 0 and 5. The calculations considers many factors such as RTP issues, RTCP reports, codec problems, packet loss, packet delay, jitter and processing delay.
  The following values are defined:
  - QUALITY_UNKNOWN: unrecognizable (-1)
  - QUALITY_NO: no audio or non-recognizable voice (0)

    - QUALITY_LOWEST: very bad quality (1)
    - QUALITY_LOW: bad quality (2)
    - QUALITY_MEDIUM: medium quality (3)
    - QUALITY_HIGH: good quality (4)
    - QUALITY_HIGHEST: excellent quality (5)
- **getQualityText()**: the quality as string
- **getSent()**: RTP packets sent
- **getRec()**: RTP packets received and played
- **getIssues()**: number of issues (any issues are counted, such as sequence number mismatch or packet drop)
- **getLoss()**: lost packets

See the SIPNotification.RTPStat in the javadoc for the SIPNotification object details.

More details here.

Group

Sent for conference calls. You might update the displayed peer name with the peers strings received here.
Template string: GROUP,line,peers

SIPNotification.Group members:

getLine(): endpoint channel number
getPeers(): list of members separated by |. For example: Kate | John | Linda

See the SIPNotification.Group in the javadoc for the SIPNotification object details.

Start

This message is sent immediately after startup (so from here you can also know that the SIP engine was started successfully).
Template string: START,what

SIPNotification.Start members:

getWhat(): START_API or START_SIP
getWhatText(): the what value as string:
"api" -api is ready to use
"sip" –sipstack was started

See the SIPNotification.Start in the javadoc for the SIPNotification object details.

Stop

This message is sent when the SIP stack is terminated/destroyed.
Template string: STOP,api

SIPNotification.Stop members:

getWhat(): STOP_API or STOP_SIP
getWhatText(): the what value as string:
"api" -api is not available anymore
"sip" –sipstack was stopped (might not be emitted)

See the SIPNotification.Stop in the javadoc for the SIPNotification object details.

ShouldReset

You should restart AJVoIP (or re-init the library) when you receive this notification.
This is usually sent when network was changed (connection type, IP address), thus it might be best to reinitialize the whole SIP stack to recalculate the optimized environment variables and perform the auto network discovery process again (such as STUN lookup, but there are many others).
The message is not sent while in call.
Template string: SHOULDRESET,reason text

SIPNotification.ShouldReset members:

getReason(): reason text

See the SIPNotification.ShouldReset in the javadoc for the SIPNotification object details.

Line

This message is sent when the active line is changed (the line parameter is the current active line).
Template string: LINE,line

SIPNotification.Line members:

getLine(): the new active line number

See the SIPNotification.Line in the javadoc for the SIPNotification object details.

Event

Important events which should be displayed for the user.
Template string: EVENT,TYPE,txt
Example: EVENT,EVENT,any text

SIPNotification.Event members:

getType(): the event type: TYPE_EVENT, TYPE_WARNING, TYPE_ERROR, TYPE_CRITICAL, TYPE_UNKNOWN
getTypeText(): the type as string: EVENT, WARNING, ERROR
getText(): the message string

See the SIPNotification.Event in the javadoc for the SIPNotification object details.

Note: These messages will not be received if you set the "events" parameter below 2 (default is 2) or the loglevel parameter below 1.

Popup

Should be displayed for the users in some way.
Its name is a bit misleading. You should not display a popup or modal dialog on these notifications, just some hint/toast/snackbar.
Template string: POPUP,txt

SIPNotification.Popup members:

getText(): the message string

See the SIPNotification.Popup in the javadoc for the SIPNotification object details.

Example display: Toast.makeText(appcontext, txt, Toast.LENGTH_LONG).show();

Log

Detailed logs (may include also the SIP signaling).
Template string: LOG,TYPE,txt

SIPNotification.Log members:

getType(): the log type: TYPE_EVENT, TYPE_WARNING, TYPE_ERROR, TYPE_CRITICAL, TYPE_UNKNOWN
getTypeText(): the type as string: EVENT, WARNING, ERROR, CRITICAL, RTP
getText(): the log string

See the SIPNotification.Log in the javadoc for the SIPNotification object details.

Note: These messages will be received only if you set the events parameter to 3 and also depends on the loglevel.

With RTP you might receive the media stream details at the end of the calls.
Example: RTP: sent 15695 lasts: 0 (p2p: 0 sdp: 0 rrp: 15695 tnl: 15701), rec: 17281 lastr: 0, loss: 201 1%, vsent: 0/0, vrecv: 0/0, cpu: 0.0%, cpurel: 0.0% (0.0), srvsent: 10326 srvrec: 10302 srvloss: 10 0%

Other notifications

Some other rarely required/used notifications are described below:
Format: messageheader, messagetext.

"**CREDIT**" messages are received with the user balance status if the server is sending such messages.

Example: “CREDIT,Credit: 2 USD”
See the SIPNotification.Credit in the javadoc for the SIPNotification object details.

“**RATING**” messages are received on call setup with the current call cost (tariff) or maximum call duration if the server is sending such messages.
Example: “RATING,15 cents/min”
See the SIPNotification.Rating in the javadoc for the SIPNotification object details.

“**NEWUSER**” new contact request
Example: NEWUSER,usrname,displayname
See the SIPNotification.NewContact in the javadoc for the SIPNotification object details.

“**SERVERCONTACTS**” contact found at local VoIP server (only by supported servers)
Example: SERVERCONTACTS,details
See the SIPNotification.ServerContacts in the javadoc for the SIPNotification object details.

“**ANSWER**” answer for previous request (usually http requests)
Example: ANSWER,RESULT:the_answer, REQUEST:the_request,EOFANSWER
See the SIPNotification.Answer in the javadoc for the SIPNotification object details.

“**APIRESULT**” API function return value, which might be useful if you call the API via socket or via broadcast intents.
These notifications are sent only if the apiresultnotifications parameter is set to 1.
Template: APIRESULT,requestid,functionname,returnvalue
Example: APIRESULT,-1,api_call,true
Class name: SIPNotification.APIResult

Example session

Here is how the notifications looks like in a typical session (start, register, make a call, hangup, terminate):

```
START,api
START,sip
STATUS,-1,Initialized
STATUS,-1,Register...
EVENT,EVENT,Connecting...
STATUS,-1,Registering...
STATUS,0,Registering,istvantest2,istvantest2,1,voip.mizu-voip.com,[2e112329450424958431 1k49333rmwp],,1,4,0,0,0,0,0
CREDIT,Credit: 4.2 USD
STATUS,-1,Registered.
EVENT,EVENT,Authenticated successfully. [ep: 0 2e112329450424958431 1k49333rmwp Registering 12538]
STATUS,0,Registered,istvantest2,istvantest2,1,voip.mizu-voip.com,[2e112329450424958431 1k49333rmwp],,1,2,0,0,0,0,0
STATUS,-1,Starting call to testivr3
EVENT,EVENT,call [wpmain]
STATUS,-1,Call
STATUS,-1,Call Initiated
STATUS,1,CallSetup,testivr3,istvantest2,1,testivr3,0,0,0,100,,[6e4351661777543861707k49334rmwp],1,2,1,0,0,0,0
STATUS,1,Routed,testivr3,istvantest2,1,testivr3,[6e4351661777543861707k49334rmwp],,1,2,1,0,0,0,0
STATUS,1,InProgress,testivr3,istvantest2,1,testivr3,[6e4351661777543861707k49334rmwp],,1,2,1,0,0,0,0
STATUS,-1,Calling...
STATUS,1,Ringing,testivr3,istvantest2,1,testivr3,[6e4351661777543861707k49334rmwp],,1,2,1,0,0,0,0
STATUS,-1,Ringing...
STATUS,1,CallConnect,testivr3,istvantest2,1,testivr3,0,0,0,100,,[6e43516617775438617 07k49334rmwp],1,2,2,0,0,0,0
STATUS,1,Speaking,testivr3,istvantest2,1,testivr3,[6e4351661777543861707k49334rmwp],,1,2,2,0,0,0,0
STATUS,-1,Speaking (0 sec)
STATUS,-1,Hangup
STATUS,-1,Speaking (2 sec)
EVENT,EVENT,hangup [wpmain]
STATUS,1,CallDisconnect,testivr3,istvantest2,1,testivr3,98,98,0,100,,[6e4351661777543861707k49334rmwp],1,2,0,0,0,0,0
STATUS,-1,Call Finished
CDR,1,testivr3,istvantest2,testivr3,voip.mizu-voip.com,1499,1969,1,User Hung Up (exD),[6e4351661777543861707k49334rmwp]
STATUS,1,Finishing,testivr3,istvantest2,1,testivr3,[6e4351661777543861707k49334rmwp],,1,2,0,0,0,0,0
EVENT,EVENT,Call duration: 2 sec [ep: 1 6e4351661777543861707k49334rmwp Finishing 12538]
STATUS,1,Finished,testivr3,istvantest2,1,testivr3,[6e4351661777543861707k49334rmwp],,1,2,0,0,0,0,0
STATUS,-1,Registered.
STOP,api
```

Parameters

The Android SDK is very flexible and you have a long list of parameters that you can use, however this doesn't mean that the usage is difficult. You can ignore most of these them and use only those few relevant for your needs also outlined in the usage example. Usually you only have to pass the basic parameters and change the advanced parameters only if really required for you specific use-case.

Parameters can be specified with the SetParameter or SetParameters API calls.
Example: mysipclient.SetParameter("username","sdktest2");
All parameters can be passed as strings and will be converted to the proper type internally by the Android SIP library .

Note:

- *Once a parameter is set, it might be cached by AJVoIP and used even if you remove it later. To prevent this, always set all the parameters to their value if you changed it before. Strings can be set to "DEF" or "NULL". So instead of just deleting, set its value to "DEF" or "NULL". "DEF" means that it will use the parameter default value if any. "NULL" means empty for strings, otherwise the parameter default value. Example: transport=DEF*

- *Not all parameters can be changed at runtime. It is recommended to set all the relevant parameters (with the SetParameter or SetParameters functions) before to call the Start function. You can just restart the sip stack if you need reconfiguration at runtime (stop the current AJVoIP instance and create a new one).*

Basic Parameters

These are the most important parameters.

serveraddress

(string)
The domain name or IP address of your SIP server. By default it uses the standard SIP port (5060). If you need to connect to other port, you can append the port after the address separated by colon.
Examples:

"mydomain.com" -this will use the default SIP port: 5060
"sip.mydomain.com:5062"
"10.20.30.40:5065"

Default value is empty.

If not set, then you (or the users) will be able to call only using full SIP URI and it is more difficult to accept incoming calls. If set, then any username/phone number can be called what is accepted by the server.
As the SIP server you can use any softswitch, IP-PBX, SIP proxy server or SIP gateway (with or without registration). Some SIP services requires the use of an outbound proxy.

username

(string)
This is the SIP username (A number/Caller-ID for the outgoing calls). The AJVoIP SIP endpoint will authenticate with this username on your SIP server.
When compact is true, then this parameter must be filled properly. Otherwise it can be empty or omitted so the users will have to type it.
Default value is empty.

Note: If you need a different name for SIP user name and Auth name (authorization name) then you can use the sipusername parameter.

password

(string)
SIP authentication password.
If your server doesn't require digest authentication or if you need only peer to peer calls then this parameter can be omitted.
Default value is empty.

Advanced Parameters

These parameters are more rarely used and should be set only if you have a specific need or a good VoIP/Android knowledge. You should modify only those parameters which you fully understand, otherwise better if you leave all with the default values (the default values are already optimized for production).

sipusername

(string)
Specify default SIP username. Otherwise the "username" parameter will be used for both the username and the authentication name.
If this is not specified, then the "username" will be used for the From field and also for the authentication.
If both username and sipusername is set then
-the username will be used in the From and Contact fields (CLI/caller-id)
-the sipusername will be used for authentication only

Default is empty.

displayname

(string)
Specify default display name used in “from” and “contact” headers (Enduser full name).
Default is empty (the “username” field will be displayed for the peers)

transport

(int)
Transport protocol for the SIP signaling
-1: auto (auto guess)
0: UDP (User Datagram Protocol. The most commonly used transport for SIP)
1: TCP (signaling via TCP. RTP will remain on UDP)
2: TLS (encrypted signaling)
3: HTTP tunneling (both signaling and media. Supported only by mizu server or mizu tunnel)
4: HTTP proxy connect (requires tunnel server)
5: Auto (automatic failover from UDP to HTTP if needed)
Default is -1.

TLS/SIPS notes:
To enable full encryption set both the transport and the mediaencryption parameters to 2.
If you set TLS (2), then make sure that you are using the correct serveraddress parameter. SIP servers usually listens for TLS on port 5061, thus the serveraddress should look like yourdomain:5061.
If the SIP port is configured to 5061 and the transport parameter is not set (-1), then it might default to 2 (TLS).
TLS 1.3 enabled by default from API level 29.
If you need strict TLS certificate validations (not only encryption) then you might set the tlspolicy parameter to 3 (0: def-auto, 1: allows self signed or invalid cert, 2: medium with warning if fails but continue, 3: with server cert validation and disconnect if not secure not allowing failbacks, 4: perform also domain validation).
Client side certificate can be also set (ask mizutech support if you need this feature).
There is also a SetSSLContext function if you wish to set your custom SSLContext.

mediaencryption

(int)
Media encryption method (encryption for the RTP streams)
-1: auto (will try SRTP if transport is TLS)
0: not encrypted (default)
1: auto (will encrypt if initiated by other party)
2: SRTP
3: ZRTP (experimental)
Default is -1.

Note: To enable full encryption set both the transport and the mediaencryption parameter to 2.

strictsrtp

(int)
Media encryption method
0: most compatible
1: default auto
2: strict (might disconnect if peer is not respect the standard or on protocol error)
3: allow only strict SRTP call, otherwise disconnect
Default: 1

inet

(int)
Internet protocol (IP layer).
Specify if you wish to prefer IPv4 or IPv6.
Possible values:
-1: Auto

0: IPv4 only (disable IPv6)
1: IPv4 (parse also IPv6)
2: Both (prefer IPv4)
3: Both (no preference)
4: Both (prefer IPv6)
5: IPv6 only (disable IPv4)

Default is -1/Auto (auto guessed from configurations such as serveraddress and from the SIP signaling / SDP).

Other IP related parameters (all are set based on this inet parameter, but can be set also separately if you have some special requirement):
hasipv4 (true/false), hasipv6 (true/false), listenonipv6 (true/false), ipvdualstack (true/false), cropipv6scopeid (true/false), preferipv (-1: auto (def to 1), 0: no preference, 1: prefer ipv4, 2: prefer ipv6)

proxyaddress

(string)
Outbound proxy address (Examples: mydomain.com, mydomain.com:5065, 10.20.30.40:5065)
Leave it empty if you don't have a stateless proxy. (Use only the serveraddress parameter)
Default value is empty.

md5

(string)
Instead of using the password parameter you can pass an MD5 checksum for better protection: MD5(username:realm:password)
(The parameters are separated with the ':' character)
The realm is usually your server domain name or IP address (otherwise it is set on your server)
If you are not sure, you can find out the realm in the "Authenticate" headers sent by your server with the" 401 Unauthorized" messages. Example:
WWW-Authenticate: Digest realm="YOURREALM", nonce="xxx", stale=FALSE, algorithm=MD5
Default is empty.

realm

(string)
You might set this if your server realm (logical SIP domain which is usually used for auth) is not the same with the "serveraddress" parameter.
If the "md5" parameter was set, then this must match with the realm used to calculate the md5 checksum.
Default is empty, which means that the "serveraddress" will be used or as received from the server.

register

(int)
With this parameter you can set whether the SIP UA should register (connect) to the sip server.
0: no
1: auto guess (yes if username/password is set, otherwise no)
2: yes

AJVoIP will also reregister automatically based on the "registerinterval" parameter.
Default value is 1.

registerinterval

(int)
Registration interval in **seconds** (used by the re-registration expires timer).
Default value is -1 (auto guess)

Valid range is -1 (auto) or between 10 and 30000.

Notes:
- *Many servers will not accept values above 3600 (although the SIP stack can be auto adjust it based on server negotiation)*
- *The actual resend of the REGISTER messages might be done at a shorter interval to cover any potential packet loss and network/server delays.*
- *When set to auto guess (-1), it will calculate an optimal interval depending on the circumstances such as network type, server load, transport method, data saver settings and others. It will result to around 90-180 under normal conditions on UDP or 600 on TCP.*
- *If your softswitch supports keep-alive messages (to prevent NAT binding timeouts), then you might set to a longer interval (~3600 sec) to prevent high CPU usage on your server especially if you have many hundreds of SIP UA running at the same time. If your server doesn't support keep-alive, then you might set this to a lower value (between 30 and 90 sec. 60 sec is a good choice for most NAT devices and routers). Note that usually this is not necessary because server side support is not needed to keep the NAT bindings.*
- *If the register expire interval is not accepted by the server, then the SIP stack is capable to automatically negotiate a new value as directed by the server Min-Expire header or auto-guess.*
- *The register expires might be auto set to a higher value if push notifications are enabled to reduce the need of push re-register/rebind requests.*

needunregister

(boolean)
Set to false to prevent unregister messages to be sent by AJVoIP (for example to prevent unregister on web page reload).
Default is true

unregall

(int)
Set to 1 to unregister all endpoints for the users, not only the current one.
Will send Contact: * with the unregister requests (REGISTER with Expires: 0).
Default is 0.

unregonstart

(int)
Set to 1 to unregister at startup (before to register)
Default is 0.

setinstanceid

(int)
Set to 1 to enable RFC 5626 behavior (connection flows), sending reg-id, +sip.instance and the ob parameter with the SIP Contact header.
Possible values:
-1: Auto (will not send with UDP transport; will send with TCP or TLS)
0: No
1: Yes
Default is -1 (auto).

contactaddressfailback

(int)
Specify if to try to reregister with another (better) local address if register fail or if previous (first) register went with a private address (and the server is on public IP).
Possible values:

- -1: auto (same as 2 if server is on public IP and stun/rport are not disabled and server is not known NAT friendly; otherwise same as 1)
- 0: never
- 1: if failed (will try to reregister with other local address on register failure)
- 2: always (also try to reregister with public address when possible which helps for servers with poor NAT support)

Default: -1

Note:
With the unbindbeforereregister parameter you can also specify if to unbind (unregister) the previous (private) address before registering the new one.
This might be forced on servers with no or poor call fork support, otherwise most servers should be able to just replace the previous address without the need to unregister.
Possible values: -1: auto, 0: never, 1: always (default).

extraregisteraccounts

(string)
Use this setting for multi-account registration.
You can specify one or multiple (up to 99) extra SIP accounts as SIP URI's or in the following format (parameters separated by comma, accounts separated by semicolon ;):
IP,usr,pwd,t,proxy,realm,authuser,displayname,transport;IP2,usr2,pwd2;IP3,usr3,pwd3, , ,realm3;IP4,usr4,pwd4…
where:

- IP: is the SIP server address (domain or IP:port)
- usr: is the SIP username
- pwd: is the SIP password
- t: is the register expire timeout in seconds
- proxy: SIP proxy
- realm: SIP realm
- authusr: auth (sipusername if separate extension id and authorization username have to be used)
- displayname
- transport (default/empty, UDP, TCP or TLS)

- main (-1: auto, 0: no/secondary, 1: yes/main/primary)
- fcm (ignore or set to empty or false)
- enabled (true if account is enabled, false if disabled. You might just always set to true or empty)

The Username parameter is mandatory, all the others are optional. AJVoIP will use the defaults (if any) for the empty parameters.
Multiple accounts can be passed at once separated by semicolons (;).
An account must be specified as a SIP URI or as the following parameters separated by comma (,) :

Examples:
user:pwd@domain:port //a single extra account as a SIP URI
sip:john:secret@myserver.com:5060 // a single extra account as a SIP URI
"John Smith" <john:secret@myserver.com:5060> //a single extra account as a SIP URI specifying also the displayname
myserver.com,john,secret,180 //a single extra account as parameters
myserver.com:5061,john,secret,180, , , , ,TLS; //add a single extra account with TLS transport
john:jsecret@192.168.1.50:5060;kate:ksecret@192.168.1.50:5060 //multiple accounts as SIP URI's
92.168.1.50:5060,john,jsecret,180;92.168.1.50:5060,kate,ksecret;92.168.1.50:5060,mary,msecret,600 //multiple accounts as parameters with/without register interval

See the Accounts chapter and the Extra accounts format FAQ for more details.

autoswitchmainacc

(int)
Allow automatic main account switch by AJVoIP.
Possible values:
0: never (AJVoIP will not change the main account automatically)
1: yes for calls if the call URI match another account or if the current main account failed to register and the main account was not explicitly set before (if the SetMainAccount was not used previously for at least 3/6 seconds)
2: yes also on main account register failed but secondary succeed (in this case AJVoIP will switch to the "best" account if the SetMainAccount was not used previously for at least 12 seconds)
3: always, without checking if it was set explicitly (even if you used the SetMainAccount function)
Default is 2

See the Accounts chapter for more details.

sendregexpire

(int)
Weather to send and how to sent the expires with the register requests:
-1: auto (usually defaults to 1/global)
0: no (don't sent and expires value, except 0 for unregisters)
1: send in a separate global Expires header
2: send with the contact expires parameter
3: reserved (don't use)
4: both (send in both the contact and separate Expires header)
Default value is -1.

You might also set the sendregistrationevent parameter to true if "Event: registration" should be also sent with the register requests.

acceptsrvexpire

(int)
Accept the expires interval sent by the server.
0: no
1: yes (prioritize the contact expire)
2: yes (prioritize the global expire)
Default value is 1.

keepaliveival

(int)
NAT keep-alive packet send interval in milliseconds.

Keep-alive is a mechanism to keep the NAT open between register resends (so the server can initiate a new incoming transaction on the same stream with no issues, such as incoming call).

Possible values:
- -1: auto (will default to around 28000 -28 sec- on UDP and 600000 – 10 min- on TCP)
- 0: disabled
- other: interval in milliseconds (must be above 3000, otherwise treated as seconds)

Default value is -1.

keepalivetype

(int)
NAT keep-alive packet type.
0=no keep-alive
1=space + CRLF (_\r\n) (very efficient because low bandwidth and low server usage)
2=NOTIFY (standard method)
3=CRLF (\r\n) (very efficient because low bandwidth and low server usage. Not recommended)
4=CRLFCRLF (\r\n\r\n) (very efficient because low bandwidth and low server usage. After RFC draft)

Default is 4.

autocall

(boolean)
If set to true then the Android SIP library will immediately starts the call with the given parameters (for example with your page load). The serveraddress, username, password, callto must be also set for this to work.
Default value is false.

callto

(string)
Can be any phone number/username that can be accepted by your server or a SIP URI. Used with “autocall” to specify the destination number.
Default value is empty.

use_rport

(int)
Check rport in SIP signaling (requested and received from the SIP server by the VIA header)
0=don’t ask (rtport request will not be added to the VIA header)
1=use only for symmetric NAT (only when it is sure that the public address will be correct)
2=always (always request and use the returned value except if already on public ip)
3=request even on public IP (meaningless in most cases)
9=request with the signaling, but don’t use the returned value (good If you want to keep the local IP and for peer to peer calls)
Change to 0 or 2 only if you have NAT issues (depending on your server type and settings)
(You might adjust also the use_fast_stun parameter if you change the use_rport)
Default is 1

upnpnat

(int)
Nat traversal via UpNP supported routers.
0: disable
1: enable
Default is 1

use_fast_ice

(int)
Fast ICE negotiations (for p2p rtp routing):
0=no (set to 0 only if your server needs to always route the media)
1=auto
2=yes
3=always (not recommended)

Default is 1
Note: if set to 1 or 2 then the stun should not be disabled

use_fast_stun

(int)
Fast stun request on startup.
-1=force private address (if the client has both private and public IP, than the private IP will be sent in the signaling)
0=no
1=use only for stable symmetric NAT
2=use only if both tests match even if not symmetric (recommended)
3=use for symmetric NAT even if only one match
4=always
5=use even on public IP
Change if you have NAT issues (depending on your server type and settings)
(You might adjust also the use_rport parameter if use_fast_stun is changed)
Default is 2

udpconnect

(int)
Specify whether the UDP have to be connected before sending on the socket. (Some IP-PBX might require udp connect and in this way the Android SIP library can always detect its local address correctly. However this should not be used whit multiple servers or separate domain and outbound proxy)
0=no
1=on init
2=on first send (not recommended. can block)
3=on both or any (not recommended)
Default value: 0

changesptoring

(int)
If to treat session progress (183) responses as ringing (180). This is useful because some IP-PBX never sends the ringing message, only a session progress and might start to send in-band ringing (or some announcement)
The following values are defined:
0: do nothing,
1: change status to ring
2: start local ring and be ready to accept media (which is usually a ringtone or announcement)
3: start media receive and playback (and media recording if the “earlymedia” parameter is set)
4: change status to ringing and start media receive and playback (and media recording if the “earlymedia” parameter is set to true)
Default value is 2.

*Note: on ringing status AJVoIP is able to generate local ringtone. However this locally generated ringtone playback is stopped immediately when media is started to be received from the server (allowing the user to hear the server ringback tone or announcements)

allowcallredirect

(int)
Set to 1 to auto-redial on 301/302 call forward.
Set to 0 to disable auto call forward.
Default value is 1.

natopenpackets

(int)
Change this option only if you have RTP setup issues with your server(s).
UDP packets to send to open the NAT device and initiate the RTP. Some servers will require at least 5 packets before starting to send the media after the 183 “session in progress” response. In this case set this value to 10 (In this way the server will receive at least 5 packets even on high packet loss networks)
-1: auto
0: no
1: yes (on packet for all possible peer address IP:port)
2+: yes, multiple (write this number of packets)

Default is -1 which will default to 0 if AJVoIP is on public IP and to 1 if it is on private IP

*Note: instead of sending more “fake” packets, you can set the “earlymedia” to 1 or more to begin the rtp stream immediately.

*Note: you can use the “natopenpackettype” to specify the format. -1 means auto, 1 means short CRLFCRLF packet, 2 means full RTP packet with zeroed content.

earlymedia

(int)
Start to send media when session progress is received.
0: no
1: reserved
2: auto (will early open audio if wideband is enabled to check if supported)
3: just early open the audio
4: null packets only when sdp received
5: yes when sdp received
6: always forced yes
Default is 2.

*Note: For the early media to work, AJVoIP has to open the NAT when SDP is received. This can be done by sending a few fake rtp packets or by starting to send the media immediately when session in progress is received. The first method consume less bandwidth, but it is not supported by some softswitch.

usehttpproxy

(int)
Used only for HTTP tunneling with Mizu VoIP servers.
0: no
1: same as sip proxy (proxyaddress)
2: system default
3: manual (must be set by the httpproxyurl parameter –deprecated after version 3.5)
4: auto
Default value is 4.

httpserveraddress

(string)
Useful when the transport parameter is set to 4 (auto) to specify the http tunneling gateway address.
Default value is null (address loaded from the “serveraddress” parameter)

dtmfmode

(int)
DTMF send method
0=disabled
1=SIP INFO method (out-of-band in SIP signaling INFO messages)
2=auto (auto guess from peer advertised capabilities)
3=INFO + NTE
4=NTE (Named Telephone Events as specified in RFC 2833 and RFC 4733)
5=In-Band (DTMF audio tones in the RTP stream)
6=INFO + InBand

Default is 2 (and you should change it only if you have a good reason to do so, since the auto-detect should work in all circumstances, except if your SIP server is sending bogus DTMF capabilities).

Note: When more than one method is used (dtmfmode 3 or 6), the receiver might receive duplicated dtmf digits.

sendearlydtmf

(int)
Specify whether to allow sending DTMF digits before call connect
0=no
1=auto (yes if rfc2833 or inband is allowed and already sent/received rtp packets)
2=yes (always send)

Default: 1

voicemail

(int)
Subscribe to voicemail notifications (MWI). Accepted values:
0=disabled
1=display voicemail only if NOTIFY is received automatically without subscription
2=auto-detect if voicemail SUBSCRIBE is needed
3=subscribe for voicemail messages after successful registration
4=subscribe for voicemail messages on startup
Default value is 2. Set to 3 if your server has support for MWI to be sure that AJVoIP will check the voicemail.

For voicemail you can receive MWI notifications in the following format:
MWI,hasvoicemail,voicemailnumber,to,count,message

- hasvoicemail: yes or no (Messages-Waiting indicator)
- voicemailnumber: voicemail number (as configured or as received in Message-Account)
- to: username from the SIP To header (usually the local account username, useful if you have multiple accounts)
- count: number of pending messages
- message: message text if sent by the server (Voice-Message)

Example: MWI,yes,5001,1111,3,3/0

voicemailnum

(String)
Specify the voicemail address. Most IP-PBX will automatically send the voicemail access number so you don't need to set this parameter.

transfertype

(int)
-1=default transfer type (same as 6)
0=call transfer is disabled
1=transfer immediately and disconnect with the A user when the Transf button is pressed and the number entered (unattended transfer)
2=transfer the call only when the second party is disconnected (attended transfer)
3=transfer the call when AJVoIP is disconnected from the second party (attended transfer)
4=transfer the call when any party is disconnected except when the original caller was initiated the disconnect (attended transfer)
5=transfer the call when AJVoIP is disconnected from the second party. Put the caller on hold during the call transfer (standard attended transfer)
6=transfer the call immediately with hold and watch for notifications (unattended transfer)
7=transfer with no hold and no disconnect (simplest unattended transfer)
8=transfer with conference (will put the parties to conference on transfer; will mute or hold the old party by default)

Default is -1 (which is the same as 6)

Note:

- *Unattended means simple immediate transfer (just a REFER message sent)*
- *Attended transfer means that there will be a consultation call first*
- *The most popular transfertypes are 1, 5 and 6*
- *If you have any incompatibility issue, then set to 7 (unattended is the simplest way to transfer a call and all sip server and device should support it correctly)*
- *More details can be found here.*

transfwithreplace

(int)
Specify if replace should be used with transfer so the old call (dialog) is not disconnected but just replaced.
This way the A party is never disconnected, just the called party is changed. The A party must be able to handle the replace header for this.
-1=auto
0=no
1=yes
Default is -1

allowreplace

(int)
Allow incoming replace requests.
0=no
1=yes and always disconnect old ep
2=yes and don't disconnect if in transfer
3=yes but never disconnect old ep
Default is 2.

discontransfer

(int)
Specify if line should disconnect after transfer if not determined by the other transfer related configurations.
-1=auto
0=never
1= on C party connected status
2= on timeout
3= on connected or timeout
4= on ok for refer
Default is -1

disconincomingrefer

(int)
Specify if line should disconnect after incoming transfer
-1=auto
0=no
1= yes

Default is -1

transferdelay

(int)
Milliseconds to wait before sending REFER/INVITE while in transfer.
Default value is 400.

checksubscriptionstate

(int)
Specify if to handle transfer subscription state (such as reload the call on transfer failure)
-1=auto
0=no
1= disconnect
2= reload

Default is -1

subscribefortransfer

(int)
Create subscribe dialog for call transfer.
-1=auto
0=never
1= if no notify received
2= always

Default is -1

enabledirectcalls

(int)
Specify whether to enable direct call to SIP URI (peer to peer or via server)
0=no (so you should use only the username part of the SIP URI to make calls. set the domain part as the sipserveraddress parameter)
1=check IP in URI (will recognize full SIP URI if the domain part can be resolved to a valid IP)
2=always check (so you can make calls to full SIP URI without a SIP server to be set)
3=crossdomain (so you can register to domain A and make direct call to domain B)

Default is 1

charset

(string)

Set the character decoding for SIP signaling.
Default is empty (will load the local system default).
More details:
http://docs.oracle.com/javase/7/docs/api/java/nio/charset/Charset.html
http://www.iana.org/assignments/character-sets/character-sets.xhtml

checksrvrecords

(int)
SRV DNS record lookup setting:
-1: auto (will check SRV record for the VoIP server, but remembers if fails and will not check again next time)
0: don't lookup (will use only A record)
1: lookup A record first. If fails then lookup srv record (because mostly the srv record is not set anyway)
2: lookup SRV record first for VoIP server address. If fails then lookup A record (RFC compliant)
3: always lookup SRV record first. If fails then lookup A record
4: check also without the _sip._udp. prefix
If the SRV lookup returns multiple records, than SIP UA will failover to the next server on connection failures.
Default value is -1.

dnslookup

(int)
Domain record lookup mode
0=auto (same as 2)
1=yes always re-query
2=use cache if needed (default)
3=use cache whenever possible
4=from cache only
5=disable
Default value is 0.

forcewifi

(int)
Force WiFi network.
Possible values:
0=no
1=auto
2=lock in call
3=lock always
4=always
5=always lock
6=extra
Default value: 1

keepdeviceawakeincall

(int)
Keep device awake during calls.
Possible values:
0=no
1=basic only
2=auto
3=full then partial
4=always full only
Default value: 2

cpupartiallock

(boolean)
Keep a partial lock on the CPU while in call to prevent sleep.
Default value is true.

cpualwayspartiallock

(boolean)
Always keep a partial lock on the CPU to prevent sleep.
Default value is false.

unlockphone

(boolean)
Auto handle keyguard lock.
Default value is true.

proximitysensor

(int)
Enable/disable proximity sensor during calls.
Possible values:
0=disable
1=software based (deprecated)
2=hardware based
Default value: 2

checknativecall

(int)
Specify how to handle if there is a native/mobile call in progress.
Possible values:
0=don't check (will behave as it would be without native call in progress)
1=basic behavior changes (such as no ring)
2=don't alter the audio
Default value: 1
Should be set to 1 if using connection service or keep 1 always.

autousebluetooth

(int)
Specify when to use Bluetooth device.
Possible values:
0=guess
1=no
2=yes
Default value: 0

Set this parameter to 2 before the call if you have Bluetooth connectivity issues.

video

(int)
Enable/disable video.
Possible values:
-1: auto (default)
0: disable
1: enable
2: force always

RTC capabilities must be available for video to work.

aspeakermode

(int)
Speaker device to be used.
Possible values:
0=default (usually speakerphone/headphone)
1=headphone (speakerphone/headphone)
2=speakerphone (loudspeaker)
Default value: 0

Note: you can also switch at runtime using the SetSpeakerMode() API function.

audiomanagermode

(int)
Overwrite audio mode which is set by AJVoIP by calling the AudioManager.setMode API.
Should be used only if the defaults doesn't work correctly on your device / for your use-case.
By default AJVoIP will set the audio mode depending on the circumstances (call type, OS version, etc).
You might use this parameter to force an audio mode after your needs. It can be also changed at runtime and it will be applicable for the next setMode function calls.
Possible values:
-9: automatically managed by AJVoIP (recommended)
-8: never change
0: MODE_NORMAL
1: MODE_RINGTONE
2: MODE_IN_CALL
3: MODE_IN_COMMUNICATION
4: MODE_CALL_SCREENING
5: MODE_CALL_REDIRECT
6: MODE_COMMUNICATION_REDIRECT

Default value: -9

focusaudio

(int)
Specify whether to use audio "focus". (Auto decrease the volume of other apps).
Possible values:
0: auto
1: no
2: yes
Default value: 0

speakerphoneoutput

(int)
Specify how to set the speakerphone output device on SetSpeakerMode or when the aspeakermode parameter is set to 2.
Should be used only if the defaults doesn't work correctly on your device / for your use-case.
Possible values:
0: Auto guess (apply any required workarounds)
1: Speaker (just call setSpeakerphoneOn)
2: Speaker Forced (also set AudioManager.MODE_NORMAL)
3: VoIP (also set AudioManager.MODE_NORMAL above api level 11)
4: Bluetooth (try to wire audio to Bluetooth)
5: No changes (usually same as 5)
6: Ignore (don't call setSpeakerphoneOn)
Default value: 0

(Developer note: Internal usage: cfg_audiomode, localaudiomode. Affects only playback.)

audiorecorder

(int)
Audio recorder stream.
Should be used only if the defaults doesn't work correctly on your device / for your use-case.
Possible values:
0: Default (MediaRecorder.AudioSource.DEFAULT)
1: Auto guess (usually MediaRecorder.AudioSource.VOICE_COMMUNICATION, but depends also on the device)
2: Mic (MediaRecorder.AudioSource.MIC)
3: Voice call (MediaRecorder.AudioSource.VOICE_CALL)
4: Voice downlink (MediaRecorder.AudioSource.VOICE_DOWNLINK)
5: Voice uplink (MediaRecorder.AudioSource.VOICE_UPLINK)
6: Voice communication (MediaRecorder.AudioSource.VOICE_COMMUNICATION)
Default value: 1
(Normally it should be set to 0,1 or 2. Other values are very unusual)

audioplayer

(int)

Audio player stream for calls.
Should be used only if the defaults doesn't work correctly on your device / for your use-case.
Possible values:
0: STREAM_VOICE_CALL
1: STREAM_SYSTEM
2: STREAM_RING
3: STREAM_MUSIC
4: STREAM_ALARM
5: STREAM_NOTIFICATION
Default value: 0

Note: if the audio is set to loudspeaker then the audio stream is loaded from the speakerphoneplayer parameter instead.

speakerphoneplayer

(int)
Audio speakerphone player stream. Used when audio is set to loudspeaker.
Should be used only if the defaults doesn't work correctly on your device / for your use-case.
Possible values:
0: STREAM_VOICE_CALL
1: STREAM_SYSTEM
2: STREAM_RING
3: STREAM_MUSIC
4: STREAM_ALARM
5: STREAM_NOTIFICATION
6: default (same as audioplayer)
Default value: 3

(Developer note: Internal usage: cfg_speakerphoneplayer, getAltStream. Affects only playback.)

incomingcallalert

(int)
Audio player stream for ring.
Should be used only if the defaults doesn't work correctly on your device / for your use-case.
Possible values:
0: STREAM_VOICE_CALL
1: STREAM_SYSTEM
2: STREAM_RING
3: STREAM_MUSIC
4: STREAM_ALARM
5: STREAM_NOTIFICATION
Default value: 2

audioplayerusage

(int)
With this, you might overwrite the setUsage parameter for the AudioAttributes.
Default is USAGE_VOICE_COMMUNICATION.
Should be used only if the defaults doesn't work correctly on your device / for your use-case.

speakermode

(int)
Specify when to switch to speaker mode.
Should be used only if the defaults doesn't work correctly on your device / for your use-case.
Possible values:
0: auto
1: never
2: always
Default value: 0

useaudiodevicerecord

(boolean)
Set to false if you wish to disable audio recording from the local audio device.
You might set it to false if you wish to only receive the audio from the peer (not send) or if you will stream from an external source or from your app.
Default is true.

useaudiodeviceplayback

(boolean)
Set to false if you wish to disable audio playback on the local audio device.
You might set it to false if you wish to only send the audio from the peer (not receive) or if you will process the incoming audio stream yourself in your app.
Default is true.

enableaudiostreams

(int)
You can disable audio playback and/or recording streams with this option (will still open the device, but will not record/playback)
0=disable all
1=disable recording
2=disable playback
3=enable all
Default value is 3.

useaudiodevicerecord

(boolean)
Set to false to disable using the audio input device (microphone). In this case you should also set the mediatimeout parameter to 0.
Default is true.

useaudiodeviceplayback

(boolean)
Set to false to disable using the audio output device (speaker/earphone). In this case you should also set the mediatimeout parameter to 0.
Default is true.

vibrate

(int)
Vibrate on incoming call
0=auto (depends on phone settings)
1=no
2=yes
Default is 0.

playring

(int)
Generate ringtone for incoming and outgoing calls.
0=no (you can generate ringtone also by using the Java API to playback a sound file when you receive ringing notifications)
1=play ringtone for incoming calls
2=play ringtone for incoming and outgoing calls. (ringtone for outgoing calls can be generated also by your VoIP sever. When remote ringtone is received, java softphone will stop the local ringtone playback immediately and starts to play the received ringtone or announcement)
Default is 2.

ringtone

(string)
Specify a ringtone sound file or URI (system ring tones) to be used.

If you use a file instead of a system ringtone URI:

- *Set only the file name, not the full path. Copy the file to your app folder or near the AJVoIP library*
- *The file should be in the following format: PCM SIGNED 8000.0 Hz (8 kHz) 16 bit mono (2 bytes/frame) in little-endian (128 kbits).*
- *You can use any sound editor to convert your file to this format (usually from File menu -> Save as).*

Default value is empty. If not specified, then AJVoIP will use the default ringtone for call alert.

Example:

This example code illustrates how to present the UI for the user for choosing a native ringtone from android device ad how to get the path of the ringtone audio file:

Open ringtone UI:

```
public final int RINGTONE_REQCODE = 123456; // request code used in onActivityResult
String title = "Select your ringtone";

Intent intent = new Intent(RingtoneManager.ACTION_RINGTONE_PICKER);
intent.putExtra(RingtoneManager.EXTRA_RINGTONE_TITLE, title);
intent.putExtra(RingtoneManager.EXTRA_RINGTONE_SHOW_SILENT, false);
intent.putExtra(RingtoneManager.EXTRA_RINGTONE_SHOW_DEFAULT, true);
intent.putExtra(RingtoneManager.EXTRA_RINGTONE_TYPE, RingtoneManager.TYPE_ALL);
startActivityForResult(intent, RINGTONE_REQCODE);
```

Get the path to the selected ringtone in `onActivityResult`:

```
protected void onActivityResult(int requestCode, int resultCode, Intent data)
{
   if (requestCode == RINGTONE_REQCODE)
   {
      if(resultCode == RESULT_OK)
      {
         String ringTonePath = "";
         Uri uri = data.getParcelableExtra(RingtoneManager.EXTRA_RINGTONE_PICKED_URI);
         if (uri != null)
         {
             // the absolute path to the ringtone audio file selected by the user
             ringTonePath = uri.toString();
             Log.v("AJVOIP", "User selected ringtone path: " + ringTonePath);
            // pass it to AJVoIP
             mysipclient.SetParameter("ringtone", ringTonePath);
          }
      }
      if (resultCode == RESULT_CANCELED)
      {
         Log.v("AJVOIP", "user canceled ringtone selection");
      }
    }
  }
```

ringincall

(int)
Ring while in call if incoming or outgoing call
0=No
1=Only a beep for incoming call
2=Yes, normal ring
Default value is 1

checkvolumesettings

(int)
Check if audio device is muted or volume settings are too low (and un-mute and/or increase volume if necessary).
0: no
1: at first run
2: always
Default value is 1

volumein

(int)

Default microphone volume in percent from 0 to 100. 0 means muted. 100 means maximum volume, 0 is muted.
Default is 50% (not changed)

Note:
The result volume level might be affected by the AGC if it is enabled.
Volume levels above 70% might result in distorted sound.
Usually it should not be changed as users should control the volume by OS level volume controls.

volumeout

(int)
Default speaker volume in percent from 0 to 100. 0 means muted. 100 means maximum volume, 0 is muted.
Default is 50% (not changed)

Note:
The result volume level might be affected by the AGC if it is enabled.
Volume levels above 70% might result in distorted sound.
Usually it should not be changed as users should control the volume by OS level volume controls.

volumering

(int)
Default ringback volume in percent from 0 to 100. 0 means muted. 100 means maximum volume, 0 is muted.
Default is 50% (not changed)

Note: Usually it should not be changed as users should control the ringer by OS level volume controls.

beeponconnect

(int)
Will play a short sound when calls are connected
0=Disabled
1=For auto accepted incoming calls
2=For incoming calls
3=For outgoing calls
4=For all calls
Default value is 0

playdtmfsound

(int)
DTMF sounds on key press.
0=No
1=Yes for one digit
2=Yes also if multiple digits
Default is 1

agc

(int)
Automatic gain control.
0=Disabled
1=For recording only
2=Both for playback and recording
3=Guess
Default value is 3

This will also change the effect of the volumein and volumeout settings.

plc

(boolean)

Enable/disable packet loss concealment
Default is true (enabled)

vad

(int)
Enable/disable voice activity detection.
0: auto
1: no
2: yes for player (will help the jitter)
3: yes for recorder
4: yes for both
Default is 2.

Notes:
-The vad parameter is automatically set to 4 by default if the aec2 algorithm is used.
-If you wish to use VAD related statistics in your application, you might have to also set the “vadstat” parameter after your needs. Possible values: 0=no,1=auto (default),2=detect no mic audio,3=send statistics. See the VAD notification and VAD for more details.
-If you want to disable audio related notifications (microphone warning on no signal detected) then set the “enablenomicvoicewarning” parameter to 0

rtpstat

(int)
Enable/disable RTP statistics by triggering RTPSTAT notifications.
Possible values:
-1: auto (will trigger RTPSTAT events in every 6-7 seconds and more frequently at the beginning of the calls)
0: disabled
Positive value: seconds to generate RTPSTAT events.

Default is 0 (disabled)
More details here.

aec

(int)
Enable/disable acoustic echo cancellation
0=no
1=yes except if headset usage guessed
2=yes if supported
3=forced yes even if not supported (might result in unexpected errors)
Default is 1.

Note:
The aec decision might be overwritten or modified by the aectype parameter.

After tests, the success rate of the built-on AEC algorithms is above 90%. This is considered a good value as there are no any perfect aec algorithms, but it might not work (or partially work) in some devices or circumstances (depends on many factors such as audio hardware, driver, OS/device, environment/room, codec, network delay).

For more details, read here.

aectype

(string)
AEC algorithm(s) to use.
One or more of the following strings separated by comma:

- auto: will select automatically based on circumstances (CPU power, device capabilities, network)
- none: disable aec
- software: software aec (requires extra CPU processing)
- hardware: android hardware aec capabilities (not supported on all phones)
- fast: a fast software aec implementation (used preferably on slow/old devices or in addition for the above algorithms)
- volume: this will just decrease the volume when speech detected from other end (using VAD)

Default is auto.

Note:

- *It is recommended to leave both the aec and aectype values with its default values*
- *To force all echo cancellations algorithms you might set the aec to "2" and the aectype to "software,hardware,fast,volume" (this might be too much, forcing the software to do unnecessary processing and in some circumstances it might remove too much audio)*

For more details, read here.

denoise

(int)
Noise suppression.
0=Disabled
1=For recording only
2=Both for playback and recording
3=Auto guess
Default value is 3

silencesupress

(int)
Enable/disable silence suppression
Usually not recommended unless your bandwidth is really bad and expensive.
-1=auto
0=no (disabled)
1=yes
Default is -1 (which means no, except mobile devices with low bandwidth)

hardwaremedia

(int)
Enable the usage of OS/hardware level audio processing for aec/denoise/silence suppress.
0=auto
1=no
2=yes
Default is 0

Note: even if this is set to 2, the media stack might not use the android hardware capabilities if you disable aec/denoise/silencesupress by their respective parameters

rtcp

(boolean)
Enable/disable rtcp. (RFC 3550. Partial support)

codec

(string)
List of allowed audio codec's separated by comma.
Will accept one or more of the following strings (upper or lower case doesn't matter):
pcmu, pcma, g711 (for both PCMU and PCMA), g729, gsm, ilbc, speex, speexwb, speexuwb, opus, opuswb, opusuwb, opusswb, def

By default the Android SIP library will automatically choose the best codec depending on available codec's, circumstances (network/device) and peer capabilities.
Set this parameter only if you have some special requirements such as forcing a specific codec, regardless of the circumstances.
Default: empty (which means auto detection and negotiation)
Recommended value: leave it empty for "best" codec negotiation unless if you have some specific requirement.

Example: OpusWB,G.729,PCMU (This will disable Speex, GSM, iLBC, GSM and PCMA).
You can also set one single codec (if your server/peers allows only one codec or for testing to rule out all kind of codec negotiation related issues).
You can set to "def" to enable all the default codec's again.

Under normal circumstances, the following is the built-in codec priority:

I. *Wideband Speex and Opus (These are set with top priority as they have the best quality. Likely used for VoIP to VoIP calls if the peer also has support for wideband)*
II. *G.729 (Usually the preferred codec for VoIP trunks used for mobile/landline calls because it's excellent compression/quality ratio for narrowband)*
III. *iLBC, GSM (If G.729 is not supported then these are good alternatives. iLBC has better characteristics and GSM is better supported by legacy hardware)*

IV. G.711: PCMU and PCMA (Requires more bandwidth, but has the best narrowband quality)

The default codec order will highly depends on the device CPU power (high-complexity codec's are deprioritized on low-end devices) and network type (high bandwidth codec's are deprioritized on slow networks such as GPRS, 2G or slow 3G). On a decent device with good internet connection (WiFi, LTE, 5G, 4G or good 3G) the softphone should pick up the Opus wideband if peer has support for this codec. If peer has support only for narrowband then G.729 should be selected, otherwise another codec negotiated with the peer endpoint.

More details here.

Note: for the codec configuration, either use the codec/prefcodec settings OR the use_xxx settings (where xxx is the codec name; listed below).
You can achieve similar result with both. The use_xxx settings are a little bit more flexible regarding the codec prioritization.

prefcodec

(int)
Set your preferred audio codec.
Will accept one the following strings (upper or lower case doesn't matter):
pcmu, pcma, g711 (for both PCMU and PCMA), g729, gsm, ilbc, speex, speexwb, speexuwb, opusnb, opuswb, opusuwb, opusswb
Default is empty which means the built-in optimal prioritization.
By default the SIP stack will present the codec list optimized regarding the circumstances (the combination of the followings):

- available client codec set (not all engines supports all codecs)
- server codec list (depending on your server, peer device or carrier)
- internal/external call: for IP to IP calls will prioritize wideband codecs if possible, while for outbound calls usually G.729 will be selected if available
- network quality (bandwidth, delay, packet-loss, jitter): for example iLBC is more tolerant to network problems if supported
- device CPU: some old mobile devices might not be able to handle high-complexity codec's such as opus or G.729. G711 and GSM has low computational costs

You can also fine-tune the codec settings with the use_xxx settings described below.

vcodec

(string)
List of allowed video codec's separated by comma (H264, VP8, VP9).
You might use this parameter to exclude some codec from the offer list.
For example if you wish to use only H.364, then set this to: "H264"
Default: empty (which means auto detection and negotiation)

video_bandwidth

(int)
Max bandwidth for video in kbits.
It will be sent also with SDP "b:AS" attribute.
Default is 0 which means auto negotiated via RTCP and congestion control.

video size parameters

(int)
You can suggest the size of the video (in pixels) with the following parameters:

- video_width
- video_height
- video_min_width
- video_min_height
- video_max_width
- video_max_height

use_gsm

(int)
GSM codec setting. 0=never,1=don't offer,2=yes with low priority,3=yes with high priority
Default is 1.

use_ilbc

(int)
iLBC codec setting. 0=never,1=don't offer,2=yes with low priority,3=yes with high priority
Default is 1.

use_speex

(int)
Narrowband speex codec setting. 0=never,1=don't offer,2=yes with low priority,3=yes with high priority
Default is 1

use_speexwb

(int)
Wideband speex codec setting. 0=never,1=don't offer,2=yes with low priority,3=yes with high priority
Default is 2

use_speexuwb

(int)
Ultra wideband speex codec setting. 0=never,1=don't offer,2=yes with low priority,3=yes with high priority
Default is 1

use_opusnb

(int)
Narrowband (8000 Hz) opus codec setting. 0=never,1=don't offer,2=yes with low priority,3=yes with high priority
Default is 1

use_opuswb

(int)
Wideband (16000 Hz) opus codec setting. 0=never,1=don't offer,2=yes with low priority,3=yes with high priority
Default is 2

use_opusswb

(int)
Super wideband (24000 Hz) opus codec setting. 0=never,1=don't offer,2=yes with low priority,3=yes with high priority
Default is 1

Note: this codec might not work on all devices, depending on the audio device and driver capabilities!

use_opusuwb

(int)
Ultra wideband (fullband at 48000 Hz) opus codec setting. 0=never,1=don't offer,2=yes with low priority,3=yes with high priority
Default is 1

use_g729

(int)
G.729 codec setting. 0=never,1=don't offer,2=yes with low priority,3=yes with high priority
Default is 2
**In some countries a license/patent is required if you use G.729 so enable only if you have licenses or licenses are not required in your case (consult your lawyer if you are not sure)*

use_pcma

(int)
G711alaw codec. 0=never,1=don't offer,2=yes with low priority,3=yes with high priority
Default is 2

use_pcmu

(int)

G711ulaw codec. 0=never, 1=don't offer, 2=yes with low priority, 3=yes with high priority
Default is 1

disablewbforpstn

(int)
This setting will disable speex and opus wideband and ultrawideband for outgoing calls to regular phone numbers since these are usually not supported for pstn calls and they might requires longer initialization.
0: no
1: check at first call
2: check all calls
Default is 0.

setfinalcodec

(int)
Some server cannot handle the final codec offer in the ACK message correctly.
In this case you will have to set this setting to 0, otherwise you will have one way audio.
0=never (RFC compliant)
1=auto guess (not send in case of certain servers and autocorrect in subsequent calls)
2=when multiple codecs are received
3=always reply with the final codec in the ACK message
Default value is 1.

ims3gpp

(int)
Enable 3GPP (IMS/5G/VoLTE) features.
Possible values:
-1: auto detect (default)
0: disable
1: basic
2: full/all

It is also possible to enable only certain 3GPP features. In this case use the following parameters instead of ims3gpp:

- ims3gpp_phonenum: use tel URI and set user=phone (-1: auto, 0: no, 1: user=phone, 2: also tel uri). Note: you might also use the sipproto param.
- ims3gpp_minphonelen: tel uri only if phone number is longer then this. Default is 4. Set to 0 to use tel also for non-phone number peers
- ims3gpp_authuname: auth username must contain the domain (-1: auto, 0: no. 1: yes)
- ims3gpp_ussd: IMS USSD described at 3GPP TS 24.390. Send with SendUSSD, receive as USSD notifications (-1: auto, 0: no, 1: yes)
- ims3gpp_sms: +g.3gpp.smsip with MESSAGE using binary encoded PDU's described in 3GPP TS 24.011 (-1: auto, 0: no, 1: yes)
- ims3gpp_smsc: SMSC Center Address number. Otherwise AJVoIP will use the destination number (if that is a phone number)

enable_3pcc

(int)
Specify if to enable 3PCC Third Party Call Control as described at RFC 3725.
Note: 3PCC has nothing to do with 3GPP. They are completely different technologies.
This feature is often used in callcenters or for click-to-call to setup calls between two or more other parties.
Possible values:
0: disable (block 3PCC requests)
1: don't handle
2: accept (handle incoming requests)
3: initiate 3PCC calls by sending with no SDP
4: initiate 3PCC calls by sending SDP with connect IP set 0.0.0.0
5: initiate 3PCC calls by sending SDP with no media line
Default is 2.

If 3pcc is enabled, then AJVoIP will send the codec answer in ACK if not received by INVITE or it will re-INVITE if received SDP with no media line or with 0.0.0.0 contact IP, thus allowing all the possible third party call controls.
Some more related information can be found here, here, here and here.
You might also set the autoaccept parameter to true if AJVoIP acts as an auto-responder.

ed137

(int)
Enable/disable the ED-137 behavior to be used for air traffic management services.
Set to 0 to ignore the ED-137 specification (act as normally)
Set to 1 to turn on the ED-137. This will change a list of internal settings and behaviors to conform with the ED-137 specification.
Default value is 0.
See the ED-137 guide for more details.

alwaysallowlowcodec

(int)
Set to 2 to always put low computational and low bandwidth codec in the offer list, specifically GSM and PCMU. Low CPU or bandwidth devices might choose these codecs (such as a mobile phone on 3G).
Set to 0 to disable.
Default is 1 (auto)

codecframecount

(int)
Number of payloads in one UDP packet.
By default it is set to 0 which means 2 frames for G729 and 1 frame for all other codec.

udptos

(int)

Sets traffic class or type-of-service octet in the IP header for packets sent from UDP socket which can be used to fine-tune the QoS in your network. As the underlying network implementation may ignore this value applications should consider it a hint.

The value must be between 0 and 255.
Valid values:

- 0: disabled
- 1: automatic (set to 10 under normal conditions and disabled when in tunneling)
- 2: low-cost routing
- 4: reliable routing
- 8: throughput optimized routing
- 10: low-delay routing
- or'ing the above values (from above 2)

Default value is 1.

Notes:

for Internet Protocol v4 the value consists of an number with precedence and TOS fields as detailed in RFC 1349. The TOS field is bitset created by bitwise-or'ing values such the following :

IPTOS_LOWCOST: 2
IPTOS_RELIABILITY: 4
IPTOS_THROUGHPUT: 8
IPTOS_LOWDELAY: 16

The last low order bit is always ignored as this corresponds to the MBZ (must be zero) bit.
For Internet Protocol v6 tc is the value that would be placed into the sin6_flowinfo field of the IP header.

This parameter might work only in preset environments; the runtime might not have enough rights to modify the IP headers.

automute

(int)
Specify if other lines will be muted on new call
0=no (default)
1=on incoming call
2=on outgoing call
3=on incoming and outgoing calls
4=on other line button click
Default is 0

autohold

(int)

Specify if other lines will be put on hold on new call
0=no (default)
1=on incoming call
2=on outgoing call
3=on incoming and outgoing calls
4=on other line button click
Default is 0

holdontransfer

(int)
Specify if line should disconnect after transfer
-1=auto
0=no
1= yes
2= hold and reload if needed
Default is -1

holdtypeonhold

(int)
Specify call hold type.
Call hold is usually initiated by the Hold function and with parameter you can specify which type of call hold do you wish to request.
Possible values:
-2: no
-1: auto (defaults to 2)
0: no (don't hold. a=sendrecv)
1: reserved (not used)
2: hold: instructs the peer to not send audio (mute speaker. Local AJVoIP: a=sendonly, peer: a=recvonly)
3: hold: not sending audio to the other (mute microphone. Local AJVoIP: a=recvonly, peer: a=sendonly)
4: hold: both (mute both in/out audio. a=inactive)

Default is -1.

Note:
By default the hold will check the previous state. For example if previously it was local hold (2) and you switch to remote hold (3) then actually will switch o both hold.
There is also a holdexplicite parameter which if set to 1, then the hold will be done strictly after the holdtypeonhold parameter, without considering previous state. This way you can also change between local and remote hold without the need to unhold first.

muteonhold

(int)
Specify if call also have to be muted with hold (stop recording/playback and stop the according RTP stream).
Possible values:
-2: no mute,
0: mute in and out
1: mute out (speakers)
2: mute in (microphone)
3: mute in and out (same as 0)
4: mute default
5: according to call hold.

Default is 5.

Usually you should set this either to -2 or 5.

musiconhold

(int)
Specify if to play music on hold (MOH).
Possible values:
-1: auto (defaults to 0/disabled)
0: no/disabled
1: yes on hold with a=sendonly request initiated locally
2: yes on hold with a=recvonly received from the peer
Default is -1.

Note:

- *Option 1 (hold with* a=sendonly *initiated locally) is the recommended value if you wish to enable music on hold*
- *For MOH to work, you must have a "music.wav" resource file. You might use this file.*
- *MOH might be played only if the holdtype is -1 (auto) or 2 (sendonly).*
- *MOH might be handled on the SIP server side instead.*

defmute

(int)
Default mute direction
0: both
1: mute out (speakers)
2: mute in (microphone)
3: both
4: both
5: disable mute

ackforauthrequest

(int)
If to send ACK for authentication requests (401,407).
0=no
1=yes (default)
Should be changed only if you have compatibility issues with the server used.

favorizecontactaddr

(int)
You may change it if you have compatibility issues with stateless proxies
0=never
1=no
2= conform RFC
3= yes. Sending for both server and contact URI (default)
4=always

prack

(boolean)
Enable 100rel (PRACK)
Set to false if you have incompatibility issues.
Default is false.

sendmac

(boolean)
Will send the client MAC address with all signaling message in the X-MAC header parameter.
Default value is false.

Note: The SDK might not have enough rights to read the MAC address.

useragent

(string)
This will overwrite the default User-Agent setting.
Do not set this when used with mizu VoIP servers because the server detects extra capabilities by reading this header.
Default is empty (which usually defaults to "AJVoIP/versionnumber")

sendsessionid

(int)
Specify if to send the Session-ID header as specified in RFC 7989.

Possible values:

-1: if received (default)

0: no (disable)

1: yes (enable)

Default is -1.

customsipheader

(string)
Set a custom sip header (a line in the SIP signaling) that will be sent with all messages. Can be used for various integration purposes (for example for sending the http session id). You can also change this parameter runtime with the SetSIPHeader function.
Default value is empty.

customsdpmediafield

(string)
Set a custom media SDP field (a line in the SDP body after the m= line) that will be sent with all messages.
It can be set also at runtime with the SetSDPField function.
Default value is empty. Multiple lines can be separated by semicolon (;).

rtpextraheader

(string)
Set RTP extra header bytes.
The string will be converted to RTP extra header word(s).
If only one extra word needs to be set, then pass it as an integer (integer value converted to string).
Multiple words can be set by separating them with semicolon (;) .Example: 98;76543 will set the first word to 98 and the second word to 76543.
It can be set also at runtime with the RTPHeaderExtension function “extension” parameter.
See the RTP header extension FAQ point for more details.

rtpextraheader_profile

(int)
Set the profile number (the first two bytes) for the RTP extra header if required.
If not set then the profile bytes will be set to 0.
It can be set also at runtime with the RTPHeaderExtension function “profile” parameter.
See the RTP header extension FAQ point for more details.

sip_uui

(string)
Specify UUI data (User-to-User Call Control Information as described in RFC 7433).
Can be used to send data to other endpoints. Not all endpoints and SIP servers supports this feature.
*If prefixed with *0: then the UUI will be sent only to peer with the User-to-User header.*
*If prefixed with *1: then the UUI will be sent only to target party with call transfer or redirect escaped in Contact or Refer-To URI.*
*If not prefixed or prefixed with *2: the UUI will be sent both way.*

Example:

It can be set also at runtime with the SetUUI function.
Default value is empty.

Simple example: mydata
Example with parameters: *2:mydata;encoding=hex;purpose=foo;content=bar

techprefix

(string)
Add any prefix for the called numbers.
Default is empty.

normalizenumber

(int)
Normalize (called) numbers by removing .-,;()[]: and space if the string otherwise doesn't contains a-z or A-Z characters (looks like a phone number).

Possible values:
-1: auto (usually defaults to 1 yes, except if self username also contains special characters)
0: no
1: yes
Default is -1.

honordatasaver

(int)
Take in consideration Android OS Data Saver settings.
0: no
1: yes
Default is 1.

If 1, then it will change a few internal settings to use less data such as longer kee-palive and reregister interval, prioritizing codec's which requires less bandwidth, auto enable silence suppression and more.

honordnd

(int)
Take in consideration Android DND settings (Do Not Disturb).
0: no
1: yes
Default is 1.

honorairplan

(int)
Take in consideration Android OS Airplane mode settings.
0: no
1: yes
Default is 1.

If 1, then it will not force network (and wifi) reconnect if no network.

mustconnect

(boolean)
If set to true, than users must register before to make any calls.
Default value is false.

rejectonphonebusy

(int)
Set to true to reject the incoming calls if there is already a native phone call in progress (GSM/mobile call).
Possible values:
0: Reject calls
1: Allow calls
2: Forward calls
Default value is 1.

For better accuracy, this feature requires "dangerous" READ_PHONE_STATE permission.

rejectonbusy

(boolean)
Set to true to reject the incoming calls if there is already a VoIP call in progress.
This feature might not be enabled by default as it requires dangerous READ_PHONE_STATE permission.
Default value is false.

callforwardonbusy

(String)

Specify a number where calls should be forwarded when the user is already in a call. (Otherwise the new call alert will be displayed for the user or a message will be sent on the API)
Default is empty.

callforwardalways

(String)
Specify a number where ALL calls should be forwarded.
Default is empty.

calltransferalways

(String)
Specify a number where ALL calls should be transferred.
This might be used if your softswitch doesn't support call forward (302 answers).
Default is empty.

autoignore

(int)
Set to ignore all incoming calls.
0=don't ignore
1=silently ignore
2=reject
Default value is 0.

autoaccept

(boolean)
Set to true to automatically accept all incoming calls (auto answer).
Default value is false.

Note: Autoanswer can be also forced from the server by the "P-Auto-Answer: normal" SIP header.

blacklist

(string)
Block incoming communication from these users. (users/numbers separated by comma).
Default value is empty.

whitelist

(string)
Allow incoming SIP requests only from these users (users/numbers separated by comma).
Be aware that your server might send requests on it's own, such as OPTION requests. In this case the username used by the server should be also allowed, otherwise such requests will be ignored.
Default value is empty.

blockmode

(int)
Specify how to apply the blacklist and the whitelist.
Possible values:
0: disable (black/white list will be applied only for presence and BLF)
1: ignore messages at transport level
2: reject call sessions
3: both
Default: 1

rejectcallto

(int)
Set to ignore calls if target doesn't match
0=accept all incoming calls

1=check if target user match
2=check rinstance
3=check rinstance strict
4=check all strict
Default value is 0.

hideautocall

(int)
Set to 1 to suppress notifications (STATUS, CDR) from automatically handled calls (ignored, forwarded, rejected and similar).
0=send status notifications also about auto handled calls
1=do not send status notifications from auto handled calls
Default is 0.

ringtimeout

(int)
Maximum ring time allowed in millisecond.
Default is 90000 (90 second)

calltimeout

(int)
Maximum speech time allowed in millisecond.
Default is 10800000 (3 hours)

startsipstack

(int)
Automatically start the sipstack after a specified time.
0=no (the sipstack will be started on the first register or call event)
1=on startup if not tunneling or serveraddress/username/password are set (the sipstack will be started at app init)
2=on startup always (the sipstack will be started at app init)
Other=seconds (the sipstack will be started after the specified seconds)
Default value is 1

You can set to 0 if there is less change that AJVoIP will be used once the users will open the webpage hosting AJVoIP.
You can set to 1 or higher if there is a high probability that the user will use AJVoIP to make calls (this will shorten the setup time for the first call)

timer

(int)
You can slow down or speed up the SIP protocol timers with this setting. You may set it to 15 if you have a slow server or slow network.
Default value is 10.

timer2

(int)
Same as “timer” but it affects idle, connect and ring timeout and maximum call durations.
Default value is 10.

mediatimeout

(int)
RTP timeout in seconds to protect again dead sessions.
Calls will be disconnected if no media packet is sent and received for this interval.
You might increase the value if you expect long call hold or one way audio periods.
Set to 0 to disable call cut off on no media.
Default value is 300 (5 minute)

mediatimeout_notify

(int)
RTP timeout in seconds for API notify.
After this timeout a warning message is sent via notifications without any further action.

The following log will be generated: “WARNING,media timeout (notify)”
Default value is 0 (disabled)

rtpkeepaliveival

(int)
RTP stream keep-alive packet send interval in milliseconds.
This is useful if your PBX has an RTP timeout setting to prevent disconnects when the java softphone is hold or muted.
Default value is 0. (You might set it to 25000 for example)

sendrtponmuted

(boolean)
Send rtp even if muted (zeroed packets)
Set to true only if your SIP server is malfunctioning when no RTP is received.
Default value is false.

discmode

(int)
For call disconnect compatibility improvements. Some VoIP devices might have bugs with CANCEL forking, so it is better to always send a BYE after the CANCEL message on call disconnect. In this case set the discmode parameter to 3.
1: quick
2: conform the RFC
3: send BYE after CANCEL when needed
4: double: always repeat the CANCEL and the BYE messages
Default value is 2.

waitforunregister

(int)
Maximum time in milliseconds to wait for unregistration when the Unregister is called or the java sip stack is closed.
If set to 0 that an unregister message is sent (REGISTER with Expires set to 0) but AJVoIP is not waiting for the response, which means that it will not repeat the un-register in case if the UDP packet was lost.
Default value is 2000.

detectlanpeers

(int)
Auto detect other SIP endpoints on the same LAN (broadcast message) such as colleagues at the network place or family members behind the home wifi.
When other user is found, a NEWUSER event notification is triggered.
Possible values:
0: no (will disable also the listener so other endpoints will not be able to detect it)
1: listen only (will be discoverable)
2: yes find others
Default is 2.

textmessaging

(int)

Specify text messaging mode (IM/chat/SMS/API)

-1: auto guess or ask (default)

0: disable all

1: disable incoming messages and auto guess outgoing mode

2: disable message sending and auto guess incoming mode

3: reserved (you might implement HTTP API based chat for this externally)

4: reserved (you might implement native SMS for this externally)

5: VoIP SMS

6: VoIP IM/chat (SIP MESSAGE)

Set the ims3gpp_sms parameter to 1 or 2 to use 3GPP binary encoded messages for SIP MESSAGE or to 0 to always disable.

Note: the old haschat and chatsms parameters are deprecated now but still supported

offlinechat

Will try to resend not delivered messages later (on next register and/or when any message received from peer). The offline message queuing will take care of filtering out duplicates and will try to resend the message multiple times on failure until max number of resend or timeout reached.
(int)
-1=auto (usually yes, no for 3gpp sms)
0=no (disable offline chat)
1=yes (default)
2=force always

Default is -1.

encrypted

(boolean)
Specify if the transport will be encrypted (both media and the signaling)
Compatible only with Mizu VoIP servers.
Automatically turned on when using http tunneling.
For standard encryption you can use TLS/SRTP with the transport/mediaencryption parameters.
Default is false.

authtype

(int)
Some IP-PBX doesn't allow "web" or "proxy" authentication.
0=normal
1=only proxy auth
2=only simple auth

pwdencrypted

(int)
Specify if you will supply encrypted passwords via parameters or via the Java API
0=no (default)
1=xor
2=des+base64
3=xor+base64 (this is the preferred method; easiest but still secure enough)
4= base64
This method is deprecated from version 3.4. All parameters can be passed encrypted now by just prefixing them with the "encrypted__X__" string where X means the id of the encryption method used.

From version 4.8 there is no need to specify this parameter anymore. Just prefix any parameter with encrypted_X as described here.

voicerecording

(int)
0=no (default)
1=local (in the user home directory)
2=remote ftp or http upload
3=both

More details here.

voicerecfilename

(int)
The format of the recorded filenames.
0=date-time + peer name (default)
1=date-time + sip call-id
2=sip call-id
3=date-time + username
4=date-time + username + peer name

The date-time will be formatted in the following way: yyyyMMddhhmmss

Note: You can also use the "voicerecfilenameprefix" parameter to add a prefix for the file name.

voicerecftp_addr

(string)
FTP location for the recorded voice files if the “voicerecording” parameter is set to 2 or 3.
Format: ftp://USER:PASS@HOST:PORT/PATH/TO/THEFILE
Example: ftp://user01:pass1234@ftp.foo.com/FILENAME

The FILENAME part of the string will be replaced with the file name according to the “voicerecfilename” parameter.

voicerecformat

(int)
Recorded file compression.
0: PCM wave stereo files with separate channels for in/our (default)
1: raw gsm. 2 files will be generated for each call. One for the recorder file and another for the playback. These files can be played with players supporting gsm codecs for example quicktime (which works also as a browser plugin) or a winamp plugin is downloadable from here.
2: ogg/vorbis (optional, on request; module might not be included by default)

voicerecordingbuff

(int)
The maximum recorded file length.
-1: dynamic, no limit
1: max around 1 minute
2: max around 2 minute
…

Default is -1.

syncvoicerec

(int)
How to synchronize the recording/playback side:
-1: Auto
0: No (don’t synchronize)
1: Yes (fill with noise the other channel)
2: Yes (wait for both side)
Default: 2

uploadretry

(int)
Specify whether the file upload should be retried on failure if the voicerecording parameter is set to 2 or 3.
0: no
1: once
2: until success
Default: 1

ftp_addr

(string)
FTP location for general storage (for example for settings, contactlists)
Format: ftp://USER:PASS@HOST:PORT/PATH/TO/THEFILE
Example: ftp://user01:pass1234@ftp.foo.com/FILENAME

The FILENAME part of the string will be replaced with the actual file name.

http_addr

(string)
HTTP location for general storage (for example for settings, contactlists)
Format: http://www.yourdomain.com/storage/
(this is just an example URL format. This URL will not work. You need to change this to your own web address)

webrtcserveraddress

(int)
WebSocket server URL for WebRTC video if any. Example: wss://rtc.myserver.com/sip

autocfgsave

(int)
Configurations and statistics are stored in a local file to be reused in next sessions.
This is not critical and the Java VoIP client will work just fine if this file is lost or deleted by the user.
Sometime is useful to not allow configuration/settings storage on the user device.
The autocfgsave option can be set to the following values:

- -2: disable all file write forced
- -1: disable file write
- 0: disable config storage
- 1: save only
- 2: load only
- 3: save and load

Default is 3.

resetsettings

(boolean)
Set to true to clear all previously stored or cached settings.
Default is false.

Note: If set to true, then AJVoIP will not remember any previous settings, including its own internal optimizations and NAT related discoveries.

signalingport

(int)
Specify local SIP signaling port to use.
Default is 0 (a stable port which is selected randomly at the first usage)

Note: this is not the port of your server where the messages should be sent. This is the local port for sip user agent.

rtpport

(int)
Specify local RTP port base.
Default is 0 (which means signalingport + 2)
Note: If not specified, then the Android SIP SDK will choose signalingport + 2 which is then remembered at the first successful call and reused next time (stable rtp port). If there are multiple simultaneous calls then it will choose the next even number.

incrtpport

(int)
Increment RTP port for each call by this value.
Might be needed only with some misbehaving routers.
Default is 0.

Note: You will still have a different RTP port for each simultaneous calls even if this is set to 0.

bindip

(String)
Specify local network interface IP address for the sockets to bind to.
This parameter might be used only on devices with multiple local IP addresses to force the specified IP.
This parameter should be used only with local private IP (an IP address which is present on the device).
Default is empty (by default it doesn't bind to any IP and it is up to the OS routing table from which IP the packets are sent)

Note: This parameter should be used only in very specific circumstances when the device has multiple IP address and you wish to use only one of them.

localip

(String)
Specify local IP address to be used for the SIP signaling.
This parameter might be used only on devices with multiple ethernet interface to force the specified IP or if AJVoIP is behind NAT to specify its public IP.

This parameter should be used only with static IP (if the device IP doesn't change dynamically from DHCP) or if you can better detect the best IP to be used in your app instead of letting AJVoIP to auto detect it.
Default is empty (auto-detect best interface to be used or detect the external IP)

Note:
AJVoIP by default will auto detect the "best" IP to be used and this parameter should be used only in very specific circumstances.
It is also possible to set the address sent in SDP explicitly with the localsdpip and localsdpport parameters.

favlocalip

(String)
Specify local subnet preference.
For example if the device where AJVoIP is running might have two separate IP (such as 192.168.1.5 and 10.0.0.5) then you might set the favlocalip to 192 to always prefer the 192.x.x.x subnet.
This parameter might be used only on devices running on a known environment (local LAN) in case if you wish to suggest the subnet to be used.
Default is empty (auto-detect best subnet to be used)

Note: AJVoIP by default will auto detect the "best" IP to be used and this parameter should be used only in very specific circumstances.

increasepriority

(boolean)
This will increase the priority for the whole thread-group which might help on slow CPU's or when other applications are generating high CPU load.
Note: the priority of the threads handling media are increased regardless of this setting.
Default is false.

loglevel

(int)
Tracing level. Values from 0 to 9.

You should set the default loglevel to 1 for production and 5 for test or development.

Loglevel 0 is not recommended as the Android SIP SDK will not even display important even notifications for the user in this case.
Loglevel 5 means a full log including SIP signaling.
Loglevel higher than 5 should be avoided (this can slow down the application).

The logs are sent to the following outputs:

- status display (only level 1 –these are the most important events that should be displayed also for the enduser)
- file if the canlogtofile parameter is set to 1 or 2
- logcat if the logtoconsole parameter is set to true
- system log if the systemout parameter is set to true (System.Out)
- accessible for the GetLogs() API
- sent with notifications if the events parameter is 3

More details here.

logtoconsole

(int)
Whether to send tracing to the java console (System.out.print or often referred as STDOUT).
Possible values:

- 0: no (disable log to console)
- 1: auto (depending on loglevel)
- 2: always (always log to stdout)

Default is 1.

systemout

(boolean)
Whether to send logs to standard output.
Default is false.

logcat

(boolean)

Whether to send logs to standard Logcat.
Default is true.

canlogtofile

(int)
Specify if AJVoIP should write the logs to file.
Possible values:

- 0: never
- 1: if the loglevel is higher then 2 and not headless
- 2: always

Default value is 1.

With loglevel set to 2 or more, the logs are written also to a local file by default.
To disable the log files, set the canlogtofile parameter to "0" (or set the "loglevel" to 1 and keep the canlogtofile default at 1).

The logs are usually stored at \mwphonedata\webphonelog.dat or you can specify the path with the logpath parameter. The exact path can be queried with the GetLogPath API.
The log file is recreated with every startup and the old log file is renamed to previous_webphonelog.dat (so you will have max 2 log files: the current and the previous).

capabilityrequest

(boolean)
If set to true then will send a capability request (OPTIONS) message to the SIP server on startup. The serveraddress parameter must be set correctly for this to work. This method is useful to release the security restrictions when using the Android SIP library with the API and also to open the NAT devices.
Default value is false.

natkeepalive

(boolean)
If set to true then will send a short message (\r\n) to the SIP server on startup. The serveraddress parameter must be set correctly for this to work. This method is useful to release the security restrictions and also to open the NAT devices.
Deprecated since v.3.8.2
Default value is false.

keepaliveival

(int)
NAT keep-alive interval in milliseconds which is usually sent from register endpoints.
Default value is 28000. (28 seconds)

recaudiobuffers

(int)
Number of buffers used for audio recording.
Default is 7.

recaudiomode

(int)
Audio recording mode. 0 means default; 1 means event based; 2 means device poll.
Default is 0.

jittersize

(int)
Although the jitter size is calculated dynamically, you can modify its behavior with this setting.
0=no jitter,1=extra small,2=small,3=normal,4=big,5=extra big,6=max
Default is 3

allowspeedup

(int)
Specify whether to enable audio playback speedup on high queue size.
(Instead of dropping packets, it might attempt to speed up the playback rate for a short time until queue size drops below threshold)
Possible values: -1: Auto Guess (default), 0: No, 1: Yes

allowfirstdrop

(int)
Specify whether to enable drop of RTP packets on media start.
Sometime the other end might start sending RTP before the media is opened at AJVoIP side resulting in accumulation of packets.
Dropping the first few packets will make the playback of the rest more smooth, but might result in some loss of audible media (usually up to 100 milliseconds)
Possible values: -1: Auto Guess (default), 0: No, 1: Yes

maxjitterpackets

(int)
You can limit the jitter buffer size with this setting.
With the jittersize left as default (3) the maximum buffered packet count is limited to 8, so you might set this parameter to a lower value.
One packet means a received udp packet which might contain one or more audio frame.
For example when using G.729 the typical media stream are with 2 frames/packet. Each frame is 10 msec length.
A jitter limitation of 5 would mean maximum 100 msec to be cached. (while the default setting would allow 8 packet which means 160 msec)
Default value is 99 (no limitation)

useencryption

(boolean)
Set to true for encrypted communication (both media and signaling)
This has nothing to do with standard SIP encryption and it works only with mizu server tunneling module.

callreceiver

(int)
How to handle incoming calls.

Possible values:

-1: auto

0: disable all incoming calls

1: enable incoming calls (while app is running; no background listener)

2: enable also background calls (push or service implemented)

Default is -1.

~~*enablepush*~~

(int)
This parameter is deprecated and is to be used only for the softphone builds.
Use the pushnotifications parameter or the PushNotification API to enable/disable push notifications.
~~Specify whether to enable push notifications by default.~~
~~Possible values:~~
~~-1: auto~~
~~0: disabled~~
~~1: enabled auto~~
~~2: enabled direct to SIP server~~
~~3: enabled via gateway Default: -1~~

~~If set to 1, then will try push with your server (as described in RFC 8599 so your server must answer to REGISTER with Feature-Caps: *;+sip.pns="fcm") and if fails then it will try via push gateway.~~
For more details see the push notifications FAQ.

pushnotifications

(int)
Enable push notifications via as Google Firebase FCM.
Possible values:

-1=auto
0=disabled
1=enabled auto
2=enabled direct (with your SIP server if it has push support)
3=enabled via gateway

Default is -1 which means auto enable when required and possible. If FCM is available(if I have json file configured) and background service is not 1 AND there was incoming calls in the last 120 days or installed less then 120 days ago; otherwise it means no.
We provides push service (free tier) for all our customers enabled by default (via gateway if your server doesn't support push notifications).

To configure push notifications, you might use the PushNotification API instead of these parameters.
For more details see the push notifications FAQ.

fcmgateway

(string)
Specify FCM gateway to be used if a push gateway have to be used (instead of directly with SIP server).
Default is: fcm.webvoipphone.com:35060

To configure push notifications, you might use the PushNotification API instead of these parameters.
For more details see the push notifications FAQ.

packagename

(string)
Project ID. This is your application package name, used for export and upload to google market to be used also for push notifications.
Default is: empty or loaded from your app context or com.mizuvoip.mizudroid.app with mizu gateways.

To configure push notifications, you might use the PushNotification API instead of these parameters.
For more details see the push notifications FAQ.

fcmclientid

(string)
Your app token (token ID or Project ID returned by registering to the notification service) which can be received from your OS push/apns/fcm API.
Default is empty string.

To configure push notifications, you might use the PushNotification API instead of these parameters.
For more details see the push notifications FAQ.

pushtype

(int)
Specify what protocol to use for push notifications
Possible values:
-1: auto (1 with mizu servers, 2 with others or not sure)
1: mizu proprietary X-MPUSH
2: standard RFC 8599
3: force RFC8599 even with our servers or secondary
4: both
Default is -1

In case if you are using a Mizu server, the AJVoIP can use the mizu proprietary protocol for push binding: X-MPSUH: a:packagename:clientid.
Otherwise (with any third-party SIP servers) you should use the Push Notification for SIP standard as described in RFC 8599 which will send pn-* contact tags (Media Feature Tags) like pn-provider=fcm;pn-param=PACKAGENAME;pn-prid=TOKEN and will expect Feature-Caps: *;+sip.pns="fcm" in the answer).

For the details about SIP push notifications see the "SIP signaling" chapter in the VoIP Push Notifications guide.
For more details see the push notifications FAQ.

runservice

(int)
Set to 1 or 2 if your app is running as a service (or you are running the sipstack from a service) to enable some related optimizations.

This might be necessary if your apps needs to be always available for incoming calls but for some reason you don't implement push notifications.
Possible values:
-1: auto
0: no
1: background service
2: foreground service

Default: -1

Note: background services are heavily limited since API level 26 but AJVoIP implements all possible workarounds to bypass this limitation. Foreground services are more reliable since API level 26. Use a foreground if the always visible notification is acceptable for your user-case.
More details can be found here.

scheduledwakeup

(int)
Enable scheduled wakeup if running as a background service to help with background service limitations and doze/standby limitations.
Possible values:
0: Never
1: Yes if running as service
2: Always force

Default: 1

More details can be found here.

serviceclassname

(String)
Used for scheduled wakeup.
Set this to the name of your android service class that will run in the background to be able to receive incoming calls.
Default value: empty

Example:
SetParameter("serviceclassname", "com.mycompany.appname.mainclass.myservice");

askforwhitelist

(int)
Might ask the user to white-list your app if running as a background service to help with background service limitations and doze/standby limitations.
Possible values:
0: No
1: Yes if running as service
2: Always force
Default: 1

More details can be found here.

maxlines

(int)
1: single line
2: unlimited
Default is: 2

fastexit

(int)
Set cleanup speed and wait times.
0: slow but will more care about un-registration and cleanup
1: fast
2: very fast with no unregister
Default is: 1

secondarystatus

(int)
Specify if STATUS notifications should be sent also from the extra endpoints.
Possible values:
0: no (don't send status from secondary endpoints)
1: yes (send status also from secondary endpoints)
Default is 0.

events

(int)
Defines the level of notifications:
0=no notifications,1=status and cdr,2=important events,3= all logs including SIP signaling messages (depending also on loglevel).
Default is 2.

stats

(int)
Set to a value in seconds if you wish to receive extended periodic statistics for each line (STATUS notifications).
Default is 0 (no periodic statistics)

notificationssingleton

(int)
Specify if to reuse the SIPNotification objects.
-1: auto
0: separate SIPNotification object for all function calls
1: one SIPNotification object to be reused
Default is -1 (defaults to 1)

If set to 1, you will not be able to reuse the SIPNotification (e) object later in your code as it will be always overwritten with the latest.

notificationeventthread

(int)
Notification event threading.
-1: auto
0: from the main thread only (thread safe but slower with around 20 milliseconds delay)
1: from the caller thread (faster but not thread safe. the user must synchronize; don't access your GUI directly from the notification events)
Default: -1 (defaults to 0)

notificationevents

(int)
Notification events vs string send vs string polling.
To get the event notifications from the voip SDK to your application, you can use one of the following methods:

- Notification events using the SetNotificationListener to subscribe (to receive SIPNotification objects)
- GetNotificationStrings (to blocking receive notification strings in a separate thread)
- PollNotificationStrings (polling for notification strings)

Specify which method to use with this parameter. Possible values:
-1: auto/on demand
0: don't use polling
1: use polling
2: use polling
3: use polling only
4: use SIPNotification event objects only
Default is -1

If set to -1 (default), then

- will turn to 0 on first notification events
- will turn to 3 on first PollNotificationStrings
- will turn to 4 on first SetNotificationListener function call

We recommend to use the newly implemented SIPNotification events (SetNotificationListener). The other methods are deprecated, but will remain supported as-is for backward compatibility.

FAQ

How to get my own Android VoIP SDK?

1. Have a look at the description and pricing from the SIP library home page
2. Download and try the demo from here. You might start with the quick intro and then read trough this documentation.
3. Contact Mizutech at info@mizu-voip.com with the following details
   -your VoIP server(s) address (ip or domain name). This will be hardcoded in your release; otherwise anybody could just download it from your and use as it owns)
   -your company details for the invoice (if you are representing a company)
4. Mizutech support will send your own AJVoIP build within one workday on your payment.
   The payment can be made from the home page pricing grid with paypal or credit card or by wire transfer.

What about support?

All licenses include also a support plan in the cost.
The support is done mostly by email. For the gold license we can also offer 24/7 phone emergency support.
Maintenance upgrades are also free as included with your license plan.
Email to info@mizu-voip.com with any issue you might have.
Guaranteed supports hours depend on the purchased license plan and are included in the price.
Once your initial 1-4 years support period expires, it can be extended for around $600 (This is optional. There is no need for any support plan to operate your AJVoIP as the library is shipped with life-time license).

Note:

- *We can't guarantee that all the listed features will work correctly in your environment (your SIP servers / integrated with your software). Please test the demo/trial version before purchase to make sure that all the functionality critical to your use-case works correctly.*
- *Our support answer time is usually up to one work-day for issues related to core functionality (such as connect/register and voice calls). We are trying to resolve all such issues as soon as possible. Issues affecting non-core features (such as call hold or presence) might be handled only in the upcoming new releases.*
- *Direct support is provided for the common features (voice calls, chat, dtmf and others) and not for extra features (such as presence or video). Direct support is not provided for the issues described in the known limitations section. If you have some special must-to-have requirement, we recommend to test with the demo version before purchasing your license. However we appreciate all bug reports and we are constantly working on our software shipping new features, improvements and bug fixes with each new release.*
- *Mizutech doesn't provide direct server side support. Although AJVoIP is known to work well with all common SIP servers (including Asterisk, FreeSWITCH and many others), your servers have to be managed by you and we are not responsible for proper configuration of your servers. However, in case if there is any incompatibility issue then we might be able to help you with the resolution if you send us the related logs.*

What I will receive once I have made the payment for Android VoIP SDK?

You will receive the followings:

- The AJVoIP library itself. This is one single file usually named as "AJVoIP.aar"
- AJVoIP.jar: optional, if your IDE/tools doesn't support .aar library files
- ajvoip-sources.jar: optional file with the API/interface source/documentation
- Latest documentations
- Usage examples
- Invoice (on request or if you haven't received it before or during payment)
- Support on your request according to the license plan

More details.

Can Mizutech do custom development if required?

Yes, please contact us at info@mizu-voip.com.

Note: Please contact us only with AJVoIP related or VoIP specific projects. We are highly specialized for VoIP development thus we will not accept general app development projects which can be done by any developer or team (We don't handle tasks such as building user interface or design elements).

Is it working with any VoIP servers?

Yes. The VoIP SKD uses the SIP protocol standard to communicate with VoIP servers and sofswitches. Since most of the VoIP servers are based on the SIP protocol today, AJVoIP should work without any issue. AJVoIP have been tested with numerous other SIP stack (software/hardware) and we invest extra effort to achieve maximum compatibility even with misbehaving SIP implementations.
If you have any incompatibility problem, please contact info@mizu-voip.com with a problem description and a detailed log (loglevel set to 5). For more tests please send us your VoIP server address with 3 test accounts.

How is the call quality?

The library is capable for "perfect" voice quality if that is possible in your environment (server/peer codec support, network link quality, etc). Otherwise it aims to produce the best possible call quality achievable depending on the circumstances.
AJVoIP includes wide-band OPUS and Speex codec which are capable to offer the best possible call quality (MOS score of 6). This means that the call quality can be the same or better than for popular apps like Skype.
Of course, these have to be supported also by the peer to be used. PSTN networks usually has support only for narrowband codec (such as G.729, GSM, iLBC or G.711), so the call quality will be lower here (but still better than traditional phones if the user connection fulfills the minimal bandwidth and quality required for VoIP).
The media stack implements best practices for network and audio device handling to minimize the total delay.
A wide range of audio enhancement algorithms are also implemented such as AEC, denoise, PLC and AGC.
AJVoIP is also capable to auto-fine-tune itself to the circumstances by default (based on device capabilities, CPU speed, network bandwidth/quality, peer/server capabilities), but you can also force any settings with the parameters (such as codec, framesize, aec, etc).

Is AJVoIP using any Mizutech service or will contact Mizutech servers?

The Android SIP SDK will connect to your voip server directly. No any intermediary app server is involved.
AJVoIP will not "call to home" and will not send any sensitive information to Mizutech servers.
AJVoIP might contact Mizutech servers for the following reasons:

- demo versions might contact Mizutech licensing servers time to time. This is completely removed in final builds (after your order)
- AJVoIP will use a random stun server by default hosted by Mizutech. This "fast stun" protocol is usually not required for normal functionality so you might just disable it (set the "use_fast_stun" parameter to 0) or set the "stunserver" parameter to your stun server. (However these can improve the connectivity if your VoIP server is not NAT friendly so better to leave it as is. Mizu services (non) availability can't alter the usability of your product)
- Push Notifications: if you don't implement your own push notification service then AJVoIP might use the Mizu push notification gateway which we offer for all our customers (free tier). This can be disabled or replaced with your own implementation.

Is there any way to get the source code?

The Android VoIP SDK is a close-source application. The source code is available only for internal usage at an additional higher cost.

How to register?

SIP registration means connecting to your SIP server and authenticating. From this procedure your SIP server will learn your application address and can route incoming calls to your application (or other sessions, such as chat, presence, voicemail, etc).

The SIP stack auto-start behavior can be altered with the startsipstack setting or you can use the Start function to launch the instance explicitly.
When started, the SIP stack by default (unless you set the register parameter to 0) will automatically register to your SIP server if you configured the SIP account details (serveraddress, username, password and any other parameters that might be required such as the proxyaddress and sipusername).

Otherwise you might disable auto-start (startsipstack) and/or the auto register (register), pass the above parameters dynamically from your code and use the Start and/or the Register function to initiate the start/connect/register procedure.

Registration success of failure can be obtained from the REGISTER notifications (especially useful if you are using multiple accounts as this notifications are sent separately per account.

The registration success or failure can be also obtained from the STATUS notifications as described here.
The registered state can be requested with the IsRegistered or IsRegisteredEx API.
The failure reason can be also obtained (instead of the above STATUS notifications) by using the GetRegFailReason API.

Once registered, AJVoIP will automatically maintain the registration state by resending the REGISTER request before it expires. It will also auto reconnect if there are any errors such as network connection problems, service availability, etc.

Check this FAQ point if you are having problems with connect/register.

Extra accounts format

This FAQ point explains the accounts string format which you can use with the "old-style" extraregisteraccounts parameter and the RegisterEx function. Otherwise see the Accounts chapter instead for multi-account management.

You can configure multiple (secondary) accounts using the extraregisteraccounts parameter or the RegisterEx function passing a string with the accounts fields. Multiple accounts separated by semicolon.
The accounts must be passed as SIP URI's or with the following fields (fields are to be separated by comma):

serveraddress, username, password, registerinterval, proxyaddress, realm, sipusername, displayname, transport, main, fcm, enabled

Most of these fields are optional. The mandatory parameters are the username, password (and the serveraddress if that it is different from your main SIP server). See the extraregisteraccounts using parameter for more details.

The SIP accounts can be configured also by parameters in the following way (old style, not recommended anymore):

serveraddress, username, password, registerinterval: primary account
serveraddress2, username2, password2, registerinterval2: second account
…
serveraddressN, usernameN, password, registerintervalN: N account

Or via the RegisterEx(String accounts) API call where the accounts are passed as string in the following format:

server,usr,pwd,…;server2,usr2,pwd2,…;
(accounts separated by ; and parameters separated by ,)
Example: mysipclient.RegisterEx("myserver1.com,user1,pwd1;myserver2.com,user2,pwd2,120,,username2");
Or by using SIP URI's: mysipclient.RegisterEx("user1:pwd1@myserver1.com; user2:pwd2@myserver2.com");

Or via the "extraregisteraccounts" parameter which have to be set like this:

server,usr,pwd,…;serverN,usrN,pwdN,…;
or by using SIP URI's: usr1:pwd1@server1;usr2:pwd2@server2;
Example: mysipclient.SetParameter("extraregisteraccounts","myserver2.com,user2,pwd2;myserver3.com,user3,pwd3,120,,username3");

Up to 99 secondary accounts can be used this way (let us known if you need more).
Only the serveraddress/username/password must be set, the rest are optional, defaulting to the global setting (except the ival/registerinterval parameter which will default to 3600/one hour if not set). All other parameters are applied globally for all accounts (there is no per account profile).
See the Accounts chapter for more details.

How can I make a call?

- Make sure that the "serveraddress" parameter is set correctly (otherwise you will be able to make calls only to direct SIP URI).
- Optionally: Register to the server. This can be done automatically if the "username" and "password" parameters are preset. Alternatively you can register from API (Register) or just let the user to fill in the username/password fields and click on the "Connect" button. If AJVoIP is registered, then it can already accept incoming calls (it will do it automatically, or you can handle incoming calls from your application, or you can entirely disable incoming calls)
- Now you can make outgoing calls in the following ways:
  - Automatically with AJVoIP start. For this you will have to preset the username, password, autocall and callto parameter. Then AJVoIP will immediately launch the outgoing call when starts (usually with your page load)
  - Just let the users to enter a called number and hit the "Call" button
  - or just call the **Call** function (from user button click or from your business logic)

Optionally on call you might set the following flags (for example if you have a separate call page activity, then you might set these from onCreate):

```
getWindow().addFlags(WindowManager.LayoutParams.FLAG_DISMISS_KEYGUARD);
getWindow().addFlags(WindowManager.LayoutParams.FLAG_SHOW_WHEN_LOCKED);
getWindow().addFlags(WindowManager.LayoutParams.FLAG_TURN_SCREEN_ON);
getWindow().addFlags(WindowManager.LayoutParams.FLAG_KEEP_SCREEN_ON);
if(Build.VERSION.SDK_INT >= 24) getWindow().setSustainedPerformanceMode(true);
```

Check this FAQ point if you are having problems initiating calls.

How to handle incoming calls?

To be able to receive incoming calls, make sure that you are registered to your SIP server first. Then you can easily test for example by using any third party softphone and make calls to your android SIP username (or extension id or full URI, as required by your server).

In your project all you have to do is to watch for incoming ringing STATUS notifications and from there you can show any user interface (activity) as you wish.

If you follow the example code, you receive the notifications in the public void ProcessNotifications(String msg) function.
You can find the possible strings that can be received at the "Notifications" chapter in the documentation.

To catch incoming calls, you have to look (string parse) for STATUS messages with "Ringing" text (second parameter is the state text) and the endpointtype (forth parameter) set to 2 (means incoming).
For example the following status means that there is an incoming call ringing from 2222 on the first line:
STATUS,1,Ringing,2222,1111,2,Katie,[callid]
Example code:

```
class MyNotificationListener extends SIPNotificationListener
{
   public void onStatus( SIPNotification.Status e)
   {
      if(e.getStatus() == SIPNotification.Status.STATUS_CALL_RINGING && e.getEndpointType() == SIPNotification.Status.DIRECTION_IN)
      {
         Log.v("\tIncoming call on line "+Integer.toString(e.getLine())+" from "+ e.getPeer());
         //mysipclient.Accept(e.getLine()); //auto accepting the incoming call
      }
   }
}
```

Details about the incoming call can be obtained the STATUS notification itself and/or using any of the following functions:

- GetLineDetails(-1)
- GetSIPMessage(-1, 0, 1)
- GetLastRecInvite()
- GetCallerID(-1)
- GetIncomingDisplay(-1)

Incoming calls can be also automatically answered as described here. Otherwise just use the Accept to connect the call (or the Reject to disconnect).

AJVoIP is capable for automatic line management so if you don't wish to handle lines explicitly in some specific way, then you can ignore all notifications, except those where the line (first parameter) is -1 (the global state).
If you wish to auto accept all incoming calls, you can set the autoaccept parameter to true. Otherwise use the **Accept** API to connect the call (or the Reject API to disconnect). Check the parameters to find out more call divert settings, such as auto-answer or forward if needed.

Check this FAQ point if you are having problems receiving incoming calls.

Project dependency issues

Check the followings in case if you are unable to build your project because dependency issues.
For example you might receive an error something like this:
Unable to resolve dependency for ':app@debug/compileClasspath': Failed to transform file 'AJVoIP.aar' to match attributes {artifactType=android-exploded-aar} using transform ExtractAarTransform

This has less to do with our aar and it is a gradle bug.
Possible solutions:

1. Clean project
   Rebuild project
2. https://github.com/golang/go/issues/23307
3. https://stackoverflow.com/questions/45138391/failed-to-transform-file-45907c80e09917e1b776adf038505958-to-match-attributes
4. Make sure your machine is connected to the internet.
   Go to File -> Settings & expand Build, Execution, Deployment -> Gradle -> UNCHECK Offline work -> OK
   Go to File -> click on Invalidate Caches / Restart -> Invalidate & Restart
5. Upgrade android studio and tools to latest version
6. Just create a new project and add our aar to it and the code from MainActivity.java.

Failed outgoing calls

First of all you should check the logs (logcat) to see any ERROR and the SIP signaling.
For outgoing call you should see an INVITE sent to server (or peer) and a 2xx response (usually 200 OK) on call connect or other response (4xx, 5xx, 6xxx) on call failure. Some server requires previous successful registration (REGISTER) to allow calls.

Some servers has problems with codec negotiation (requiring re-invite which is not support by some devices). In these situations you might enable only one codec which is supported by your server by setting the codec parameter (for example codec: PCMU).

AJVoIP by default doesn't allow cross-domain calls. For example if you are registered as a@A.com (to A domain) then you will not be able to make direct calls to b@B.com (to B domain). Contact mizutech support if you need this use-case to remove the limitation.

Can't connect to SIP server

If the SDK cannot connect or cannot register to your SIP server, you should verify the followings:

- You have set your SIP server address:port correctly
- Make sure that you are using a SIP username/password valid on your SIP server
- Make sure that the autostart parameter is true and the register parameter is 1 or 2. Otherwise use the Start() and/or Register() functions explicitly.
- Make a test from a regular SIP client such as mizu softphone or x-lite from the same device (if these also doesn't work, then there is some fundamental problem on your server not related to our library or your device firewall or network connection is too restrictive)
- Check if some firewall, NAT or router blocks your device or process or the SIP signaling
- Check the logs
- If there are no any answer for the REGISTER requests, turn on your server logs and look for the followings:
    - The REGISTER reaches your server?
    - Is there any response triggered (or some error)?
    - If answer is triggered, is it sent to the correct address? (it should be sent to the exact same address from where the server received the REGISTER request)
    - The answer packet reach the client PC? You can use Wireshark to see the packets at network level.
- Send us a detailed client side log if still doesn't work with loglevel set to 5 (from the browser console or from softphone skin help menu)

Calls are disconnecting

If the calls are disconnecting after a few second, then try to set the "invrecordroute" parameter to "true" and the "setfinalcodec" to 0.
If the calls are disconnecting at around 100 second, then most probably you are using the demo version which has a 100 second call limit.

See the CDR and the log about the details.

Blank screen or freezing on calls

Ask for RECORD_AUDIO permission from your app before to make calls.

Known limitations

- No technical support for Bluetooth related functionality (we provide it "as is" since incompatibility issues might occur due to the high device fragmentation)
- The video module is provided "as-is" with no specific support. Find more details here.
- AEC is a best effort algorithm and it might not work (or it might work only partially) in some circumstances. There is no any quick fix for underperforming AEC
- The full STUN specification is not implemented due to size and performance considerations (a fast and light STUN version is used when needed which always outperforms the standard STUN implementations with no drawbacks)

Please test with the demo version first before to purchase a license, to make sure that it fulfills your requirements and it works correctly in your environment.

Media statistics

Media statistics means RTP/RTCP level details, which can be used to analyze call quality or to display a call quality indicator.

- Basic details: are received with the STATUS notifications such as total rtpsent, rtprec,rtploss, rtplosspercet and server statistics if reported. These and more is also received with the RTP LOG notification.
- Quality reports: can be sent as RTPSTAT notifications if enabled by the rtpstat parameter.
  Alternatively it can be requested with the RTPStat function call.
- Logs: You can also see more details in the logs such as total rtp statistics reports for calls longer then 7 seconds if the loglevel is at least 3 as EVENT, rtp stat: sent X rec X loss X X% or the RTP notification. If you set the "loglevel" parameter to at least "5" than the important rtp and media related events are also stored in the logs. You might search for "call details" at the log which also includes media related reports after each call.

Voice activity detection

AJVoIP has built-in VAD (voice activity detection) algorithm included.
The talking/silence state of both the local user and the remote peer can be reported.
To enable VAD reports, set the vad parameter to 4.
Once this is set, you can receive the VAD state either by calling the VAD function or by the VAD notifications if the vadstat parameter is set to 3 or 4.
For the notifications interval you might also adjust the vadstat_ival parameter after your needs (default is 3000 in milliseconds, which means a VAD report in every 3 seconds).
If the vadstat is set to 4 then it will report the state also when changed from silence to speaking or inverse.

NAT settings

In the SIP protocol the client endpoints have to send their (correct) address in the SIP signaling, however in many situations the client is not able to detect it's correct public IP (or even the correct private local IP). This is a common problem in the SIP protocol which occurs with clients behind NAT devices (behind routers). The clients have to set its IP address in the following SIP headers: contact, via, SDP connect (used for RTP media). A well written VoIP server should be able to easily handle this situation, but a lot of widely used VoIP server fails in correct NAT detection. RTP routing or offload should be also determined based in this factor (servers should be always route the media between 2 nat-ed endpoint and when at least one endpoint is on public IP than the server should offload the media routing). This is just a short description. The actual implementation might be more complicated.

You may have to change Java VoIP toolkit configuration according to your SIP server if you have any problems with devices behind NAT (router, firewall).
If your server has NAT support then set the use_fast_stun and use_rport parameters to 0 and you should not have any problem with the signaling and media for AJVoIP behind NAT. If your server doesn't have NAT support then you should set these settings to 2. In this case AJVoIP will always try to discover its external network address.

Example configurations:
If your server can work only with public IP sent in the signaling:
-use_rport 2 or 3
-use_fast_stun: 1 or 2 or 4 (recommended is 2)

If your server can work fine with private IP's in signaling (but not when a wrong public IP is sent in signaling):
-use_rport 9
-use_ fast_stun: 0
-optionally you can also set the "udpconnect" parameter to 1

Asterisk is well known about its bad default NAT handling. Instead of detecting the client capabilities automatically it relies on pre-configurations. You should set the "nat" option to "yes" for all peers.
More details:
http://www.voip-info.org/wiki/view/NAT+and+VOIP
http://www.voip-info.org/wiki/view/Asterisk+sip+nat
http://www.asteriskguru.com/tutorials/sip_nat_oneway_or_no_audio_asterisk.html

For FreeSWITCH you might set the "NDLB-force-rport" and "aggressive-nat-detection" values to "true" in the sip_profiles configuration.
More details here.

Server failover/fallback

Use the following settings if you have 2 voip servers:

- serveraddressfirst: the IP or domain name of the first server to try
- serveraddress: the IP or domain name of the next server
- autotransportdetect: true
- enablefallback: true

In this way java voip framework will always send a register to the first server first and on no answer will use the second server (the "first" server is the "serveraddressfirst" at the beginning, but it can change to "serveraddress" on subsequent failures to speed up the initialization time)

Alternatively you can also use SRV records to implement failover or load balancing.

I have call quality issues

It is normal to get cropping audio with debug builds running from the IDE on your phone and especially on emulator. Call quality during test sessions can be improved by lowering the log level, disabling AEC and denoise and forcing a low complexity codec such as PCMU. If you wish to focus on call quality tests then you should build a release version and run it normally on your phone, not from your development environment such as the Android Studio IDE.

Call quality is influenced primarily by the followings:

- Codec used to carry the media (PCMU, PCMA requires the less CPU but more bandwidth; G.729 and OPUS wideband requires more CPU but it is more optimized for bandwidth). The best possible call quality can be achieved with OPUS wideband if you are using a release build
- Network conditions (check also your upload packet loss/delay/jitter)
- Hardware: enough CPU power and quality microphone/speaker (try a headset, try on another device)
- AEC and denoise algorithms increases the CPU load especially under debug
- Missing (disabled) AEC and denoise might result in suboptimal call quality
- Log details: high loglevel might decrease the sound quality. You should set the loglevel to 5 or lower
- Debugging: release versions have better quality due to optimizations

If you have call quality issues then the followings should be verified:

- whether you have good call quality using a third party softphone from the same location (try X-Lite for example). If not, than the problem should be with your server, termination gateway or bandwidth issues.
- make sure that the CPU load is not near 100% when you are doing the tests
- watch for RTP statistics
- make sure that you have enough bandwidth/QoS for the codec that you are using
- change the codec (disable/enabled codec's with the codec/prefcodec or the use_xxx parameters)
- run your app on a real device (your phone) and not on the simulator
- force a low complexity codec such as PCMU for your debug builds and test sessions
- check the AJVoIP logs (check audio and RTP related log entries with loglevel 5)
- set the loglevel to 1 (or up to 5)
- wireshark log (check missing or duplicated packets)

Performance optimizations

The SDK by default comes with optimal default value but in some circumstances you might need to further optimize for performance.
If you are running the SDK on some ancient device or any device with low-end CPU or lower-powered devices, you can optimize the performance with the following settings:

- Make sure that no other processes on your device are running with abnormally high CPU usage (remove or optimize other processes that are unnecessarily consuming the CPU)
- Prioritize low computational codecs with the following settings:
  use_pcmu=3 use_pcma=2 use_g729=1 use_gsm=2 use_speex=1 use_speexwb=1 use_speexuwb=0 use_opusnb=1 use_opuswb=1 use_opusuwb=0 use_opusswb=0 use_ilbc=1
- Set the loglevel to 1 (High log levels will slow down the SDK. But don't set it to 0)
- Set the following parameters to 0 to lower the number of threads (especially if there are only 1 or 2 CPU cores available for AJVoIP):
  audiorecthread, audioplaythread, audioplaythread2
- Disable extra audio processing by setting the followings to 0: aec, aec2, denoise, agc, silencesupress
  (In case if you don't wish to completely disable AEC, then just fine-tune the aec, aec2 and aectype settings. Avoid full software aec.)
- Increase the audioqueuemaxsize value. Default is 40. Increasing it can help with short CPU spikes and packet bursts but might increase the playback delay
- Additional optimizations:
  - codecframecount: 2 (or even 4; might be incompatible with some bogus servers)
  - voicerecording: 0
  - syncvoicerec: 0
  - vad: 1
  - plc: false
  - rtcp: false
  - aectype: none
  - enablepresence: 0
  - registerinterval: 300

    - keepaliveival: 50
    - timer: 20 (not recommended)
    - timer2: 20 (not recommended)
    - Set the cpuspeed parameter to 700 or less (or some other value between 300 and 2000)
- In case if you are using the SDK for some special purpose with on way audio then you should to set the useaudiodevicerecord or useaudiodeviceplayback parameters to false. In this case you might also set the mediatimeout parameter to 0.

It might be possible that one or two of the above already fixes any performance problems on your devices. In that case it is not necessary to change all of these settings.

I have one way audio

1. Review your server NAT related settings (set nat=yes in Asterisk config if you are using Asterisk based solution)
2. Set the “setfinalcodec” AJVoIP parameter to 0 (especially if you are using Asterisk or OpenSIPS)
3. Set use_fast_stun, use_fast_ice and use_rport to 0 (especially if you are using SIP aware routers). If these don’t help, set them to 2.
4. If you are using MizuVoIP server, set the RTP routing to “always” for the user(s)
5. Make sure that you have enabled all codec’s
6. Make a test call with only one codec enabled (this will solve codec negotiation issues if any)
7. If you still have one-way audio, please make a test with any other softphone from the same PC. If that works, then contact our support with a detailed log (set the” loglevel” parameter to 5 for this)

No ringback tone

Depending on your softswitch configuration, you might not have ringback tone or early media on call connect.
There are few parameters that can be used in this situation:

- set the “changesptoring” parameter to 3
- set the “natopenpackets” parameter to 10
- set the “earlymedia” parameter to 3
- change the use_fast_stun parameter (try with 0 or 2 or 4)
- set the “nostopring” parameter to 1
- set the “ringincall” parameter to 2

One of these should solve the problem.

The remote party hear itself back (echo)

About

Echo suppression and cancellation are special voice processing algorithms to prevent the audio played on the speaker to be recorded and sent back to the other party resulting in echo.

The perceived echo depends on many factors:

- The AEC algorithms used by AJVoIP and on the other end
- Network delay
- Device hardware capabilities (on both the AJVoIP side and the other side)
- Speaker device
- OS audio processing and audio driver
- Room characteristics

Please note that when you hear echo with AJVoIP, the culprit is usually the other end as there is nothing to solve the echo problem if it is already present in the incoming stream. The solution in this case is to use AEC capable devices also for the peers.

AEC is important only if a speaker/loudspeaker/speakerphone is used. If the user will make a normal call with off-speaker or using a headset than echo is generated only if the internal speaker or the headset is broken.

Also it is possible that some software is capable to suppress echo under specific circumstances and other software might fail in the same circumstances but might have better success in other circumstances.

Success rate

Please note that echo cancellation is not an exact science and there is no perfect algorithms for 100% echo cancellation.

AJVoIP includes five different AEC/AES algorithms:

- software: two separate software based AEC algorithm (requires extra CPU processing)
- hardware: android hardware aec capabilities (not supported on all phones)
- fast: a fast software aec implementation (used preferably on slow/old devices or in addition for the above algorithms)
- volume: this will decrease the volume when speech detected from other end (using VAD)

After our tests with numerous hardware the AJVoIP media stack is capable to eliminate more than 90% of the echo with around 92% success rate.

Settings

Echo cancellation in AJVoIP is usually turned on by default when necessary (depending on the playback device used and might be disabled by default on devices with very slow CPU such as ancient or low-budget phones).

You might adjust the following settings to fine-tune the AJVoIP side AEC if there is echo generated by AJVoIP in your environment:

- Set the aec parameter to 2.
- Set aectype parameter after your needs. For example you might set it to software,hardware,fast,volume to apply all suppression algorithms
- For better quality you can also set the denoise parameter to 1.

To completely disable AEC, set the aec parameter to 0 and the aectype to none. This might be used to save CPU time in circumstances when no echo is possible such as have only one way audio (like IVR calls) or there is no loudspeaker usage.

Support

We can't provide technical support for AEC related issues, because AEC is known to not work (partially or at all) in some circumstances and there is no any quick fix for these issues except the above mentioned settings.

Force loudspeaker

Use the following setting if you are using AJVoIP with specific devices which doesn't have an earspeaker and you wish to always force the loudspeaker:

- audiomanagermode: 0 (MODE_NORMAL instead of 3/MODE_IN_COMMUNICATION; audiomanagermode / AudioManagerSetMode)
- speakerphoneoutput: 5 (do nothing instead of 0/1/2; cfg_audiomode localaudiomode player only)
- speakerphoneplayer: 3 (default STREAM_MUSIC instead of modified 0/STREAM_VOICE_CALL;) cfg_speakerphoneplayer / getAltStream player only)
- focusaudio: 1 (disable focus)
- aspeakermode: 2 (loudspeaker instead of the default 0/auto)

This way AJVoIP will act like a simple media player to playback the audio streams and you don't need to use the SetSpeakerMode function anymore.

RTC video

Video requires RTC capabilities (handled automatically by AJVoIP or you can set your own webrtcserveraddress).

Details:
VoIP Video streaming is a moving target. In the past we had H.263, nowadays H.264 and VP8 but soon the industry will move to H.265, AV1 and VP9.
To avoid constant rewrite and optimizations to the latest preferred standards we decided to take full advantage of the WebRTC capabilities already found on all platforms. AJVoIP uses the WebRTC module offered by the Android browser via a WebView, thus always up to date but requiring RTC capabilities for SIP compatibility. This is handled with the default settings or you can use the webrtcserveraddress parameter if you wish to specify your own WebRTC server endpoint (websocket listener). A simple way to test video functionality is to make a WebRTC call from Chrome to check if it works as expected or at least test your Chrome browser WebRTC capabilities.

Video re-INVITE is not currently supported (usage with SIP devices which are capable to initiate or accept the video from start as upgrading a voice call to video will not work). RTC uses the latest standards for video negotiation and byte stream format thus might not be compatible with all SIP devices as it enforces a strict video codec negotiation that might not be supported by the peer device.

If video is your main use-case we recommend testing it first in your environment before to purchase to make sure that it works with your server/ devices. (You can make a quick test with the MizuDroid softphone downloadable from the Google Play as that is based on this AJVoIP library).
We provide the video module as-is with no specific support as we rely on the platform RTC capabilities here with the limitations mentioned above, but you can assume that android will always ship with the latest video tech which should provide optimal performance and compatibility with the latest devices.

Required permission: CAMERA
Codec support: depends on the RTC stack. Usually H264, VP8 and VP9.
Video related parameters: video, vcodec, video_bandwidth, video size parameters, webrtcserveraddress

Video related API's: VideoCall, AcceptVideo, RemoveVideo, SetVideoDisplaySize, IsIncomingVideo

How to close a video call

Once a video call is terminated, you will also need to remove or hide your video display user interface element. You can follow this sequence:

1. hangup the call by calling hangup() function (and/or catch the disconnected state of the call from the STATUS notications)
2. optional: wait 2 seconds if you need a CDR record about the video call
3. remove/unload webphone fragment from FrameLayout (resource ID of FrameLayout received in VideoCall)

Note: you might use RemoveVideo function to remove the video display, but the better way is to just hide/destroy your container.

Example code:

```
//disconnect the call
sipstackinstance.Hangup();

//if you need CDR, then execute the below code a bit later from a 2 second timer

// remove/destroy webphone fragment
FragmentTransaction ft = getSupportFragmentManager().beginTransaction();
ft.remove(finstance.getSupportFragmentManager().findFragmentById(fragmentResId)).commit();
```

Screen turns to black

Set the proximitysensor parameter to 0.

Chat is not working

Make sure that your softswitch has support for IM and it is enabled. The Java SIP client is using the MESSAGE protocol for this from the SIP SIMPLE protocol suite as described in RFC 3428.
Most Asterisk installations might not have support for this by default. You might use Kamailio for this purpose or any other softswitch (most of them has support for RFC 3428).

AJVoIP doesn't receive incoming calls

To be able to receive calls, AJVoIP must be registered to your server.
Once the SIP client is registered, the server should be able to send incoming calls to it.

The other reason can be if your server doesn't handle NAT properly: Please try to start AJVoIP with use_fast_stun parameter set to 0 and if still not works then try it with 2.

If you wish to be able to always accept incoming calls (even when the user is not actively using the app), then implement all-time availability as discussed below.

All-time availability

To be able to accept calls, IM and other notifications, the server needs to be capable to deliver the SIP signaling messages to your application.
Normally your application is "connected" to your SIP server with an active UDP or TCP register session thus the SIP server is capable to send call connect, IM and other messages via these streams.
New Android OS versions heavily restricts apps running in the background and usual applications will just stop running when not in foreground or actively used by the user.

To be able to receive incoming call while your app is in background, closed, standby or the device is in doze or sleeping state, you have two options:

- Run as a service
- Enable voip push notifications

Run as a service

One way to implement all-time availability (beside using push notifications) is to continuously run the sip stack, keeping your register session alive, always connected to your SIP server using UDP (with keep-alive to bypass NAT's) or a TCP connection. This is necessary because to be able to always accept incoming calls or text message (even when the app is not opened) your SIP server needs to be capable to deliver the INVITE/MESSAGE/other SIP messages to your application.

This should be used only if you don't wish to enable push notification for some reason or push notifications can't be used for your use-case for some reason.

This can be achieved by running the SIP stack from a service.
You need to have an android service running in the background and on an incoming call you can start your activity from that service using the below code:

```
Intent intent = new Intent(this, MyActivity.class);
intent.addFlags(Intent.FLAG_ACTIVITY_NEW_TASK);
startActivity(intent);
```

Running the app (or just the SIP stack) as service is a convenient mode to achieve availability and AJVoIP implements all best practices for battery optimizations thus requiring minimal CPU and energy while running in idle (slow down timers, auto-lowering the prioritiy of it's threads and other optimizations).

To be able to keep the register session alive the app needs to fulfill the followings conditions:

- actually running (not killed/closed or put in a dose or sleep state)
- capable to acquire wake lock
- capable to acquire wifi lock (if there is no mobile data or the user prefers Wi-Fi)

Background services are severely restricted in latest Android versions and apps are also affected by Doze and Standby limitations.
There are workarounds and good practices supported by AJVoIP and additional techniques that you might implement as described below.

The followings **should** implemented the followings in your app:

- Set the runservice AJVoIP parameter to 1 or 2
- Set the serviceclassname AJVoIP parameter to your service class name
- Run the service with START_STICKY and start the SIP stack from the service

The followings **might** be implemented/handled by your app:

- A simple what to circumvent the background service limitations since API level 26 (dosing/sleep) is to just run a foreground service (instead of a background service). This is a more reliable way to make it always available but a non-removable notification will be displayed while the app is running.
- You might also start your app service at boot (requires the RECEIVE_BOOT_COMPLETED permission. Since API level 26 only foreground services can be automatically started. More details: 1,2,3)
- From API level 26 (Android Oreo) the OS prevents the use of startService() method to start the service from background. If you call startService() from background on Android O, you will get an IllegalArgumentException. In this case you might just catch the exception and start a foreground service instead using the startForegroundService() API.
- From API level 29 (Q) you should only use a foreground service (background services are not allowed to run continuously for networking/audio as it would be required for VoIP)
- Explicitly ask the user to white-list the app from battery optimizations: ACTION_REQUEST_IGNORE_BATTERY_OPTIMIZATIONS. This is implemented also inside the AJVoIP library (see below), but better if you explicitly ask for the permission at first startup.
- Optionally you might implement a tricky technique to always restart your service if killed by the OS (just make sure to not restart if the user explicitly unregistered and stopped your application)
- Optionally check for isBackgroundRestricted and ask the user to release the restrictions

The followings are implemented/handled by AJVoIP:

- Acquire wake lock / wifi lock: by default this is handled by the AJVoIP itself with no further actions required from your side. It can be finetuned with the forcewifi, cpupartiallock and cpualwayspartiallock settings if for some reason the default behavior is not suitable for your use-case
- The SIP stack is also capable to auto wake-up by using the Android alarm manager capabilities. This will also help in some circumstances, but this kind of alarms are limited usually to a 10 minute intervals. You can enabled/disable this with the scheduledwakeup parameter.
- Ignore batter optimizations: latest Android versions might automatically enforce Doze or Standby. Whitelisting your application can be asked with the ACTION_REQUEST_IGNORE_BATTERY_OPTIMIZATIONS which can be done from your code or just set the askforwhitelist parameter to 1.
  Note: this is a dangerous permission and google might remove your app from the Google Play if used for no valid reason as described at the end of this page.
  In case if you wish to do this from your own app for some reason, set the askforwhitelist to 0 and use this copy-paste code:

```
PowerManager pmanager = (PowerManager) getSystemService(Context.POWER_SERVICE);
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M)
{
        if (!pmanager.isIgnoringBatteryOptimizations(getPackageName()))
        {
                //Ask the user to disable battery optimization
                Intent intent = new Intent();
                intent.setAction(Settings.ACTION_IGNORE_BATTERY_OPTIMIZATION_SETTINGS);
                startActivity(intent);
        }
}
```

More details here.

To summarize, there are the following ways to achieve always availability:

- Enable push notifications: Recommended method to enable always availability (discussed below). If enabled, then there is no need to run any service.
- Using a foreground service: This is a simple and reliable method with the drawback of a non-removable notification displayed on the device home screen
- Using a background service: A bit more tricky and workarounds required as describe above, but fully supported by AJVoIP
- Optional tricks such as auto restart and asking the user to reconfigure their phone settings to allow your app to run in the background
- Not using any service: You can just enable push notification support (discussed below), thus your app doesn't have to be run continuously (zero battery utilization)

VoIP Push notifications

VoIP Push notification is an effective solution to wake up/launch your closed or sleeping application on incoming SIP call or text message without the need to run your application in the background and to keep a persistent connection (or frequent UDP keep-alive or re-register) with the SIP server/proxy.

The SIP library has full support for push notifications which can be negotiated directly with your server (if your server has push support) or via gateway (if your server doesn't support push RFC 8599)

There are two ways to enable/disable/configure push notifications.

- Using the API: call the PushNotification function (recommended)
- Using parameters: configure the pushnotifications / fcmgateway / packagename / fcmclientid parameters.

Push notifications can use the mizu proprietary protocol (with mizu servers/gateways using the X-MPUSH header) or the Push Notification for SIP standard described in RFC 8599. The protocol to use is selected automatically by default or you can set it explicitly with the pushtype parameter or by the first parameter of the PushNotification function.

Push notifications works in the following ways:

1. Create a Firebase project for you app and download the google-services.json file to your project
2. Configure Firebase for your app and obtain an FCM token ID as described here
3. Configure AJVoIP with the PushNotification function (or with the related parameters)
4. AJVoIP will send your app package name and token ID with the REGISTER requests to your SIP server (or to the push gateway if your server don't have push notification support)
5. On incoming calls/chat your server/gateway will send a push notification first to wake-up your app and then your app will be able to process the incoming INVITE/MESSAGE/etc as normally.

All these steps are explained at the "AJVoIP"chapter in the VoIP Push Notifications guide.

Push notifications with your SIP server

Push Notification for SIP is described in RFC 8599.
If your server has push notification support then AJVoIP will send all the required details (Project ID, Registration token) with the SIP signaling (pn-* media feature tags in Contact) and your server must send a push notification to AJVoIP (via FCM) to wake up the app before incoming calls and it can also send IM chat messages as push notifications (along the usual SIP MESSAGE).
For more details see the instructions at the "AJVoIP"chapter from the VoIP Push Notifications guide (except the Mizu server/gateway related instructions, but similar things have to be done also on your server)
In case if your server doesn't have push support then you can still use push notifications via gateway as described below.

Push notifications via Mizu gateway or service

If you use your own (or other) SIP server which doesn't have push notification support (or you don't wish to use it's push capabilities) then you can use the mizu VoIP push notification gateway or free service to handle push notifications for your app (AJVoIP also comes with a free tier push service to handle push notifications).
In this case follow the instructions at the "AJVoIP -Mizu PUSH service" from the VoIP Push Notifications guide.

Push notifications with Mizu servers

Push notification support is included in all Mizu server side products. In case if you use a Mizutech server, then you can implement push notifications as described at the VoIP Push Notifications guide "AJVoIP" chapter.

More details about VoIP push notifications can be found here and here.

What is the best codec?

There is no such thing as the "best codec". All commonly used codec's present in AJVoIP are well tested and suitable for IP calls.
This depends mainly on the circumstances.
Usually we recommend G.729 since this provides both good quality and good compression ratio.
If G.729 is not available in your license plan, than the other codecs are also fine (GSM, speex, iLBC)
Otherwise the G711 codec is the best quality narrowband codec. So if bandwidth is not an issue in your network, than you might prefer PCMU or PCMA (both have the same quality)
Between AJVoIP users (or other IP to IP calls) you should prefer wideband codec's (this is why you should always leave the opus and speex wideband with the highest priority if you have calls between your VoIP users. These will be picked for IP to IP calls and simply omitted for IP to PSTN calls)
To calculate the bandwidth needed, you can use this tool. You might also check this blog entry: Codec misunderstandings

What is the default codec priority?

The codec priority order is calculated automatically based on circumstances (CPU, device capabilities, network). The AJVoIP library does a good job to always select the optimal codec set so you should change the codec priorities only if you have some very specific needs. The SIP library is also capable to automatically re-INVITE with changed codec set if the server or the peer device rejects the first INVITE due to codec incompatibility.

If you doesn't change the codec priorities with the parameters and you have enough CPU power and good network, than the default codec order will be the following (listed in priority order):

1. Opus wideband
2. Speex wideband
3. G.729
4. G.711 (PCMU/PCMA)
5. Opus narrowband
6. GSM
7. iLBC

The codec priorities are usually set like this by default:

1. Opus wideband (enabled low priority 2)
2. speex wideband (enabled low priority 2)
3. G.729 (enabled low priority 2)
4. PCMU (enabled low priority 2)
5. PCMA (disabled 1)
6. speex ultrawideband (disabled 1)
7. opus ultrawideband (disabled 1)
8. opus narrowband (disabled 1)
9. speex narrowband (disabled 1)
10. GSM (disabled 1)
11. iLBC (disabled 0)

To prefer a codec you just have to set it’s priority to 3 (use_myfaworitecodec=3).
This will automatically enable and put your selected codec as the highest priority one.

If you set all codec with the same priority, then the real priority will be the following:

1. Opus wideband
2. Speex wideband
3. G729
4. G711
5. Gsm
6. Opus narrowband
7. Speex narrowband
8. Ilbc (lowest priority)

Note:

- It is not guaranteed that the endpoints will consider the codec order but normally the other endpoint usually will pick up the first codec, or AJVoIP will pick-up the first in this list from the list of codec’s sent by the other peer
- Not all codec is available in the Basic and Standard license (The Advanced and Gold license contains all codec)

How to prefer one codec?

Method A:

Just set the prefcodec parameter.

Method B:

If you prefer to use the use_xxx parameters for the codec settings, then just set the desired codec priority to 3 and set the priority for all other codes to lower.
In this case you preferred codec will be used whenever the other endpoint supports it and other codec's are used only if otherwise the call would fail.
For example the following parameters will set g.729 as the preferred codec and will enable also pcmu and gsm:

```
-use_g729=3
-use_pcmu=2
-use_pcma=1
-use_gsm=2
-use_speex=1
-use_speexwb=1
-use_speexuwb=1
-use_opusnb=1
-use_opuswb=1
-use_opusuwb=1
-use_opusswb=1
-use_ilbc=1
```

Disconnect reasons

You can receive the call disconnect reason with the CDR notification or using the GetLastCallDetails or GetDiscReasonText API.
There are endless possibilities why a call can fail, the most common reasons are listed below.
The disconnect reasons are reported in the following format: code text. (So you have the text after a space)
Code:

Is a SIP disconnect request or answer code including BYE, CANCEL or any SIP response code above 300.
If no disconnect message were received or sent then the code is -1 or empty/not set.

Text:

The text is one of the followings:

1. local disconnect reasons (listed below)
2. disconnect reason extracted from SIP Warning or Reason headers
3. response text extracted from the first line of SIP responses (textual representation of the response code)
4. textual representation of the disconnect code

The local disconnect reasons can be one of the followings (extra details might be appended and new disconnect texts might be added in the future):

- notaccepted
- User Hung Up
- bye received
- cancel received
- authentication failed
- endpoint destroy
- no response
- call setup timeout
- ring timeout
- media timeout
- endpoint timeout
- tunneling calltime limit
- call max timer expired
- max call time expired
- max speech time expired
- not acked connection expired
- disconnect on transfer
- transferalways
- transfer timeout
- transfer fail
- transfer done
- transfer terminated
- transfer (other)
- refer received
- cannot start media
- not encrypted
- srtp fail
- srtp init fail
- disc resend
- failed media
- forward
- forwardonbusy
- forwardonnoanswer
- forwardalways
- rejectbusy
- rejectonphonebusy
- call rejected by peer
- rejected by the peer
- rejected
- autoreject
- autoignore
- ignored

The SIP disconnect codes (3xx, 4xx, 5xx, 6xx) are described in the SIP RFC.

With the CDR notification ("Discparty") or the GetLastCallDetails ("Disc By") you will also receive the party which was initiated the disconnect.
This can be one of the followings:

- 0: unknown/not set
- 1: local AJVoIP
- 2: remote peer
- 3: undefined/timeout

You can also get the exact disconnect reason / disconnect message from the SIP signaling using the GetSIPMessage() function.
For example to extract the disconnect reason from the last incoming message, you can use something like this:

```
String sipmsg = GetSIPMessage(-1, 0, 0) ;
String disconnectcode = "";
if(sipmsg.startsWith("BYE ")) disconnectcode = "BYE";
else if(sipmsg.startsWith("CANCEL ")) disconnectcode = "CANCEL";
else if(sipmsg.startsWith("SIP/") && sipmsg.contains(" "))
{
        sipmsg = sipmsg.substring(sipmsg.indexOf(' ')+1);
        if(sipmsg.contains(" "))
        {
                sipmsg = sipmsg.substring(0, sipmsg.indexOf(' ')).trim();
                int codenum = -1;
                try{ codenum = Integer.parseInt(sipmsg); } catch(Throwable e) {}
                if(codenum >= 300) disconnectcode = sipmsg;
        }
}
if(disconnectcode.length() > 0) Log.v("AJVoIP", "The disconnect code is: "+disconnectcode);
```

SIP SDK Android integration

You can easily integrate your app with any other external service, Android service. These as nothing to do with services offered by a SIP library thus it has to be handled by yourself, however further integration with SIP can be easily handled by the AJVoIP capabilities as described in this documentation.
For example you might add tight integration with your or third-party apps or services by launching other activities, handling links, interacting with other apps, consuming HTTP API's or displaying web content from your website in your app.

Native phone integration

Native integrations means extending the native phone services with SIP capabilities by integrating your app with the Android OS.
For example you can easily implement native call integrations means handling calls made from the standard android dialer. With other words you can capture calls initiated by the native android dialer and handle it with your SIP application instead of the mobile service network.
We haven't added this functionalities inside the library since it requires user interface integration which is not suitable for a headless SDK, however we wrote a Android Native SIP Call Integration guide to help you with these tasks, covering the most important topics.
Beyond SIP call integration, this guide also covers some other topics such as managing the Android phone call history or acquiring the user phone number.

Contact management

This FAQ point is to help you with managing contacts if that is required for your needs.
Basic contact management API is included also in AJVoIP for your convenience, however usually it is recommended to handle this in your app after your specific requirements as it is usually highly coupled with user interface thus not much to do with a headless SIP library.

Below we will present a few simple code examples for your convenience:

a. Get contacts list
b. Get one contact details
c. Create a contact
d. Open native contact editor
e. Call history

For the below examples to work, you will need READ_CONTACTS and WRITE_CONTACTS permissions.

a) Let's start by getting the contact list with the below code

```
    // retrieving contact list (id and name)
    // if "searchName" parameter is not an empty String, then it will search for the contacts which names contain the specified String
    public void GetContactsList(String searchName)
    {
       Cursor cur = null;
       ContentResolver cr = MyApplication.getAppContext().getContentResolver();
           String selection =  ContactsContract.Contacts.HAS_PHONE_NUMBER + " = 1";

       if (searchName != null && searchName.length() > 0) { selection += " AND " + ContactsContract.Contacts.DISPLAY_NAME + " LIKE \"%" +
    searchName + "%\""; }
           String[] projection = new String[] { ContactsContract.Contacts._ID, ContactsContract.Contacts.DISPLAY_NAME };

       // query the Contacts database and return contacts in Ascending order by name
           cur = cr.query(
                 ContactsContract.Contacts.CONTENT_URI,
                 projection,
                 selection,
                 null,
             ContactsContract.Contacts.DISPLAY_NAME + " ASC");

           if (cur != null && cur.getCount() > 0)
       {
          // iterate through the cursor and get each contact ID and name
          while (cur.moveToNext())
          {
             if (cur.getString(cur.getColumnIndex(ContactsContract.Contacts._ID)) != null &&
    cur.getString(cur.getColumnIndex(ContactsContract.Contacts.DISPLAY_NAME)) != null)
             {
                // one single contact ID and name
                String contactId = cur.getString(cur.getColumnIndex(ContactsContract.Contacts._ID));
                String contactName = cur.getString(cur.getColumnIndex(ContactsContract.Contacts.DISPLAY_NAME));

             }
          }
       }
       if(cur != null){ cur.close(); cur = null; }
    }
```

b) Once we have the contact list, we can get any contact details by ID. In the following example we are querying just for the contact phone number, but any other field of interest can be queried the same way:

```
// get numbers and number types from a contact
public void GetContactPhoneNumbers(String contactId)
{
   ContentResolver cr = MyApplication.getAppContext().getContentResolver();
   Cursor cur = null;

   // get phone numbers and phone number types
   cur = cr.query(
         ContactsContract.CommonDataKinds.Phone.CONTENT_URI,
         null,
         ContactsContract.CommonDataKinds.Phone.CONTACT_ID +" = ?",
         new String[]{contactId}, null);

   int i = 0;
   if (cur != null && cur.getCount() > 0)
   {
      while (cur.moveToNext())
      {
         // a contact can have multiple phone numbers associated to it
         // for more details about phne number type, check the documentation here:
https://developer.android.com/reference/android/provider/ContactsContract.CommonDataKinds.Phone
         String phoneNumber = cur.getString(cur.getColumnIndex(ContactsContract.CommonDataKinds.Phone.NUMBER));
         String numberType = cur.getString(cur.getColumnIndex(ContactsContract.CommonDataKinds.Phone.TYPE));
         String numberId = cur.getString(cur.getColumnIndex(ContactsContract.CommonDataKinds.Phone._ID));
      }
   }

   if(cur != null){ cur.close(); cur = null; }
}
```

c) We can also create new contacts using the below code example:

```
    public void CreateContact(String name, String phone)
    {
       ContentResolver cr = getContentResolver();

       String accountType = null;
       String accountName = null;

       // create a raw contact and insert its name
```

```
    ContentValues values = new ContentValues();
    values.put(ContactsContract.RawContacts.ACCOUNT_TYPE, accountType);
    values.put(ContactsContract.RawContacts.ACCOUNT_NAME, accountName);

    Uri rawContactUri = cr.insert(ContactsContract.RawContacts.CONTENT_URI, values);
    long rawContactId = ContentUris.parseId(rawContactUri);

    values.clear();
    values.put(ContactsContract.Data.RAW_CONTACT_ID, rawContactId);
    values.put(ContactsContract.Data.MIMETYPE, ContactsContract.CommonDataKinds.StructuredName.CONTENT_ITEM_TYPE);
    values.put(ContactsContract.CommonDataKinds.StructuredName.DISPLAY_NAME, name);

    cr.insert(ContactsContract.Data.CONTENT_URI, values);

    // insert contact details (phone numbers and number types)
    Uri insertDetailsUri = Uri.withAppendedPath(rawContactUri, ContactsContract.Contacts.Data.CONTENT_DIRECTORY);
    values.clear();
    values.put(ContactsContract.Data.MIMETYPE, ContactsContract.CommonDataKinds.Phone.CONTENT_ITEM_TYPE);
    values.put(ContactsContract.CommonDataKinds.Phone.NUMBER, phone);
    values.put(PHONE_TYPE, ContactsContract.CommonDataKinds.Phone.TYPE_MOBILE);

    getContentResolver().insert(insertDetailsUri, values);
}
```

d) Open native contact editor

```
Intent intent = new Intent(Intent.ACTION_EDIT);
Uri uri = Uri.withAppendedPath(ContactsContract.Contacts.CONTENT_URI, String.valueOf(contactId));
intent.setData(uri);
act.startActivity(intent);
```

e) Call history

An example code to manage the native call history can be found in this guide.

SIP SDK for Delphi

The AJVoIP SIP SDK can be also used from Delphi or C++ Builder, for example to build a softphone in Embarcadero FireMonkey.
The AJVoIP library was created with API level suitable for Delphi, using support library for new features, thus it fulfills to Delphi requirements.
You will need to use the .jar file included in the AJVoIP download (not the .aar file) and create a native bridge to it using the Java2OP tool.

Follow this guide to add the AJVoIP jar to your Delphi or C++Builder project.

You can find more examples and guides over the internet.

Once the AJVoIP.jar file is added to your project, have a look at the quick start guide first, then check the details from this documentation.

The sample project was created for Android Studio, but the logic is the same. The relevant details are in the MainActivity.java file. Just rewrite it with Delphi or C++ Builder syntax or implement a similar logic in Delphi / C++ Builder.

Chrome OS

AJVoIP based apps can be converted also to ChromeOS and run on any compatible device such as Chromebooks.
For this a conversion is needed to ARC (App Runtime for Chrome).
You can convert the Android apk using any tool, for example as described here.
More details can be found here.

SIP library for Xamarin

AJVoIP can be used to add VoIP call capabilities to your Xamarin app:
Binding a Java Library
More details
MAUI

New user registration / sign-up

You might need such functionality if you wish to implement a SIP softphone, however there is no such service provided by the standard SIP protocol thus each service is handling this functionality in a proprietary way. Usually the softswitch will expose a simple HTTP API (XML or JSON GET/POST/PUT) for this which you can call from your application. Contact your service provider or SIP server documentation for the details about such an API.
Another possibility would be to integrate your "Sign-Up" webpage into your app as a WebView.

How to send SMS

Since this SIP library is intended to run mainly on SMS capable devices, usually there is no demand for built-in SMS functionality.
However if you wish integrate SMS functionality into your application, we can recommend the following options:

Use the platform native SMS capabilities

This can be achieved with a single line of code:

```
startActivity(new Intent(Intent.ACTION_VIEW, Uri.parse("sms:" + phonenumber)));
```

In case if you wish to preset the message text:

```
Intent intent = new Intent(Intent.ACTION_VIEW, Uri.parse("sms:" + phoneNumber));
intent.putExtra("sms_body", message);
startActivity(intent);
```

You can also send SMS without user interaction using the android.telephony.SmsManager capabilities.
In this case you will need to add the following permission: <uses-permission android:name="android.permission.SEND_SMS"/>

Use your SIP server SMS capabilities

It might be possible that you wish to offer SMS capabilities for your endusers, especially if you can get offer better pricing than the mobile service provider.
In this case the SMS are sent usually with one of the following methods:

- Using your server API: some SIP servers expose a HTTP API (PUT/POST/GET with clear text, JSON, XML or other playload) which can be easily integrated with SIP client applications. Just perform the API call as specified in your server documentation.
- Using standard SIP IM: some SIP server is capable to convert regular SIP text messages (RFC 3428) to SMS when the target number is a routable mobile phone number.
  In this case you just need to use the SendSMS API. Optionally you might set the haschat parameter to 2 to signal to the SIP server that your intention is to send SMS.

Handing incoming SMS

Special handling of incoming SMS messages doesn't have much meaning on an SMS capable devices since the OS is already capable to handle this task.
In case if your SIP server might send SMS over IP, then you have a few possibilities to handle it:

- Using the SIP protocol to handle SMS message (RFC 3428). In this case you don't need to do anything special as the messages are processed like regular IM extra. This should be the preferred delivery method of SMS messages in SIP networks.
- PUSH: you can use a server initiated push protocol for SMS delivery. The most effective is the usage of the cloud based push notifications
- Polling: this means requesting a server API from a timer to see if there is any new message arrived. Ineffective as it consumes extra CPU on both the server and client side thus it is not recommended especially in large networks, however due to its easy implementation this might be convenient in specific apps with a smaller user base if your SIP server (softswitch/PBX) doesn't support any other method
- Other protocols: you might handle incoming SMS with other proprietary or standard protocol like XMPP

How to implement IM

IM (Instant message/chat) is well supported by AJVoIP. Make sure that your server has support for SIP MESSAGE.

Basic IM:

The core chat functionality can be implemented very easily:

- To send a text message just use the SendChat function. Example: SendChat(-1, "john", "", "Hi!")
- Incoming text messages can be cached by the CHAT notifications. Example: "CHAT,1,john,Hi!"

Usually you will have to maintain a separate thread with each peer (displaying the conversations on a separate page by peer).

Optional features:

Once the simple chat send/receive is working, you might add some extra functionalities:

- Handle delivery success/failure:
  - You might parse the CHATREPORT notifications to notify the user if the message was delivered successfully or failed.

- Typing notifications:
  - You might use the SendChatIsComposing function to send typing notification to the peer.
  - Also parse the incoming CHATCOMPOSING messages (usually displaying something like “John is typing…”.
- Group chat:
  - If you wish to implement group chat, then you should send/accept the group name which is usually the members separated by | . Example: "emma | john | linda"
  - When sending, use the group parameter of the SendChat. Example: SendChat(-1, "emma", "emma | john | linda ", "Hi")
  - For receiving, check if the message begins with the "GROUP: xy:" substring where xy is the group.
  - Usually you have to display each group as a separate thread (separate window for each group name). If group name doesn't exists, then use the peername.
- Offline messaging:

  Offline messaging means queuing messages for later deliver when immediate delivery fails.

  Offline messaging is automatically handled by AJVoIP unless you set the offlinechat parameter to 0.

Beyond the IM SIP protocol offered by AJVoIP, chat is mostly about user interface. Implement a nice GUI with handy features such as threaded messaging, send picture (using the file send API), emoticons and any other features for your users.

How to implement PTT?

Push-to-talk or press-to-transmit (PTT) is an one-way communication method to mimic legacy half-duplex communication lines using a using a button to switch from voice reception mode to transmit mode. The feature is often used for door phones, walkie-talkies or industrial control functions.

PTT can be implemented by using the Hold function.
If somehow call hold is not well supported by your server or the remote peer, then you might use Mute instead.

Other related functions and parameters which you might use are the followings:
IsMuted, IsOnHold, HoldChange, automute, autohold, holdtypeonhold, muteonhold, defmute, sendrtponmuted

Caller ID display

For outgoing calls the Caller ID (CLI)/A number display is controlled by the server and the application at the peer side (be it a VoIP softphone or a pstn/mobile phone). Caller-ID means the remote username, extension number or real phone number as sent by your SIP server. The SIP server might or might not forward the remote real caller-id or might replace it to any arbitrary value (such as the CLI number associated to the user, the user extension or auth id).

You can use the following parameters to influence the caller id display at the remote end:

-username
-authusername/sipusername
-displayname

Some VoIP server will suppress the CLI if you are calling to pstn and the number is not a valid DID number or AJVoIP account doesn’t have a valid DID number assigned (You can buy DID numbers from various providers. This is up to your SIP server configuration and has nothing to do with AJVoIP).
The CLI is usually suppressed if you set the caller name to “Anonymous” (hide CLI).

For incoming calls the Java softphone will use the caller username, name or display name to display the Caller ID. (SIP From, Contact and Identity fields extracted from the incoming INVITE).

You can also use headers such as preferred-identity to control the Caller ID display.
The Caller-ID is received with the STATUS notifications or you can query it with the GetCallerID API. You can also use the GetIncomingDisplay API which can return some more details about the caller such as the display name.

Example (fields containing information about the caller name or number highlighted with bold):

INVITE sip:bob@biloxi.com SIP/2.0
Via: SIP/2.0/UDP pc33.atlanta.com;branch=z9hG4brn7wmmo4h
Max-Forwards: 35
To: Bob <sip:bob@biloxi.com>
From: Alice <sip:**alice**@atlanta.com>;tag=871822
Call-ID: a82c41f1b65
CSeq: 1 INVITE
Contact: <sip:**alice**@pc33.atlanta.com>
P-Asserted-Identity: "**Cullen Alice**" <sip:**alice**@atlanta.com>

P-Asserted-Identity: tel:**+14095361002**
Content-Type: application/sdp
Content-Length: 142

...sdp content...

You can learn more here and here.

How to implement IM

IM (Instant message/chat) is well supported by AJVoIP. Make sure that your server has support for SIP MESSAGE.

Basic IM:

The core chat functionality can be implemented very easily:

- To send a text message just use the SendChat function. Example: SendChat(-1, "john", "", "Hi!")
- Incoming text messages can be cached by the CHAT notifications. Example: “CHAT,1,john,Hi!”

Usually you will have to maintain a separate thread with each peer (displaying the conversations on a separate page by peer).

Optional features:

Once the simple chat send/receive is working, you might add some extra functionalities:

- Handle delivery success/failure:
  - You might parse the CHATREPORT notifications to notify the user if the message was delivered successfully or failed.
- Typing notifications:
  - You might use the SendChatIsComposing function to send typing notification to the peer.
  - Also parse the incoming CHATCOMPOSING messages (usually displaying something like “John is typing…”.
- Group chat:
  - If you wish to implement group chat, then you should send/accept the group name which is usually the members separated by | . Example: "emma | john | linda"
  - When sending, use the group parameter of the SendChat. Example: SendChat(-1, "emma", "emma | john | linda ", "Hi")
  - For receiving, check if the message begins with the "GROUP: xy:" substring where xy is the group.
  - Usually you have to display each group as a separate thread (separate window for each group name). If group name doesn't exists, then use the peername.
- Offline messaging:
  Offline messaging means queuing messages for later deliver when immediate delivery fails.
  Offline messaging is automatically handled by AJVoIP unless you set the offlinechat parameter to 0.

Beyond the IM SIP protocol offered by AJVoIP, chat is mostly about user interface. Implement a nice GUI with handy features such as threaded messaging, send picture (using the file send API), emoticons and any other features for your users.

Auto-answer

The android SIP library has full support for auto answer incoming call (automatically connecting incoming calls as they arrive without the need for user interaction).
Auto-answer can be used for various purposes such as intercom, free-hand mode, walkie-talkie, push to talk, call centers, door station or for any other after your business needs.
Auto-answer can be easily implemented multiple ways:

- **Accept**
  Just use the Accept function to pickup incoming call after your app logic as described here (you also have the possibility to inspect the caller details and decide which call to allow/auto-accept/reject)
- **Auto accept all calls**
  Set the enableautoaccept parameter to 3 to auto-answer all incoming call.
- **Server-side auto answer**
  Auto-answer can be also triggered/initiated by the SIP server.
  The SDK can auto-answer incoming calls if a special SIP header is sent with the incoming invite. To enable this feature, set the enableautoaccept parameter to 2 first.

Auto-answer can be triggered be any of the standard or non-standard auto-answer headers such as Call-Info, Alert-Info or Auto-Answer.
The Answer-After delay is also considered if set by your server (if the autoacceptdelay parameter is left with its -1 default value).
Example SIP headers that can trigger auto-answer:

```
Alert-Info: info=alert-autoanswer
Call-Info: Answer-After=0
Auto-Answer: normal
Answer-Mode: auto
```

For example if you are using Asterisk, add something like this into your extensions.conf:

```
exten => 100,1,SIPAddHeader(Call-Info:<sip:>\;answer-after=0)
exten => 100,n,Dial(SIP/1111)
```

It is also possible to use any special SIP header for auto answer (such as something proprietary for your server) with the autoacceptheader parameter.

- **Barge-in calls**
  You can specify enable barge-in hidden calls by using a special SIP header set with the bargeinheader parameter.
  This is similar with the above mentioned server-side initiated auto answer, but barge-in calls are hidden calls and they are often used in callcenters by supervisors to listen to AJVoIP calls with the purpose of monitoring agents activity.

It is also possible to accept the incoming call after some delay. This can be specified with the autoacceptdelay parameter. (By default it can be loaded from the Answer-After SIP header flag if received from your server with the incoming INVITE.)

P2P

How to use AJVoIP for peer-to-peer calls?

Registering to a SIP server has various advantages, but AJVoIP can be used also for peer to peer calls without using any SIP server for example to make direct calls between two AJVoIP instances or between AJVoIP and any other SIP endpoint (such as a third-party softphone on your local LAN).
In this case just set the register parameter to 0 and set the serveraddress parameter to the peer SIP endpoint address (IP:port) or use full SIP URI to make calls such as Call(-1, '1111@192.168.1.8:5678').

To make AJVoIP reachable for incoming calls, you should set a fix value for the signalingport.
For example if your phone IP address is 192.168.1.8 and you have set the signalingport to 5678, then remote peers can reach your AJVoIP instance by calling to sip:1111@192.168.1.8:5678.

Set also the startsipstack parameter to 2 so it will auto start.
You might also set the bindip (if your server has multiple network interfaces) and the localip parameters (for example if AJVoIP is running on a private address and you wish to specify the external address that is communicated with the clients). There is also a favlocalip and a bindtolocalip parameter or similar purposes.

In case if you wish to make direct (not via a SIP server) calls to other peers then you do everything the same way as you would work with a server, with the following changes:

- Set the serveraddress parameter to your peer address. Be aware that SIP apps might not use the default SIP port (5060) so you should check the SIP listener port of the peer first and use the same in the serveraddress parameter (which is usually specified as IP:port in this case). Alternatively pass a full SIP URI for the Call function (instead of the target number/user only)
- Set the setserverfromtarget parameter to 2 if you preconfigured a serveraddress but you need direct calls elsewhere
- Set the register parameter to 0 to disable registrations, because register is usually not required in this use-case

Example parameters to make direct calls from machine A to machine B (with IP 192.168.1.12):

- Machine B (UAS/server endpoint where you will accept the call):
  - startsipstack: 2
  - register: 0
  - signalingport: 5070
  - username: 5555
- Machine A (UAC/client endpoint from where you make the call):
  - username: 4444
  - serveraddress: 192.168.1.12:5070
  - startsipstack: 2
  - register: 0

- Then make a call like: Call(-1, "5555"); or Call(-1, "5555@192.168.1.12:5070");

Note:

- *It is also possible to configure the endpoints differently with the SetLineParameter function.*
- *This FAQ point has little to do with P2P media (which can happen also for "normal" calls routed via a SIP server, depending on the circumstances: AJVoIP parameters, SIP server configuration, networking/NAT).*
- *When calling to a direct URI, the username/user part of the URI usually doesn't matter, so you might even set it to anonymous (except if you have set the rejectcallto parameter)*

How to USSD

IMS USSD (Unstructured Supplementary Service Data) messages are described in the 3GPP TS 24.390 standard.
This service provides the support for UE or network initiated MMI strings including single requests and dialogs.
It is commonly used to send quick feature codes to/from mobile networks used for various purposes such as balance request, callback, MMI supplementary services and other operations.

First of all, make sure to enable USSD by setting the ims3gpp_ussd parameter to 1 or the ims3gpp parameter to 2 as described here.
Use the SendUSSD function to send USSD strings.
Received USSD strings can be cached by the USSD notifications.

A simple USSD message exchange looks like this:

1. API function call: SendUSSD(-1, "INVITE", "*135#"); //initiate USSD dialog (for example ask for available credit)
2. notification: USSD,1,1,sent //message successfully sent
3. notification: USSD,1,2,Enter PIN code //incoming USSD message (for example network asking for extra info)
4. API function call: SendUSSD(1, "INFO", "1234"); //send USSD answer with INFO
5. notification: USSD,1,1,sent //message successfully sent
6. notification: USSD,1,2,Your credit is 5 USD //incoming USSD message (for example final answer received in BYE)
7. API function call: Hangup(1); //just for sure in case if somehow the call was not disconnected by the server

How to DTMF

DTMF means Dual-tone multi-frequency signaling and it is commonly used to send touch tones to the server or the other peer. On phone user interface this is usually handled by handling key press or presenting a phone interface to the user who can press the digits to be sent.

Using the SIP protocol, there are multiple ways to send DTMF digits which can be set by the dtmfmode parameter.

If you are using the built-in user interface then you can send DTMF digits by pressing the number buttons during a call and if a dtmf digit is received, then you will see it displayed on the user interface notification area.

DTMF digits can be sent using the Dtmf API.
Feedback about the delivery can be obtained by watching for the INFO notifications (you will receive INFO,OK when the message was successfully sent or INFO,ERROR if failed).

On incoming DTMF you will receive a DTMF notification.

Custom INFO messages

You can send and receive custom SIP INFO messages in the SIP signaling as described in RFC 2976.
Note: If you are looking to send or receive DTMF digits, then you should look at the above FAQ point instead.

To send a custom INFO message, use the Info function.

Feedback about the delivery (if necessary) can be obtained by watching for the INFO notifications (you will receive INFO,OK when the message was successfully sent or INFO,ERROR if failed).

INFO notifications are also triggered for incoming SIP INFO messages (INFO,REC).

Transfer API usage

Call transfer is about transferring a connected call with SIP REFER.
If you are looking to forward a (not yet connected) incoming call then you should use the Forward function instead.

The transfer mode can be set with the transfertype parameter and the calls can be transferred with the Transfer function.
The call transfer is handled with the SIP REFER method as described in RFC 3515, RFC 5589 and RFC 6665.

With **unattended transfer** using transfertype 1 the transfer will be executed immediately once you call the Transfer function (blind transfer; only a SIP REFER is sent).
With **unattended transfer** using transfertype 6 and holdontransfer 1 or 2 the call might be put on hold before transfer and reloaded on transfer fail.
With **attended transfer** (transfertype 5) you can use the following call-flow, supposing that you are working at A side and wish to transfer B to C:

With unattended transfer the transfer will be executed immediately once you call the Transfer function.
With attended transfer (transfertype 5) you can use the following call-flow (supposing that you are working at A side and wish to transfer B to C):

1. A call B (outgoing) or B call to A (incoming)
   A speaking with B
2. Call Transfer (-1,C)
   The call between A and B will be put on hold by A
   A will call to C and connect (consultation call; if call fails, then the call between A and C will be un-hold automatically)
   A speaking with C
3. The actual call transfer will be initiated when A disconnect the call (Hangup)
   REFER message will be sent to B (which tells to B to call C. usually by automatically replacing the A-B call with A-C)
4. After transfer events:
   -If transfer fails (B can't call C) the call between A and B will be will be un-hold automatically (if server sends proper notifications)
   -If the transfer succeeds (B called C) the call between A and B will be will be disconnected automatically (if server sends proper notifications)
   -If the server doesn't support NOTIFY, then the call can be un-hold or disconnected from the API (Hold(-2,false), Hangup(-1))
5. B speaking with C at this point if the transfer was successful

There are a few optional parameters related to call transfer which you might set after your needs:
transfertype, transfwithreplace, allowreplace, discontransfer, disconincomingrefer, transferdelay, checksubscriptionstate, subscribefortransfer, holdontransfer, calltransferalways

Line parameter

For simple use-case the line parameter for the API functions almost always can be just set to **-1**.

More details:
The line parameter is part of most of the API functions and means the channel number.
With this parameter you can specify the session for which a function call to be applied (such as a specific call if there are multiple simultaneous calls).
The following values are defined:

- -2: all channels
- -1: the current channel set previously by SetLine API or by other functions. Usually this will mean the first channel (1)
- 0 : undefined (this should not be received/sent for endpoints in call, but might be used for other endpoints such as register endpoints)
- 1: first channel
- 2 : second channel
- etc (you can control the max number of the channels with the maxlines parameter which is set to 4 by default)

Most commonly, you will have to always pass -1 as the channel number. You will have to use other values only if you will present a GUI were the user can select different lines or if you have some special requirement to handle the lines explicitly.
Otherwise, AJVoIP can do this automatically allocating new channels when needed.

The line parameter is also part for most of the notifications to inform you which session triggered the notification.

See the "Multiple lines" FAQ point below if you wish to handle the lines explicitly.

Multiple lines

Multi-line means the capability to handle more than one call at the same time (multiple channels).
Multi-lines are handled automatically by default and you can skip this description if you don't wish to explicitly manipulate the different lines (for example allowing the enduser to hold/forward/transfer or do any other action per line separately).

This can be managed by the line parameter of the API functions (most functions has a line parameter) and by checking the line parameter of the STATUS and other notifications (most notifications has a line number).

By default you don't need to do anything to have multi-line functionality as this is managed automatically with each new call on the first "free" line. This means that you just need to pass -1 as the line number with any function call and look only the notifications with the line number set to -1 (the global phone state).

If you wish to manage the lines explicitly, then you should pass the desired line numbers with the API calls and also look for the notifications received from the individual lines (such as the line parameter of the STATUS notification). This is necessary for example if you wish to create a user interface where the users can select/change to a specific line (line selector or line buttons).

Multi-line vs multi-account

Multiple accounts refers to the feature when you might have more than one SIP account (on the same or separate SIP servers).
Multi-lines is described here and refers to the feature when you might need to handle multiple phone channels usually for multiple simultaneous calls or call transfer.
You can also have multi-accounts and multi-lines at the same time (multiple simultaneous calls via multiple accounts).

Multi-line vs Conference

When we refer to "multi-line" we mean the capability to have multiple calls in progress at the same time. This doesn't necessarily means conference calls. You can initiate multiple calls by just using the *Call* API multiple times (to initiate calls to more than one user/phone), so you can talk with remote peers independently (all pears will hear you unless you use hold on some lines, but the peers will not hear each-others).
To turn multiple calls into a conference, you need to use the *Conf* API (or use the conference button from the softphone.html). When you have multiple peers in a conference, all peers can hear each-other.

Usage

- Just set the line parameter to -1 for all/most function calls and parse only notifications with line -1 if you don't wish to deal with multiple lines explicitly. In this case AJVoIP will handle all the details and will guess which lines to use.
- However, if you have some special requirements to handle the lines/calls in a different way, then try to be strict with the line numbers: try to always pass the correct line parameter for the API calls and process the notifications from different lines according to your app requirements.

Enable multi-line

There is nothing to be done to enable multi-line functionality as it is enabled and handled by default.
Optionally you might set the following parameters if your use-case requires multiple simultaneous calls:

- aec: 0
- aec2: 0
- agc: 0
- focusaudio: 1

Disable multi-line

You can disable multi-line functionality with the following settings:

- set the "multilinegui" parameter to 0
- set the "rejectonbusy" setting to "true"

Other related parameters are the "*automute*" and "*autohold*" settings.

Channel settings

Individual lines are initialized with the global parameters and their behavior might change depending on the circumstances and usage.
Some parameters can be also set line by line using the SetLineParameter function (avoid using this when possible and use the other API's instead to influence the behavior of specific individual lines).

API

Most API functions has a line parameter which you can set to specify the line number. Some API's such as the GetLineStatus have been implemented to help you using multiple lines.
Also there are two specific API to set/get the current active line:

- SetLine(line); // Will set the current line. Just set it before other API calls and the next API calls will be applied for the selected line
- GetLine(); //Will return the current active line number. This should be the line which you have set previously except after incoming and outgoing calls (will automatically switch the active line to a new free line for these if the current active line is already occupied by a call)

The active line is also switched automatically on new outgoing or incoming calls (to the line where the new call is handled).

Notifications

Check the line parameter (getLine()) for the STATUS and other notifications and update your line state machine accordingly and/or change your user interface accordingly.

Channels

The following line numbers are defined:

- -3: in some functions you can refer to the next session with the line parameter set to -3
- -2: all (some API calls can be applied to all lines. For example calling hangup(-2) will disconnect all current calls)
- -1: current line (means the currently selected line or otherwise the "best" line to be used for the respective API)
- 0: undefined (this should not be received/sent for endpoints in call, but might be used for other endpoints such as register endpoints)
- 1: first channel
- 2: second channel
- …
- N: channel number X

Note: If you use the SetLine with -2 and -1, it might be remembered only for a short time; after that the GetLine will report the real active line or "best" line.

API usage example:

```
Call(1,'1111');  //make a call on the first line
Call (2, '2222');  //make a second call on the second line

//setup conference call between all lines
Conf(); //interconnect current lines

//put first call on hold
Hold (1,true);

//disconnect the second call
Hangup(2, true);
```

Call recording

AJVoIP is capable to record the conversations and it can store the result in a stereo voice file (wav, mp3, gsm, ogg) with the left side containing the caller voice and the right side containing the called party voice. The recorded voice files can be saved to the user local device or uploaded to your WEB or FTP server.
You can use the parameters and/or the API to active this feature.

If all calls have to be always recorded, then just set the voice recording parameters after your needs:
voicerecording is the most important parameter to be set.
If you set the voicerecording parameter to 2 or 3 then either the voicerecftp_addr or the http_addr parameter should be also set.
Other related parameters which you might change: voicerecfilename, voicerecformat, syncvoicerec, uploadretry.

If you wish to turn on/off the recording at runtime (for example to record only certain calls or only part of calls) then you can use the **VoiceRecord** function.
In this case you might set the voicerecording parameter to 0 to disable recording by default for the rest of the calls.

You can also receive notifications about the voice recording progress/success/failure: VREC.

How to get the audio stream?

The VoIP SDK is capable to stream the received/sent media to your app (your local application or other application which will process the audio/video data).

You can receive the media stream in two ways:

- **GetMedia**()
  Receive the media packets from AJVoIP by calling the GetMedia function from your app (from a thread).
  Set the sendmedia_mode parameter to 2.
  This is the new recommended way for local audio streaming.
  Code example here.
- **UDP packets**
  Receive the media packets from AJVoIP over UDP.
  Set the sendmedia_mode parameter to 1.
  Launch an UDP server socket to listen on any port and set the same port number as the sendmediain_to and/or sendmediaout_to parameter for the SDK.
  Then you will receive the media streams (the audio and/or video packets) from the calls as UDP packets and from there you can process them as you wish.
  The UDP packets can be sent also elsewhere as specified by the sendmedia_to_ip parameter.
  This is the old way for local audio streaming (will remain fully supported).

"Stream" and "streaming" in this context means sending the RTP or the raw audio packets to a non-VoIP application (such as your app).
It doesn't meant the usual VoIP RTP streams (streaming RTP to remote VoIP peers or from/to SIP server).

Streaming is useful if you wish to make some processing on the media streams such as custom audio playback, video recording, speech to text, AI speech processing or forward to other API/SDK/cloud service for any further processing. For example real-time translation via the Azure or Google Cloud API or ChatGPT integration.

Technically the only difference from normal calls is that in this case the media (the incoming and/or outgoing RTP media streams) will be forwarded to your UDP listener or received by GetMedia function calls, instead to be sent to the speaker/audio output device (it can be also streamed to both your listener and the audio device in the same time or you can set the useaudiodeviceplayback to false or mute the call to disable playback on the sound device).

Parameters:

sendmedia_mode

Specify how to get the media streams
-1: auto (will be set to 1 if you set the sendmediain_to or sendmediaout_to parameter; will be set to 2 if you call the GetMedia function)
0: disable
1: UDP (received on UDP socket)

2: API (received by the GetMedia function)
3: both (this is usually unnecessary/not recommended)

sendmedia_atype

Specify the audio media stream format sent by AJVoIP to your app:
0: raw wave format (linear PCM) for narrowband (8 kHz 16 bit mono PCM files at 128 kbits - 15 kb/sec) or 16 kHz for wideband.
1: for RTP format (RTP header + payload with the actual media codec)
2: convert sample rate to raw PCM 8kHz (useful if original stream is in 16kHz format such as Opus or Speex wideband, but you need 8kHz PCM)
3: convert sample rate to raw PCM 16kHz (useful if original stream is in narrowband 8kHz format but you need 16kHz PCM)
4: RTP data only, without RTP header (raw codec format)
5: convert to L16 mono, 16khz, Big Endian (16-bit signed linear PCM)
6: convert to L16 mono, 16khz, Little Endian (16-bit signed linear PCM)
Default is 0.
Note:
This parameter is applied only for audio data. Video is always sent as unmodified RTP packets.
It is recommended to set this to 0 or 1 and force a codec to match your needs (use OPUS, Speex or G.722.1 if you need 16kHz PCM wideband or other codec if you need 8kHz PCM narrowband), however if you need raw audio in other sample rate then you might set it to 2 or 3 for sample rate conversion.
With Google STT we recommend to set the sendmedia_atype to 3 and set the codec encoding: LINEAR16 and sampleRateHertz: 16000 for the Google speech API.

sendmedia_mtype

Specify media type:
-1: default (usually will default to 4)
0: undefined/reserved
1: audio only
2: video only
3: both with media type header byte (the first byte will be set to 1 for audio data or to 2 for video data)
4: both, without media type header byte

sendmedia_dir

Specify which streams to forward.
-1: auto (will be set to 1,2 or 3 depending on the sendmediain_to or sendmediaout_to parameter)
0: undefined/reserved
1: incoming (RTP received to AJVoIP from the remote peer)
2: outgoing (RTP sent by AJVoIP to remote peer)
3: both with direction header byte (the first byte will be set to 1 for incoming RTP or to 2 for outgoing)
4: both, without media type header byte

sendmedia_line

Specify if packets should have a line number header. Useful in case of you wish to handle multi lines in the same stream (sent to same UDP port).
0: no header (default)
1: add line header string. In this case the packets will begin with the line number, followed by comma, followed by the media payload (binary data).
Example: 2,xxxxx
(You must extract the line number from each packet: get the string until the first comma and convert it to int. The bytes after the comma will be the audio data)
2: add line and SIP Call-ID string header. Example: 3,abc,xxxxx
3: add line header byte. In this case the first byte in the packet will be the line number (0-127). This should be used only if you have less than ~100 simultaneous calls.
4: add line header byte and VAD status byte. The VAD status (second byte) will be 0 if unknown, 1 if silence, 2 if speaking.

sendmedia_marks

Specify if you wish to receive BOF/EOF packets (udp packets containing 3 bytes at the beginning and at the end of streams).
Default is 0. Set to 1 if you wish to have these packets.

sendmedia_conf

Set to 1 if you wish to receive the audio streams in conference calls separately (separate stream for each endpoints).
Default is 0 which means that you will receive a single stream (with conference mixed audio packets)

sendmediain_to

Specify your UDP port where you wish to receive remote media (received from other peer for playback on local speaker)
Default is 0 which means no streaming.
Not required if you use the GetMedia API (instead of UDP socket)

sendmediaout_to

Specify your UDP port where you wish to receive local media (recorded from microphone, which is sent to the other end)
Default is 0 which means no streaming.
Not required if you use the GetMedia API (instead of UDP socket)

sendmedia_to_ip

Specify the IP address where the media have to be forwarded (to be used with the above ports)
Default is 127.0.0.1.
Not required if you use the GetMedia API (instead of UDP socket)

For example if you are interested only in incoming audio in raw PCM 8kHz format, then set the following parameters:

- sendmedia_mtype: 1 //audio only
- sendmedia_dir: 1 //incoming only
- sendmedia_atype: 2 //narrowband

Code example:

- A simple working example using GetMedia API can be downloaded from here.

Note:

- The *sendmedia_mtype*, *sendmedia_dir* and *sendmedia_line* settings might insert additional bytes at the beginning of the media buffer. You should process or ignore them (in the above order).
- In case if you just wish to store the audio and don't need real-time processing, then you should use voicerecording / VoiceRecord instead of this streaming as described here.
- If you just need to stream audio to a remote SIP endpoint, then use the PlaySound or the StreamSoundBuff function instead.
- You might force G.711 codec to simplify the audio formats conversion (disable all codec's except PCMU and PCMA).

How to send a media stream?

The VoIP SDK is capable to playback media (audio and/or video) streams to the remote peer. You just need to use the PlaySound function to play a sound file to the remote or the StreamSoundBuff/StreamSoundStream function if you wish to play from buffer/stream in real time.
Technically the only difference from normal calls is that in this case you will stream audio from a file or from a custom buffer/stream instead of from the microphone/audio input device.

Here are the detailed systematic instructions for this use-case:

1. **Configure AJVoIP** with the required parameters/credentials by using the SetParameter(s) function and passing the serveraddress, username, password and any other parameters after your needs.
   You might set the useaudiodevicerecord parameter to false if you don't need recording from the local audio device.
2. AJVoIP can **register** automatically on startup if you passed the credentials, or you can set the register parameter to 0 if no registration is required or you might use the Register function to register explicitly.
3. To **make a call**, just use the Call function.
   Until this we just described the basic usage and for all these you can also find a simple **example code** here or inspect the sample downloadable from here.
4. To **detect call failure** (such as remote peer busy or doesn't respond), watch for the STATUS notifications for call disconnects or for the CDR notifications.
   After each call you will receive a CDR notification (with the duration parameter set to 0 if the call failed) and from there you can redial if you wish by calling the Call function again.
5. To **stream sound to the peer**, wait for call connect (STATUS notification with the statustext set to "In Call" on line -1 or "CallConnect" on the specific line) and then call the PlaySound function to stream from file.
   Instead of PlaySound, you might use the StreamSoundBuff or the StreamSoundStream function if you want to play from buffer or stream instead.
6. If you wish to **disconnect the call** once the streaming is ready, then just call the Hangup function when you receive the PLAYREADY notification.

In case **if you don't wish to record or playback to audio device** at all (only stream from file or buffer), then set the useaudiodeviceplayback and useaudiodevicerecord parameters to false.

Work with audio streams

If you have set audio streaming as it was discussed above, then you have the following possibilities to handle it:

1. Third party software:

Use some third-party software which is capable to playback raw PCM packets.

2. RTP stream:

If you have a software which is capable to process RTP packets then just set the sendmedia_type to 1 so the AJVoIP will emit RTP packets which can be processed as is by such software.

3. PCM stream:

Use the default PCM stream if you don't have any ready to use software and you have to write it yourself.
The packet emitted by the SIP media stack can be processed as-is by any audio player module.

4. Wave files:

If you don't need real-time audio, then just use the voicerecording and related parameters to obtain voice files in wave or other formats at the end of each call.

5. Barge-In:

If you just wish to listen into conversation, then you can use the AJVoIP itself for the job. Its barge-in feature allows you to bare into any call and create a hidden conference endpoint, so you can hear the conversation. This is often used in callcenters by supervisors. Contact our support if you need this module.

6. SIP media streaming:

Use the PlaySound function if you need to stream an audio file to a remote SIP endpoint.

Disable local audio

If you wish to use the library only for call recording and/or streaming and you don't need recording/playback from/to the local audio device, then you might set the following parameters:

- useaudiodevicerecord: false
- useaudiodeviceplayback: false
- audiomanagermode: -8
- focusaudio: 1
- checkvolumesettings: 0
- vibrate: 0
- playring: 0
- ringincall: 0
- beeponconnect: 0
- agc: 0
- aec: 0
- denoise: 0
- syncvoicerec: 0
- mediatimeout: 0

RTP header extension

With RTP header extension you can send/receive extra 32 bit words with the RTP headers as described in Section 5.3.1 of RFC 3550.
This is a rarely used RTP feature. Make sure that your VoIP server and/or the peers supports this before to try sending any extra RTP header.

Sending:

If you wish to always send the same data, then you can use the rtpextraheader and the rtpextraheader_profile parameters to set it globally.
If you wish to modify it at runtime or to send different data per call, then use the RTPHeaderExtension function.

Receiving:

Set the rtpextraheadernotify parameter to 1 if you wish to receive RTPE notifications about the received RTP extra header changes.
Note: if the ed137 parameter is set, then you will receive RTPT notifications (instead of RTPE). See the ED-137 documentation for more details.

Notification strings

Notifications can be also received as strings by the PollNotificationStrings or GetNotificationsStrings API calls.

Usually you will have to parse the received strings from your code. The parameters are separated by comma ',', First you have to check the first parameter (until the first comma) to determine the event type. Then you have to check for the other parameters according to the specification below.
Please note that you can get more than one event at once, separated by newline (or ,NEOL \r\n) so you should check for new lines first and then parse all events.
Notifications might be prefixed with the "WPNOTIFICATION," string (you should handle and remove this before parsing the rest of the line).
It is also possible to convert the strings to SIPNotification objects by using the ParseNotification() function.

Handling the notifications as strings are deprecated now (since the SIPNotification events have been introduced), but we keep it maintained as it might be useful in some special circumstances.

Full documentation and example:

See the following example and documentation for the details about handling the notifications as strings:

- Test/Example: https://www.mizu-voip.com/Portals/0/Files/AJVoIPTest_With_Notification_Strings.zip
- Documentation:
  - PDF: https://www.mizu-voip.com/Portals/0/Files/AJVoIP_With_Notification_Strings.pdf
  - HTML: https://www.mizu-voip.com/Portals/0/Files/documentation/ajvoip_with_notification_strings/index.html
  - WinHelp: https://www.mizu-voip.com/Portals/0/Files/AJVoIP_With_Notification_Strings.chm
  - JavaDoc: https://www.mizu-voip.com/Portals/0/Files/ajvoip_with_notification_strings_javadoc/index.html

In the above example and documentation the notifications are always handled as plain strings, not using the SIPNotificationListener with SIPNotification objects.

Here are some more relevant notification strings:

STATUS statustext strings:

The following **statustext** values are defined **for general status (with line set to -1):**

- Starting...
- Idle.
- Ready
- Connecting…
- Securing…
- Register…

- Registering… (or “Register...”)
- Register Failed
- No network
- Server address unreachable
- Not Registered
- Registered (or “Registered.”)
- Unregistered
- Accept
- Starting Call
- Call
- Call Initiated
- Calling…
- Ringing…
- Incoming…
- In Call (xxx sec)
- Hangup
- Call Finished
- Chat (or Messaging)

Note: general status means the “best” status among all lines. For example if one line is speaking, then the general status will be “In Call”.

The following **statustext** values are defined **for individual lines** (line set to a positive value representing the call channel number starting with 1):

- Unknown (you should not receive this)
- Init (voip library started)
- Ready (sip stack started)
- Outband (notify/options/etc. you should skip this)
- **Register** (from register endpoints) (or “Register…” or “Registering...”)
- Unregister
- Subscribe (presence)
- Chat (IM)
- **CallSetup** (one-time event: call begin)
- Setup (call init)
- InProgress (call init)
- Routed (call init)
- Ringing (SIP 180 received or similar)
- **CallConnect** (one-time event: call was just connected)
- InCall (call is connected)
- Muted (connected call in muted status)
- Hold (connected call in audio hold status)
- Speaking (call is connected)
- Midcall (might be received for transfer, conference, etc. you should treat it like the Speaking status)
- **CallDisconnect** (one-time event: call was just disconnected)
- Finishing (call is about to be finished. Disconnect message sent: BYE, CANCEL or 400-600 code)
- Finished (call is finished. ACK or 200 OK was received or timeout)
- Deletable (endpoint is about to be destroyed. You should skip this)
- Error (you should not receive this)

You will usually have to display the call status for the user, and when a call arrives you might have to display an accept/reject button.
For simplified call management, you might check only the global state or you can just check for the one-time events (CallSetup, CallConnect, CallDisconnect).
These are sent only for the actual line (1,2,etc) and not as global state (-1).
Not all call statuses might be sent (for example the Finishing state can be often skipped and you will receive only the Finished state after a call disconnect).

STATUS notifications related to global/main account registration success or failure:

- Proceeding notification strings:
  - Register…
  - Registering… (or “Register...”)
  - Register Failed
- Success notification strings:
  - Registered
  - Registered.
- Failure notification strings:
  - Connection lost
  - No network
  - Server address unreachable
  - No response from server
  - Server lost
  - Authentication failed
  - Rejected by server
  - Register rejected

- Register expired
- Register failed

Parse notifications code example:

Instead of parsing the strings yourself, you can convert them to SIPNotification objects with ParseNotification() or even better: receive the notifications directly as SIPNotification objects by using the SetNotificationListener().
Notifications handling is all about a bit of string parsing so maybe better if you just implement it from scratch so you will be more familiar with your own logic then the string handling practices in this example code.

You will have to parse the received strings from your code by first splitting them by line, since more than one line can be received at once (separated by CRLF or with ",NEOL \r\n ", each line represents a new notification). The most important notification is the STATUS message which can be used to learn the SIP stack state (global state or per line state).
Notifications might be prefixed with the "WPNOTIFICATION," string (you should remove this before parsing the rest of the line).
The parameters are separated by comma ',' . First you have to check the first parameter (until the first comma) to determine the event type. Then you have to check for the other parameters according to the specification below.

Below is a short code snippet demonstrating basic notification parsing.
In this example we assume that we receive the notifications in a function called *ProcessNotifications* (called from PollNotificationStrings or GetNotificationStrings).

Below is a short code snippet demonstrating basic notification parsing.
Notifications handling is all about a bit of string parsing so maybe better if you just implement it from scratch so you will be more familiar with your own logic then the string handling practices in this example code.
In this example we assume that we receive the notifications in a function called ProcessNotifications.

```
public void ProcessNotifications(String receivednot)
{
    if (receivednot == null || receivednot.length() < 1) return;

    // we can receive multiple notifications at once, so we split them by CRLF or with ",NEOL \r\n" and we end up with a
String array of notifications
    String[] notarray = receivednot.split(",NEOL \n");
    for (int i = 0; i < notarray.length; i++)
    {
        String notifywordcontent = notarray[i];
        if (notifywordcontent == null || notifywordcontent.length() < 1) continue;
        notifywordcontent = notifywordcontent.trim();
        notifywordcontent = notifywordcontent.replace("WPNOTIFICATION,", "");

        // now we have a single notification in the "notifywordcontent" String variable
        Log.v("AJVOIP", "Received Notification: " + notifywordcontent);

        int pos = 0;
        String notifyword1 = ""; // will hold the notification type
        String notifyword2 = ""; // will hold the second most important String in the STATUS notifications, which is the
third parameter, right after the "line' parameter

        // First we are checking the first parameter (until the first comma) to determine the event type.
        // Then we will check for the other parameters.
        pos = notifywordcontent.indexOf(",");
        if(pos > 0)
        {
            notifyword1 = notifywordcontent.substring(0, pos).trim();
            notifywordcontent = notifywordcontent.substring(pos+1, notifywordcontent.length()).trim();
        }
        else
        {
            notifyword1 = "EVENT";
        }

        // Notification type, "notifyword1" can have many values, but the most important ones are the STATUS types.

        // After each call, you will receive a CDR (call detail record). We can parse this to get valuable information
about the latest call.
        // CDR,line, peername,caller, called,peeraddress,connecttime,duration,discparty,reasontext
        // Example: CDR,1, 112233, 445566, 112233, voip.mizu-voip.com, 5884, 1429, 2, bye received
        if (notifyword1.equals("CDR"))
        {
            String[] cdrParams = notifywordcontent.split(",");
            String line = cdrParams[0];
            String peername = cdrParams[1];
            String caller = cdrParams[2];
            String called = cdrParams[3];
```

```
            String peeraddress = cdrParams[4];
            String connecttime = cdrParams[5];
            String duration = cdrParams[6];
            String discparty = cdrParams[7];
            String reasontext = cdrParams[8];
        }
        // lets parse a few STATUS notifications
        else if(notifyword1.equals("STATUS"))
        {
            //ignore line number. we are not handling it for now
            pos = notifywordcontent.indexOf(",");
            if(pos > 0) notifywordcontent = notifywordcontent.substring(pos+1, notifywordcontent.length()).trim();

            pos = notifywordcontent.indexOf(",");
            if(pos > 0)
            {
                  notifyword2 = notifywordcontent.substring(0, pos).trim();
                  notifywordcontent = notifywordcontent.substring(pos+1, notifywordcontent.length()).trim();
            }
            else
            {
                  notifyword2 = notifywordcontent;
            }

            if (notifyword2.equals("Registered."))
            {
                // means the SDK is successfully registered to the specified VoIP server
            }
            else if (notifyword2.equals("CallSetup"))
            {
                // a call is in the setup stage
            }
             else if (notifyword2.equals("Ringing"))
            {
                // check the other parameters to see if it an incoming call and display an alert for the user
            }
            else if (notifyword2.equals("CallConnect"))
            {
                // call was just connected
            }
            else if (notifyword2.equals("CallDisconnect"))
            {
                // call was just disconnected
            }
            else if (notifyword1.equals("CHAT"))
            {
                // we received an incoming chat message (parse the other parameters to get the sender name and the text to
be displayed)            }
        }
        else if(notifyword1.equals("ERROR"))
        {
            // we received an error notification; at least log it somewhere
            Log.e("AJVOIP", "ERROR," + notifywordcontent);
        }
        else if(notifyword1.equals("WARNING"))
        {
            // we received a warning notification; at least log it somewhere
            Log.w("AJVOIP", "WARNING," + notifywordcontent);
        }
          else if(notifyword1.equals("EVENT"))
        {
            // display important event for the user
            Log.v(notifywordcontent);
        }
    }
}
```

SIP API via Broadcast Intents

You can access the API and call API methods also via android broadcast intents so you can control the SIP stack also from a different/third-party app.
This feature is rarely used (since you can access the API directly from your code), but it might be useful in some circumstances if you wish to make SIP calls, accept calls, mute/hold, send DTMF and perform many other SIP actions with android broadcast intents.
With this feature you can send API requests via broadcast intent by passing the API name (with the “function” key) and its parameters (with the keys set to the name of the parameter) by Intent extras. Most of the API functions are available also via broadcast intents.

In your app in which you are using AJVoIP, add the following lines into your manifest file (AndroidManifest.xml) if you need this functionality (to export the BroadcastReceiver):

```
<manifest ...>
        <application ...>
```

```
        ...
        <receiver android:name="com.mizuvoip.jvoip.BroadcastAPI"
                android:exported="true">
                <intent-filter>
                        <action android:name="com.mizuvoip.jvoip.VOIP_API" />
                </intent-filter>
        </receiver>
        ...
    </application>
</manifest>
```

To send API request to AJVoIP, you will need to create an Intent with action set to **com.mizuvoip.jvoip.VOIP_API** and pass the following extras:

- **function**: the name of the API function you wish to call (for example "Call")
- **param1**: value1 (for each of the function parameters, where param is the parameter name as written in this documentation and value is the parameter value)
- **param2**: value2 (for example line: -1. See the API chapter in the AJVoIP documentation for the exact name of the function methods parameters)
- …
- **paramN**: valueN
- **requestid**: optional. It might be useful if you wish to receive the function result and in this case you might pass an auto-increment sequence number which you will receive back in the APIRESULT notification.

Note: The function parameter names can be passed also as "param1", "param2" … "paramN" keys (in the expected order), otherwise use the exact parameter name as listed in this documentation (see the API functions chapter*).*

If you wish the get feedbacks about the function call results, set the apiresultnotifications parameter to 1 and make sure that the externapi_event parameter contains "APIRESULT" or it is set to "ALL". Then you will receive APIRESULT broadcast intent notifications for every API function call you made. This is often unnecessary because most functions are executed asynchronously so you can learn about their result from the other notifications.

Example code to make a SIP call with broadcast intent from another app:

```
Intent intent = new Intent("com.mizuvoip.jvoip.VOIP_API"); //create intent
intent.setComponent(new ComponentName("com.mizuvoip.jvoip", "com.mizuvoip.jvoip.BroadcastAPI")); //make it explicit (optional, but recommended to avoid any Android OS restrictions)
intent.putExtra("function", "Call"); //API function name
intent.putExtra("line", -1); //the first parameter for the Call function
intent.putExtra("peer", "1234"); //the second parameter for the Call function
intent.putExtra("requestid", ++mysequencenumber); //optional. useful if you wish to receive the function result
sendBroadcast(intent); //send the broadcast
```

Notifications via Broadcast Intents

It is also possible to configure AJVoIP to send the notifications as broadcast intents.
This is a rarely used feature (since you can catch the notifications directly in your code), but it might be useful in some circumstances, for example if you wish to notify some other/third-party app about the SIP stack events.

This feature can be configured with the following settings:

- **broadcastintent_name**: android broadcast intent name. For example com.example.app.SIPNOTIFICATION.
- **broadcastintent_events**: the notification(s) to be sent as intents. Multiple notifications can be separated by comma. "ALL" means all notifications.
- **broadcastintent_line**: if set, then the notification will be triggered only from this endpoint line. -9 or null means global status and all lines (no line filter), -2 means all lines (not the global status), -1 means global state, 1+ means an individual line
- **broadcastintent_dir**: if set, then the notification will be triggered only if direction match this value. -9 or null means all directions (no dir filter), 1 means out, 2 means in (useful for STATUS)
- **broadcastintent_filter**: if set, then the notification will be triggered only if it contains this substring

You can set any parameter to "null" to clear the previous value if any.
It is also possible to configure a list of notifications (up to 99) by using broadcastintent1_name, broadcastintent2_name … broadcastintentN_name; using broadcastintent1_event broadcastintent2_event … broadcastintentN_event; etc parameters.

If you wish to make them explicit intents (to avoid any Android OS limitations), then set the followings parameters:

- **explicitintent_package**: set to your app package name (the app which will receive the notifications from your app with AJVoIP) if you wish explicit intents
- **explicitintent_class**: set to the app BroadcastReceiver class name (if not set, then it will default to explicitintent_package.SipNotification)

The following intent extras will be set:

- **notificationfull**: the whole notification string (for example: "DTMF,1,5")
- **eventname**: the notification name (for example "STATUS", "DTMF", "CDR")

- **key1**: value1 (for each of the notification parameters where key is the parameter name as written in this documentation and value is the parameter string)
- **key2**: value2 (for example line: 1. See the notifications chapter for the exact name of the parameters mentioned as “Template string” for all of them)
- …
- **keyN**: valueN
- **GLOBAL_SEQ**: an auto increment sequence number (if this is useful for you)
- **GLOBAL_USERNAME**: local username (the main account username if you are using multiple/extra accounts)
- **GLOBAL_SERVERADDRESS**: the configured SIP server address
- **GLOBAL_DEVICEID** a device ID calculated from the machine settings (such as computer name, hardware, OS version, language, etc)
- **GLOBAL_GUID**: an auto generated GUID for this AJVoIP instance (generated and saved at first launch)

All of the intent extras are set as String. You might convert it to other types such as Integer or Boolean when necessary.

Example to get the incoming DTMF keys as broadcast intents:

Configure AJVoIP:

```
mysipclient.SetParameter("broadcastintent_name", "com.example.app.SIPNOTIFICATION");
mysipclient.SetParameter("broadcastintent_event", "DTMF");
```

In your manifest file (AndroidManifest.xml):

```
<manifest ...>
  <application ...>
    ....
    <receiver android:name=".SIPNotificationReceiver"
      android:exported="true">
      <intent-filter>
        <action android:name="com.example.app.SIPNOTIFICATION" />
      </intent-filter>
    </receiver>
    ...
  </application>
</manifest>
```

Note: if you SIPNotificationReceiver BroadcastReceiver is not in your main package, then you will have to set the package name explicitly.
Example: receiver android:name="com.example.anotherpackage.SIPNotificationReceiver"

Notifications handler code:

```
public class SIPNotificationReceiver extends BroadcastReceiver {
        @Override
        public void onReceive(Context context, Intent intent) {
                if(intent.getStringExtra("eventname").equalsIgnoreCase("dtmf")) {
                        Log.w(TAG, "DTMF " + intent.getStringExtra("msg") + " received on line " + intent.getStringExtra("line"));
                }
        }
}
```

Note: You might use the intent.getExtras() to list all the extras for the intent.

For your understanding, this is how AJVoIP will send the notifications as broadcast intents (this is the simplified code inside AJVoIP, you don’t have to use this anywhere):

```
Intent intent = new Intent(broadcastintent_name);
if(explicitintent_package.length() > 0) {
        if(explicitintent_class.length() < 1) explicitintent_class = explicitintent_package+".SipNotification";
        intent.setComponent(new ComponentName(explicitintent_package, explicitintent_class));
}
intent.putExtra("notificationfull", notificationstring_all);
intent.putExtra("eventname", notificationname);
for(int i=0;i<param_count;i++) {
        intent.putExtra(param_name[i], param_value[i]);
}
intent.putExtra("GLOBAL_SEQ", seq++);
intent.putExtra("GLOBAL_USERNAME", username);
intent.putExtra("GLOBAL_SERVERADDRESS", serveraddress);
intent.putExtra("GLOBAL_DEVICEID", deviceid);
intent.putExtra("GLOBAL_GUID", guid);
context.sendBroadcast(intent);
```

Native libraries

AJVoIP is implemented mostly in pure Java, however it also includes some native libraries for codec and audio processing.
The SDK works fine even without these native libraries since all of them has Java failback implemented.

The following ABI's are included by default, which will cover 99% of the devices with native audio and codec support:

- armeabi-v7a
- arm64-v8a

We found that this is the optimal set of libraries to be able to cover all platforms with minimal additional size.
The rest will failback to java code (thus you will have 100% device coverage!)

A good estimation about the market can be learn from here. See the CPU models chart at the bottom of the page.

- armeabi-v7a: this alone covers 98% of all devices on the market
- arm64-v8a: this is for the 64bit ARM architecture which is required by the Google Play
- x86: less then 1% of devices are using Intel x86 devices and they can use the java implementation
- mips: we include only fake support for mips so the Google Play will not exclude these kind of devices. MIPS market size is near 0% and on these kind of devices AJVoIP will just failback to pure java code
- armeabi: not supported. This is a very old architecture so we haven't included armeabi libs due to size considerations as its market size is near 0% and also Android removed ARMv5/ARMv6 support since Android 4.4 (API level 19). We neither included a fake library because we found that if included then some devices might load the armeabi library even if they have support for armeabi-v7a
- x86_64: there are no such devices on the market and it can failback to java if required by some new devices released in the future
- mips64: there are no such devices on the market and anyway it is covered with the fake mips support with Java fallback

If somehow you wish to change the included native libraries, you can download the whole set from here. For example you might include the armeabi libs if your app uses some other native library and you wish to provide support for these ancient devices for some reason.

Dependencies

AJVoIP uses one or more of the following Android libraries, depending on your build (listing here in case if somehow you encounter some incompatibility issues due to other versions used by your app):

- android.jar
- com.android.support:multidex:1.0.3 (classes.jar, to split the output)
- androidx.legacy:legacy-support-v4:1.0.0

Old libraries (not required anymore in latest version, but you might use some of these in your app):

- firebase-messaging-10.0.1.aar (in classes.jar, might be required for push notifications)
- firebase-core-10.0.1.aar (in classes.jar, might be required for push notifications)
- firebase-analytics-10.0.1.aar (in classes.jar, might be required for push notifications)
- firebase-analytics-impl-10.0.1.aar (in classes.jar, might be required for push notifications)
- firebase-iid-10.0.1.aar (in classes.jar, might be required for push notifications)
- firebase-common-10.0.1.aar (in classes.jar, might be required for push notifications)
- play-services-tasks-10.0.1.aar (classes.jar)
- play-services-basement-10.0.1.aar (classes.jar)
- support-v4-24.0.0.aar (in classes.jar, might be required for forward compatibility)
- support-v4-24.0.0.aar (in internal_impl-24.0.0.jar, might be required for forward compatibility)
- support-annotations-24.0.0.jar (might be required for forward compatibility)
- org.apache.http.legacy.jar (might be required for backward compatibility)

How to work with logs?

Set the loglevel parameter to 5 and then the logs will appear on the Logcat (or can be polled with the GetLogs() function call).
You might look at the followings in the logs:

- Search for "catch", "ERROR" and "WARNING" messages to find any issues. Please note that logs generated with loglevel 3 or higher will often includes lot's of not so important "WARNING" messages which can be safely ignored
- Check for the SIP signaling. You can find out register or call issues by just following the SIP signaling logs. (Find out the suspicious message then search for it's Call-ID value across the log to walk trough the messages which belongs to the same session)
- Search for "call details" to get a high level overview about your calls(s). A "call details" statistics is generated after each call.
- Other log messages might be too specific or technical but you might still be able to have a gasp about the cause of the problems by looking at the logs

ERROR messages in the log

If you set AJVoIP loglevel higher than 1 than you will receive messages that are useful only for debug. A lot of ERROR and WARNING message cannot be

considered as a fault. Some of them will appear also under normal circumstances and you should not take special attention for these messages. If there are any issue affecting the normal usage, please send the detailed logs to Mizutech support (info@mizu-voip.com) in a text file attachment.

Some internal exceptions (such as "java.lang.NumberFormatException") are normal in some circumstances and can be safely ignored.

RTP warning in my server log

AJVoIP will send a few (maximum 10) short UDP packets (\r\n) to open the media path (also the NAT if any).
For this reason you might see the following or similar Asterisk log entries:

WARNING[8860]: res_rtp_asterisk.c:2019 ast_rtp_read: RTP Read too short
Unknown RTP Version 1

These packets are simply dropped by Asterisk which is the expected behavior. This is not a AJVoIP or Asterisk error and will not have any negative impact for the calls. You can safely skip this issue.
You might turn this off by the "natopenpackets" parameter (set to 0). You might also set the "keepaliveival" to 0 and modify the "keepaliveival" (all these might have an impact on AJVoIP NAT traversal capability)

Recording errors

You might see the following errors if AJVoIP doesn't have permission for the audio device:

E/AudioRecord: AudioFlinger could not create record track, status: -1
E/AudioRecord-JNI: Error creating AudioRecord instance: initialization check failed with status -1.
E/android.media.AudioRecord: Error code -20 when initializing native AudioRecord object.
WARNING,recorder state is 0

Solution: make sure that the app has RECORD_AUDIO permissions.

Method exceeds compiler instruction limit

You should enable multidex as described here:

```
dependencies {
        implementation 'com.android.support:multidex:1.0.3'
}
```

How to send logs?

In short:
Set the AJVoIP loglevel parameter to 5 and send the full logcat output to info@mizu-voip.com.

Details:
If you run into any issue with the Android SIP library, Mizutech support most probably will ask you to send a problem description (your question, bug report or any integration issue) and logs about the problem.

For this the loglevel parameter must be set to 5 and then the logs will appear on the **logcat** with the severity level set to "Verbose" (or can be polled with the GetLogs() function call or logs written to file).
Once you reproduced the problem, send the content of the logs to info@mizu-voip.com as email text file attachment and an exact description of the problem (also include instructions for reproductions if possible).

Make sure that the log contains also the AJVoIP startup (send the logs from the very beginning, not only the part when the problem might appear).
If the problem is call related, then make sure to have only one call in the log. That one in which the problem occurs.
If your log contains multiple calls, please let us know which is a wrong call (SIP Call-ID, log line or any other details after which we can find that call in the log, or just stop the first time when the problem happens so we can know that it is the very last call in the log).

In addition to the log file, Mizutech support might ask one or two SIP account valid on your server to be able to reproduce the problem.

In case if AJVoIP is integrated into your app, make sure that your app has some debug functionality, capable to collect and save/export/send AJVoIP logs, implementing some kind of log upload functionality in your app (uploading the logs to a web service, uploading to FTP or sending it by email).
This can be useful if some issue can't be reproduced at development (when you have your IDE with logcat), so the users can report problems with logs.
You can collect the logs in the following ways:

- Set the events parameter to 3 and collect the log events (EVENT and LOG notifications) in your app (write to file or store some in memory)
- Configure AJVoIP to write logs to file by setting the canlogtofile parameter to true. Use the GetLogPath() function to get the log file path
- Set the logpolling parameter to 1 and use the GetLogs() API function to get the logs as strings.

This should be configurable in your app (when turned on, set the loglevel parameter to 5 and use some of the above methods to collect the logs)

Resources

Android VoIP SDK home-page
Android VoIP SDK Download

Sample project
Quick Start Guide
Documentation (PDF)
Documentation (HTML)
Documentation (WinHelp)
Javadoc (download)
Javadoc (online)
Native integration
For help, contact info@mizu-voip.com