Documentation for Android Video Dialer

# Android basics

1. Activity
   * Package: android.app.Activity
   * Reference: <http://developer.android.com/reference/android/app/Activity.html>
   * API Guide: <http://developer.android.com/guide/components/activities.html>
2. Manifest file
   * API Guide: <http://developer.android.com/guide/topics/manifest/manifest-intro.html>
   * See specially: <uses-feature>, <uses-permission>, <intent-filter>
3. Layout
   * API Guide: <http://developer.android.com/guide/topics/ui/declaring-layout.html>
4. R file
   * Reference: <http://developer.android.com/reference/android/R.html>
5. View
   * Package: android.view.View
   * Reference: <http://developer.android.com/reference/android/view/View.html>
   * See specially:
     + android.widget.LinearLayout  
       <http://developer.android.com/reference/android/widget/LinearLayout.html>
     + android.widget.RelativeLayout  
       <http://developer.android.com/reference/android/widget/RelativeLayout.html>
     + android.view.SurfaceView  
       <http://developer.android.com/reference/android/view/SurfaceView.html>
     + android.view.SurfaceHolder  
       <http://developer.android.com/reference/android/view/SurfaceHolder.html>
     + android.view.VideoView  
       <http://developer.android.com/reference/android/widget/VideoView.html>
     + android.widget.Button  
       <http://developer.android.com/reference/android/widget/Button.html>
6. Intent
   * Package: android.content.Intent
   * Reference: <http://developer.android.com/reference/packages.html>
   * API Guide: <http://developer.android.com/guide/components/intents-filters.html>
7. Preferences
   * Package: android.preference.Preference
   * Reference: <http://developer.android.com/reference/android/preference/Preference.html>
   * See also:
     + android.preference.PreferenceCategory  
       <http://developer.android.com/reference/android/preference/PreferenceCategory.html>
     + android.preference.PreferenceGroup  
       <http://developer.android.com/reference/android/preference/PreferenceGroup.html>
     + android.preference.PreferenceManager  
       <http://developer.android.com/reference/android/preference/PreferenceManager.html>
     + android.preference.PreferenceScreen  
       <http://developer.android.com/reference/android/preference/PreferenceScreen.html>
     + android.preference.PreferenceActivity  
       <http://developer.android.com/reference/android/preference/PreferenceActivity.html>
     + android.content.SharedPreferences  
       <http://developer.android.com/reference/android/content/SharedPreferences.html>
8. Log
   * Package: android.util.Log
   * Reference: <http://developer.android.com/reference/android/util/Log.html>

# Projects to look at

1. API Demos

* Camera preview: com.example.android.apis.graphics.CameraPreview
* Playing video: com.example.android.apis.media.VideoViewDemo
* Preferences: com.example.android.apis.app.PreferencesFromXml, com.example.android.apis.app.PreferencesFromCode

1. SpyDroid
   * Link: <http://code.google.com/p/spydroid-ipcamera/>
   * Source code: [http://code.google.com/p/spydroid-ipcamera/source/browse/  
     #svn%2Ftrunk%2Fsrc](http://code.google.com/p/spydroid-ipcamera/source/browse/#svn%2Ftrunk%2Fsrc)

# Test projects

1. Simple Camera 4
   * Camera selection
   * Custom camera preview
   * Recording video
2. Streaming Camera
   * Capturing video and writing to socket
   * RTP packetization of captured stream
3. Video Player
   * Receiving RTP stream
   * Implementation of RTSP server
   * Playing video on VideoView
4. RTSP Server
   * Desktop RTSP server
   * Redirecting RTP stream from one phone to another

# Final project

Duplex video: Simultaneous streaming and receiving video between two devices.

# Topics to know

## Video codecs

There are two codecs supported by android API: H.263 and H.264. According to the Android Supported Media Format specifications (<http://developer.android.com/guide/appendix/media-formats.html>), H.263 is available for encoding and decoding on all versions and H.264 encoding is available on devices running android 3.0+ (decoding available for all). But it was seen that the phones running android 2.3.6 (Samsung Galaxy B7510, Ace, Y, S2, S3, HTC Desire, One V) can encode H.264 videos.

Though H.263 encoded video is supposed to be simpler and easier to handle, only Galaxy Y was capable of playing H.263 video stream with very low quality. No other phone could play the streamed H.263 video. But H.264 video could be played on almost all of the phones (Galaxy S2 was an exception).

So, finally H.264 was chosen as the codec to work with, and H.264 encoded videos show much better quality than H.263 on most of the phones at low bitrate (e.g. 80 kbps).

## H.264 video format

H.264 video is encoded into segments called NAL (Network Abstraction Layer) units. In files recorded to SD card, all the NAL units are stored under ‘mdat’ atom. The mdat atom, along with other blocks, is stored within a container depending on file format. The other blocks contain some information most of which are not necessary for streaming. The other container blocks are added later to the file, after recording is finished.

But when writing recorded data to socket directly, these blocks are not written. All we need to do while streaming is to skip until the mdat atom appears, then read the NAL units one by one and send them. The NAL units can be identified in one of two ways: by detecting a special bit sequence (aka start code) between NAL units, or by reading the length of next NAL units from a 4-byte-long value. All the android devices follow the second method, so the parsing was done accordingly.

While streaming H.264 video in RTP packets, the RTSP server requires two parameters of the video stream: SPS and PPS. These values are not parts of the streamed data, they can be found in a video file with proper headers. For this reason, a sample video of duration ~1 second (1 second is used for Galaxy tab, it cannot a record video file with duration less than that. For other devices 400-500 milliseconds is enough) is recorded first to the SD card. This file is then read and parsed. From the ‘avcC’ block, the SPS and PPS parameters can be found. The values of SPS and PPS depends on the recording parameters (resolution, bit rate etc.) so these values should be obtained before each streaming session.

## Handling camera

Android camera class can be found in package android.hardware.camera. There are methods for finding the number of camera or accessing any particular camera using its ID. In this project camera was not handled explicitly, it was just indicated that the MediaRecorder object will use camera (back camera by default) as the video source. The MediaRecorder class handles the camera by its own.

To enable the user to choose any camera of the device, acquiring and releasing camera resource should be handled manually. In that case the intended camera should be opened, initialized and then unlocked so that MediaRecorder can access it. When streaming is done, the camera should be locked again and released. This feature has yet not been added.

When working with cameras, the application manifest file must have these two lines:  
<uses-feature android:name=*"android.hardware.camera"* /> and   
<uses-permission android:name=*"android.permission.CAMERA"* />

Camera resources should be handled with care. Once a camera is acquired, it must be ensured that it is released properly upon exit. Otherwise the camera would remain locked and no other application can access camera until a device restart.

Another problem with cameras is debugging. When the Camera class fails to do something, it provides a numeric error code that comes to no use. The problem may arise if the parameters are not set properly or the preview display is not prepared. But there will be no useful stack trace or error indication. So it is wise to stick to the code format on android reference and API guide pages.

## Showing camera preview

The preview from a camera should be shown in a SurfaceView. Before that the SurfaceHolder of it should be obtained, implementing its Callback interface. The methods of the callback interface are called when the SurfaceView is created, modified or destroyed. If the preview display is set before the surface is created, it will crash and provide some illegible error code. The preview display should be set after the SurfaceHolder.surfaceCreated() method is called.

If the application layout is designed to change while the device goes from portrait to landscape (or vice versa), the preview should be stopped, and then started again in the SurfaceHolder.surfaceChanged() method.

One more thing to note: the preview from a camera is in landscape orientation by default. For tabs this is natural, but awkward for phones. So when cameras are handled manually, the preview orientation should be set accordingly. In this project, the application layout is forcefully defined to be landscape, and the auto-rotation is disabled. If auto-rotation is enabled, the camera preview should be rotated as needed using the Camera.setDisplayOrientation() method.

The same SurfaceView can be used as the preview display for both Camera and MediaRecorder.

## MediaRecorder method calling sequence

The MediaRecorder class contains all the methods for capturing video. In this project, a custom Recorder class is defined which extends android.media.MediaRecorder class. The Recorder is defined only for recording video in a single track (i.e. no audio track). The customization was done for writing the captured video data directly to a LocalSocket’s file descriptor (<http://developer.android.com/reference/android/net/LocalSocket.html>) instead of a file. A separate MediaRecorder is used for recording the sample file. Both the recorders should have the same settings, so the settings were defined in a class named RecordingParameters, and the same RecordingParameters object was passed to both MediaRecorders.

There are some methods in MediaRecorder class that are to be called sequentially. The sequence is illustrated in the MediaRecorder reference page through a state diagram.

If the method calling sequence is somehow out of order, MediaRecorder will throw an IllegalStateException, with some illegible numeric error code. So the methods should be called in the right order.

## RTP packetization

The standard RTP packetization rules were followed in this project. A class named RTPSocket has been created, extending the DatagramSocket class. The maximum packet size is assumed to be 1400 bytes, including a 12-byte header. An RTPSocket contains a byte array which holds the packet data. This buffer is passed to the packetizer classes. This class itself handles generating a random SSRC value, assigning payload type and starting and maintaining sequence numbers.

The packetization process (especially for H.263) is inspired by the SpyDroid project. There is an abstract VideoPacketizer class that holds the basic functions, and two classes H263Packetizer and H264Packetizer extending it with different data parsing rules. The packetizer runs on a separate thread.

The packetization process is simpler for H.263 videos. The frames are separated by start codes and each frame can be put into one of more packets with two additional bytes of information.

For packetization of H.264 videos, the NAL units are read knowing their length, and sent in single or multiple packets. Each NAL unit contains its NAL header. If a large NAL unit is to be fragmented, one more fragmentation indicator byte is added. Moreover, some RTP packets containing the SPS and PPS are generated periodically and sent through the stream. This is done so that the RTSP server forwarding the stream to the receiver can obtain the SPS and PPS values to put in SDP.

## RTP timestamp

Timestamp is an important field of RTP packets. For H.264 videos the timestamp field is the only way to control the video output speed. So this should be set properly. Otherwise the output video can run in slow-motion causing an increasing lag, or fast forward with occasionally getting stuck.

Both H.263 and H.264 encoding has a sampling rate of 90000. So, the 4-byte timestamp value should advance by 90000 per second. To handle this, the timestamp value is set as the current system timestamp of the instant, multiplied by 90 (because the system timestamp is represented in milliseconds).

H.264 videos can cause some problems while assigning the timestamps. Some devices write all the NAL units of the next second at once, so the system timestamps assigned to them become very close. This creates a large gap between two consecutive NAL units: the last one of a second and the first one of the following second. While decoding, NAL units with close timestamps are decoded very fast, then the video gets stuck. This fast forward-stuck-fast forward sequence goes on.

Currently the system timestamp works very well for H.264, video transitions are very smooth. However, this may not be the case for all devices. In that case it may be needed to assign the timestamps by calculation and simultaneously preserving the synchronization.

## RTSP communication

Android devices do not support RTP streaming, but can play RTSP streams. So, the RTP stream can be played through RTSP. For this reason, a simple RTSP server has been implemented at the receiving end. The server is implemented in the receiver so that it can decode any generic RTP stream from any source.

The class RTSPServer is implemented through TCP ServerSocket, and it is designed to accept only one client. The receiver (client) just needs to request the server at the loopback address. The server communicates with the client using typical RTSP conversation to initiate the session. After this the server just forward the RTP packets received from the sender to the client. It also replies to the periodic request made by the client to keep the TCP connection alive.

For forwarding the video, there is an abstract class VideoPipe. This class is instantiated by the server depending on the video encoding. There are two different implementations of this class: H263Pipe and H264Pipe.

The server needs to compose an SDP string to describe the media format to player. For H.263 videos the SDP is a static one and it can be sent instantly. But for H.264, the SDP must contain some specific parameters (SPS and PPS). For this reason, the H264Pipe is designed in a way so that it can parse the packets and extract the periodic packets containing SPS and PPS. So it is required that the stream carries these packets at least once after RTSP session starts. SPS and PPS need not do be periodic, but sending those once a second ensures that the receiver can play it with a delay of at most one second.

## Playing video

The video is played in a VideoView, which is capable of playing RTSP streams. All we need to do is providing the RTSP URL to it. VideoView automatically conducts the RTSP conversations. For smooth streaming, it must be ensured that the RTSP server is running and the RTP stream is available before VideoView requests the server for it.

# Workflow

There are two basic modules in this project:

1. Sender
   * **package** sufian.video.stream
   * **package** sufian.video.stream.packetizer
2. Receiver
   * **package** sufian.video.server
   * **package** sufian.video.server.pipe

## Sender

The sending module starts with a Streamer object, which contains a Recorder and a VideoPacketizer. The Recorder does the recording jobs and writes the data to a LocalSocket’s OutputStream. The InputStream receiving that is passed to the VideoPacketizer for streaming. The VideoPacketizer runs in a separate thread which is spawned just after recording starts. The VideoPacketizer thread runs until Recorder is stopped.

## Receiver

In the receiving side, the RTSPServer is created first, which instantiates a VideoPipe. Both run in separate threads. The RTSPServer handles the receiver’s requests, but before that it makes sure that an RTP stream is available with necessary information. For H.263 videos it does nothing, but for H.264, it waits until the VideoPipe gets two packets containing SPS and PPS. Then it accepts only one client and runs until it gets a request to stop streaming.

There are three threads in this project that run simultaneously.

1. A thread for packetizing and sending RTP packets (sender)
2. A thread running the RTSP server (receiver)
3. A thread for receiving and forwarding the RTP packets (receiver)

Another package (**package** sufian.video.duplex) contains the GUI elements. This part loads the sender and receiver settings from the saved Preferences and instantiates the sender and receiver modules.

# Suggestions

## Front camera support

The project does not support using from camera now. Support for front camera should be added. For this, the camera has to be acquired separately and unlocked for MediaRecorder’s use. This should be done with care because releasing the acquired resources is essential.

## Defining settings as a package

Instead of setting the video resolution, frame rate and bit rate individually, different combinations of these parameters can be defined as packages of different quality (e.g. high, medium and low). The combinations can be defined though testing and bandwidth analysis.

## Pre-calculating SPS and PPS

If the video qualities are defined as some fixed packages, recording and parsing a sample file may not be needed. For each quality type the SPS and PPS can be extracted at the very first time from a dummy file, and these values can be saved for future usage.

## Sending SPS and PPS before playing

This application runs on its own, so it needs to wait for the SPS and PPS being available. But when it will run as a part of another application, the SPS and PPS can be sent beforehand, during call initiation. In that case the RTSP server will no longer need to wait for PPS and SPS packets to create SDP.

## Redefining timestamp assignment

Currently using the system timestamps in RTP packets is running very well, but it is not sure if it will do the same for all devices. It may cause the video getting stuck sometimes. So the application should be tested on as many devices as possible, and if any problem arises, the RTP timestamps should be assigned in some different way.