Android MultiMedia Framework seminar

YAN BIXING

MAR 14, 2019

博客地址:
https://blog.csdn.net/yanbixing123

扫码关注公共号

Overview

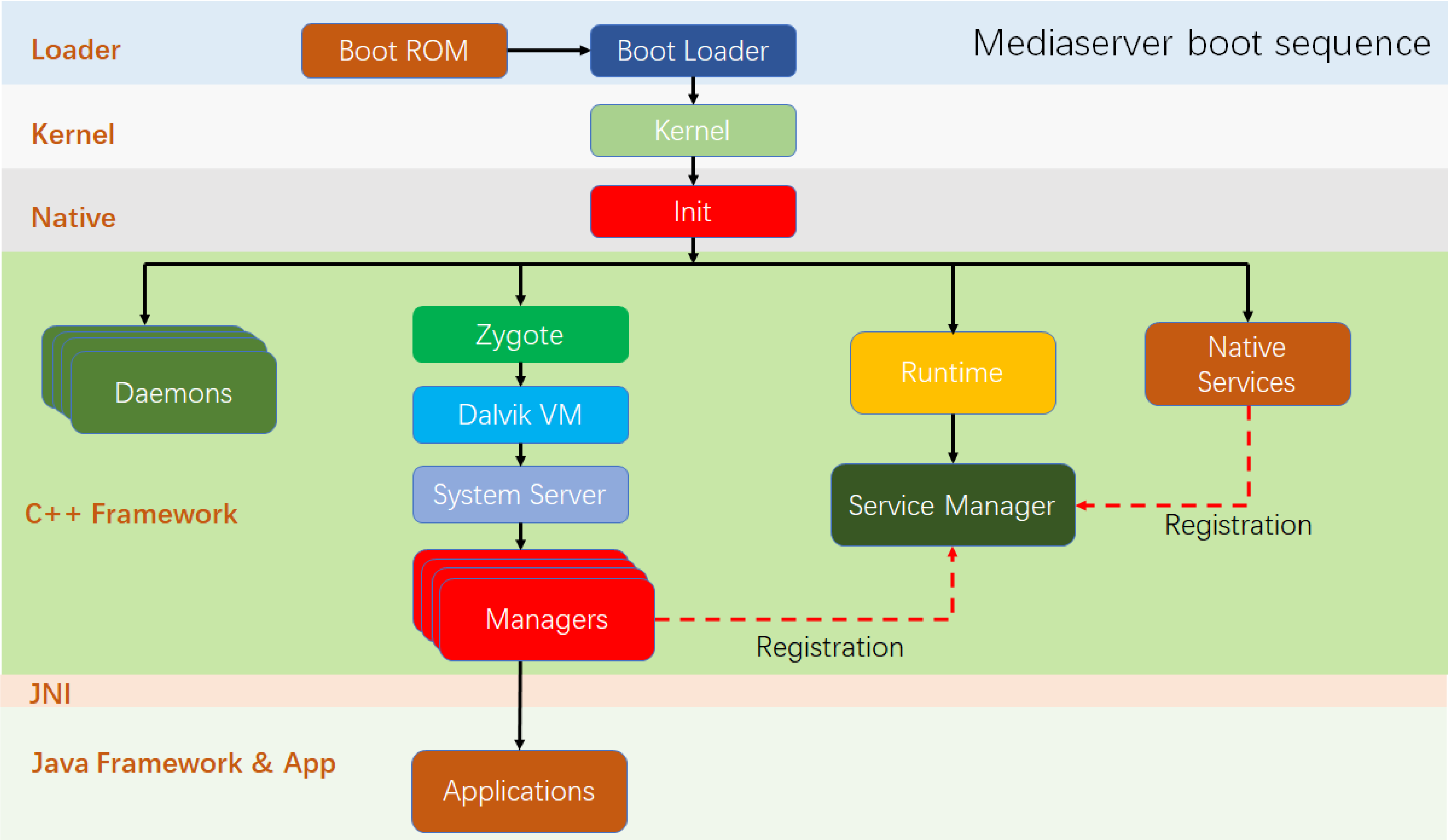
Android framework architecture

There are four layers in Android framework architecture, namely the application layer; the application framework layer; the system library and Android runtime ; Linux kernel.

The multimedia engine layer is located in the third layer of Android architecture. It will support each component through C/C++ libraries to provide us better service.

Mediaserver boot sequence

We know that Android is based on the Linux kernel. In Linux, the first process started is the init process, and other processes are child processes of the init process.
During the startup process of the init process, the configuration script init.rc file will be parsed. According to the contents of the init.rc file, the Init process will load the Android file system, create a system directory, and start the daemons of the Android system.
At the same time, the init process will also start important services such as Media Server (Multimedia Service) and ServiceManager (Binder Service Manager).
The init process also incubates the Zygote process. The Zygote process is the first Java process of the Android system. Zygote is the parent process of all Java processes.



Mediaserver

- In versions prior to Android m6.0, the startup script command for the mediaserver service was in the system/core/rootdir/init.rc file:

```
service media /system/bin/mediaserver
    class main
    user media
    group audio camera inet net_bt net_bt_admin net_bw_acct drmrpc mediadrm
    ioprio rt 4
```

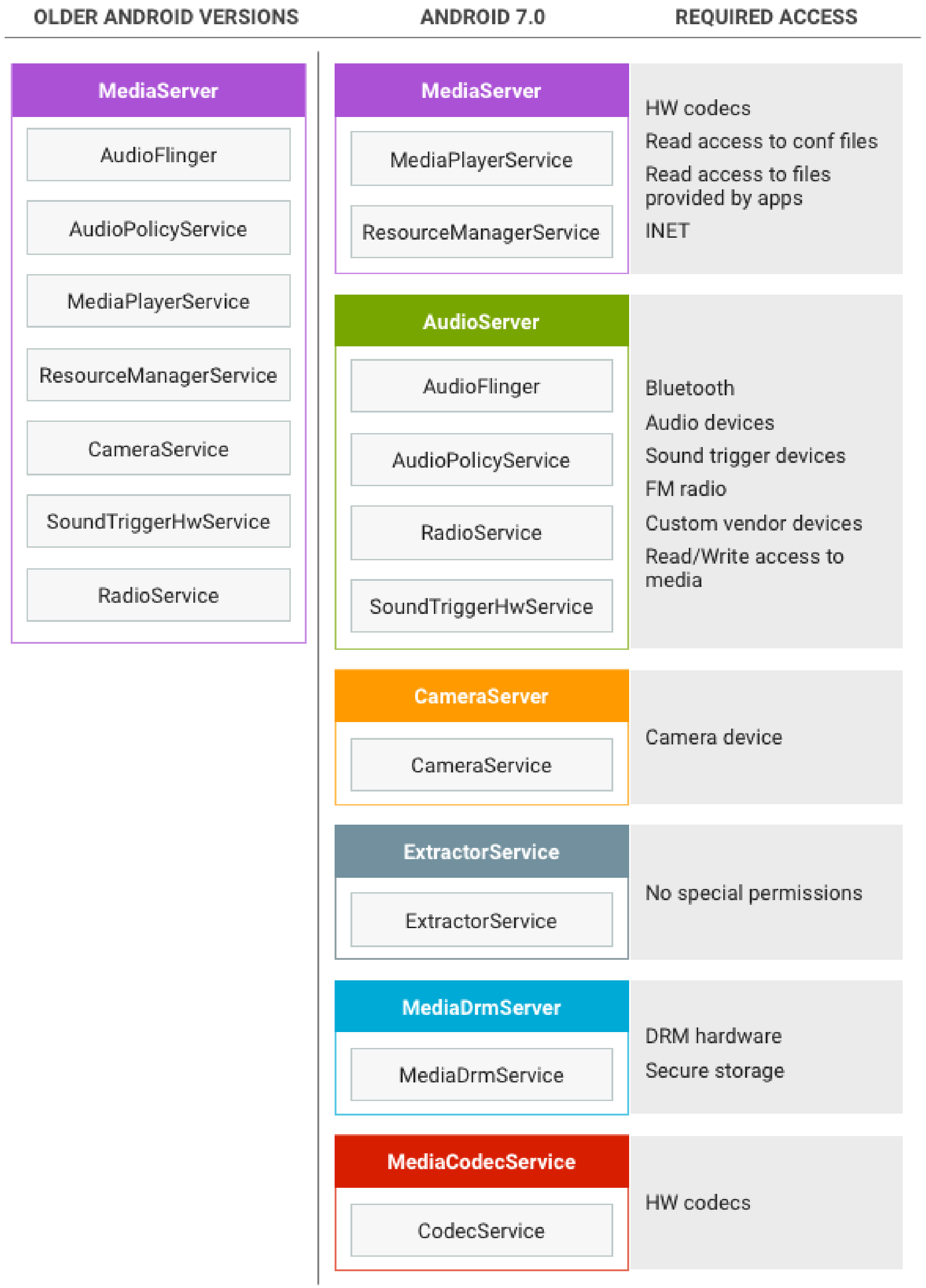
- After Android N7.0, the startup script for the mediaserver service is migrated to the system/core/rootdir/init.zygote64.rc file:

```
service zygote /system/bin/app_process64 -Xzygote /system/bin --zygote --start-system-server
    class main
    socket zygote stream 660 root system
    onrestart write /sys/android_power/request_state wake
    onrestart write /sys/power/state on
    onrestart restart audioserver
    onrestart restart cameraserver
    onrestart restart media
    onrestart restart netd
    writepid /dev/cpuset/foreground/tasks
```

MediaServer

Previous versions of Android used a single, monolithic mediaserver process with great many permissions (camera access, audio access, video driver access, file access, network access, etc.).

Android 7.0 splits the mediaserver process into several new processes that each require a much smaller set of permissions:



Function of each server

MediaServer: In Android 7.0, the mediaserver process exists for driving playback and recording, e.g. passing and synchronizing buffers between components and processes. Processes communicate through the standard Binder mechanism.

AudioServer: The AudioServer process hosts audio related components such as audio input and output, the policymanager service that determines audio routing, and FM radio service.

CameraServer: The CameraServer controls the camera and is used when recording video to obtain video frames from the camera and then pass them to mediaserver for further handling.

ExtractorServer: The extractor service hosts the *extractors*, components that parse the various file formats supported by the media framework.

MediaDrmServer: The DRM server is used when playing DRM-protected content, such as movies in Google Play Movies. It handles decrypting the encrypted data in a secure way, and as such has access to certificate and key storage and other sensitive components. Due to vendor dependencies, the DRM process is not used in all cases yet.

MediaCodecServer: The codec service is where encoders and decoders live. Due to vendor dependencies, not all codecs live in the codec process yet.

MEDIAPLAYERSERVICE

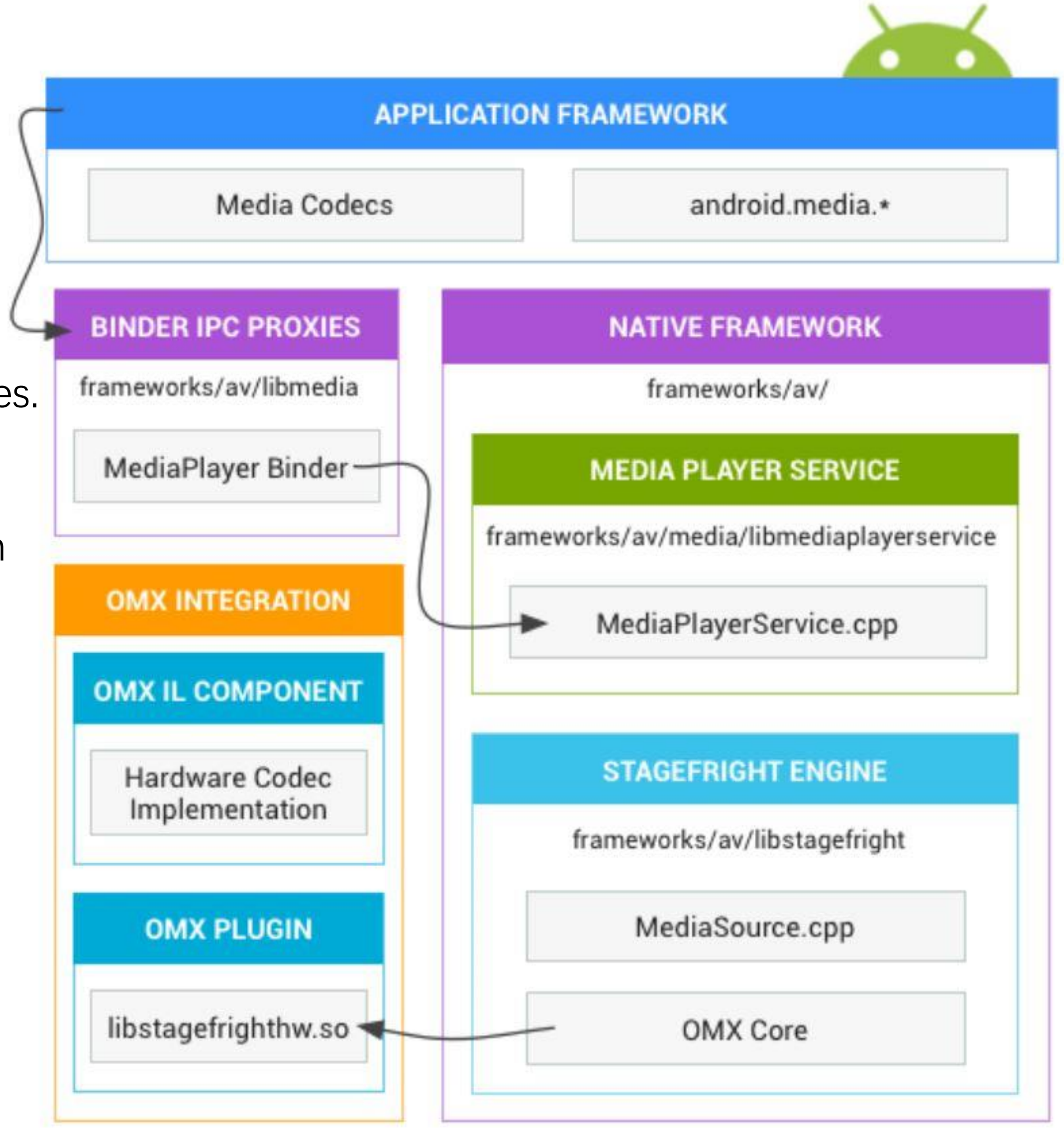
MediaServer

Application Framework
At the application framework level is application code that utilizes android.media APIs to interact with the multimedia hardware.

Binder IPC
The Binder IPC proxies facilitate communication over process boundaries.

Native Multimedia Framework
At the native level, Android provides MediaPlayerService for application layer services, each mediaplayer has a corresponding client in it and provides a state mechanism.
Android provides a multimedia framework that utilizes the Stagefright engine for audio and video recording and playback.
Stagefright comes with a default list of supported software codecs and you can implement your own hardware codec by using the OpenMax integration layer standard.

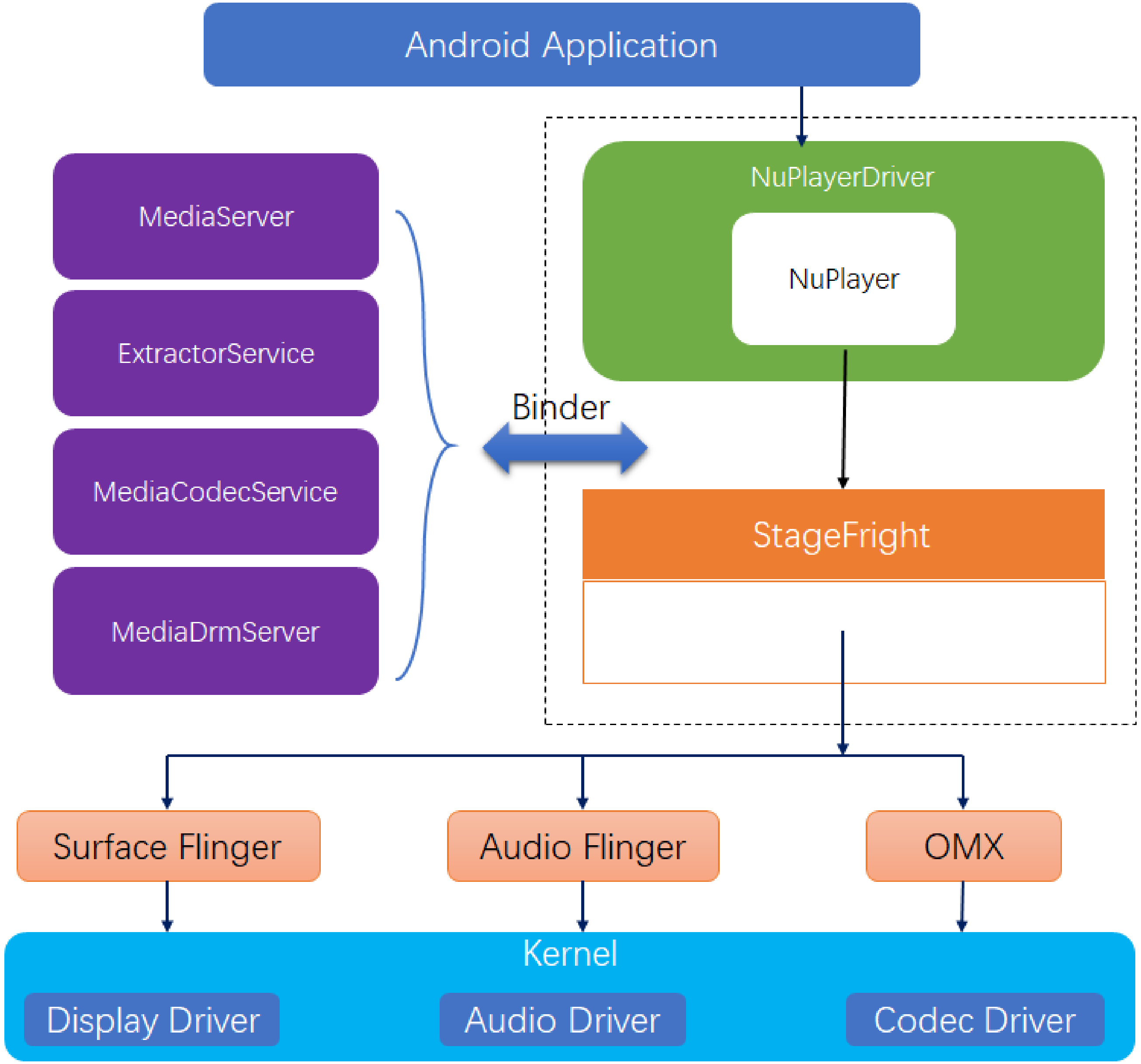
OpenMAX Integration Layer (IL)
The OpenMAX IL provides a standardized way for Stagefright to recognize and use custom hardware-based multimedia codecs called components.



NuPlayer works with the stagefright engine,
If it needs any service, it will communicate with the corresponding server.

Stagefright audio and video playback features include integration with OpenMAX codecs, session management, time-synchronized rendering, transport control, and DRM.

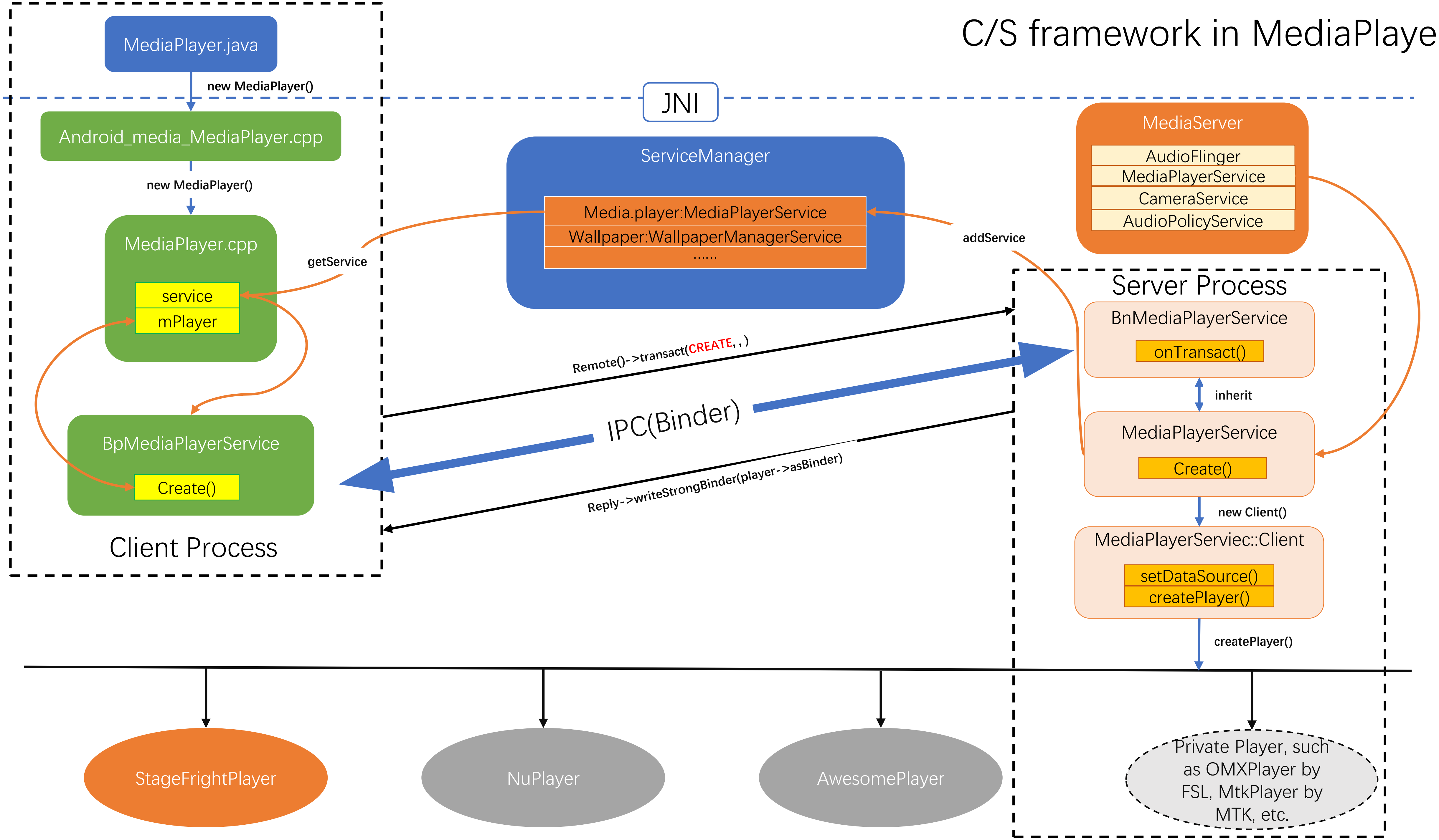
NuPlayerDriver is a Wrapper of NuPlayer, it support the state mechanism of media player.



Here is a simple process for an Android application to call MediaPlayer, we will analyze the following process around it.

```java
MediaPlayer mediaPlayer = new MediaPlayer();

mediaPlayer.setOnCompletionListener(new OnCompletionListener() {
    @Override
        public void onCompletion(MediaPlayer mp){
        mediaPlayer.release();
        mediaPlayer = null;
    }
});

mediaPlayer.setDataSource("abc.mp3");
mediaPlayer.setDisplay();
mediaPlayer.prepare();
mediaPlayer.start();
```

C/S framework in MediaPlayer



NUPLAYER AND EACH COMPONENT

History of multimedia framework

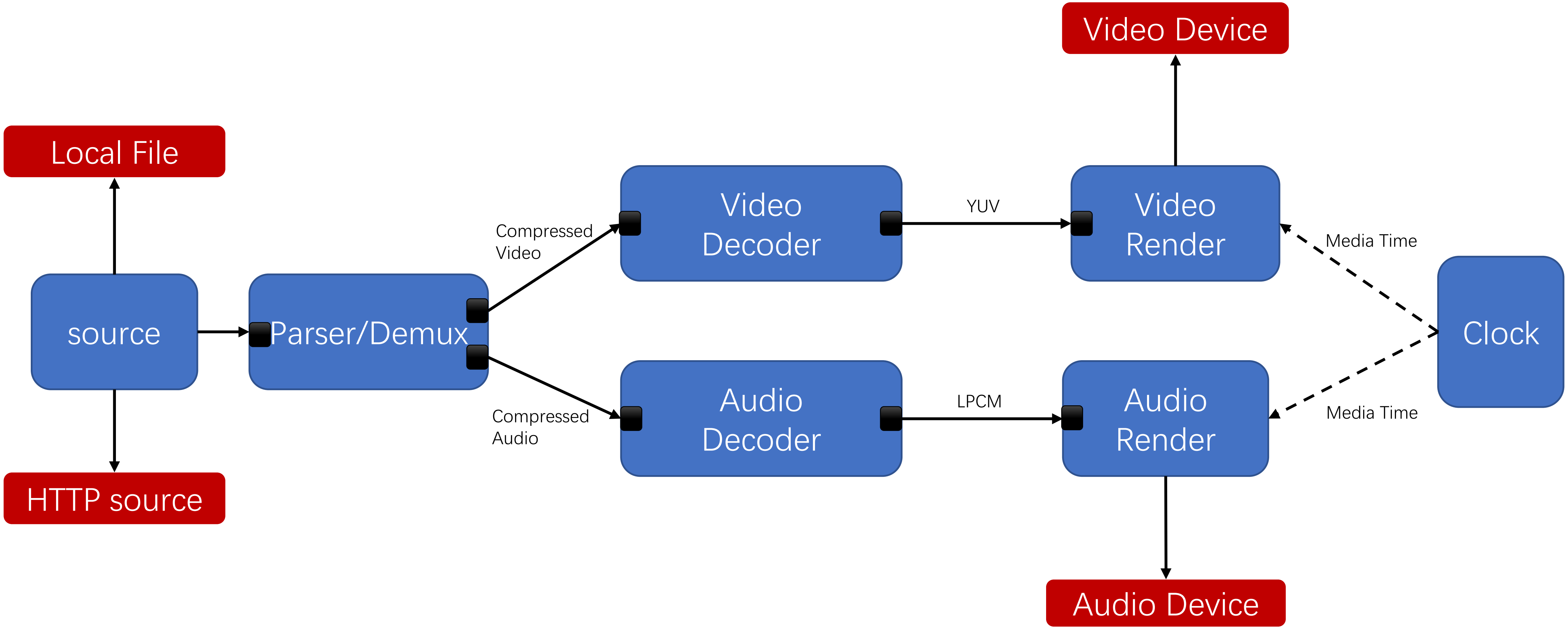
The streaming media framework was introduced in Android 2.3, and the core of the streaming media framework is NuPlayer. In previous versions, it was generally considered that Local Playback used Stagefrightplayer+Awesomeplayer, and streaming media used NuPlayer.

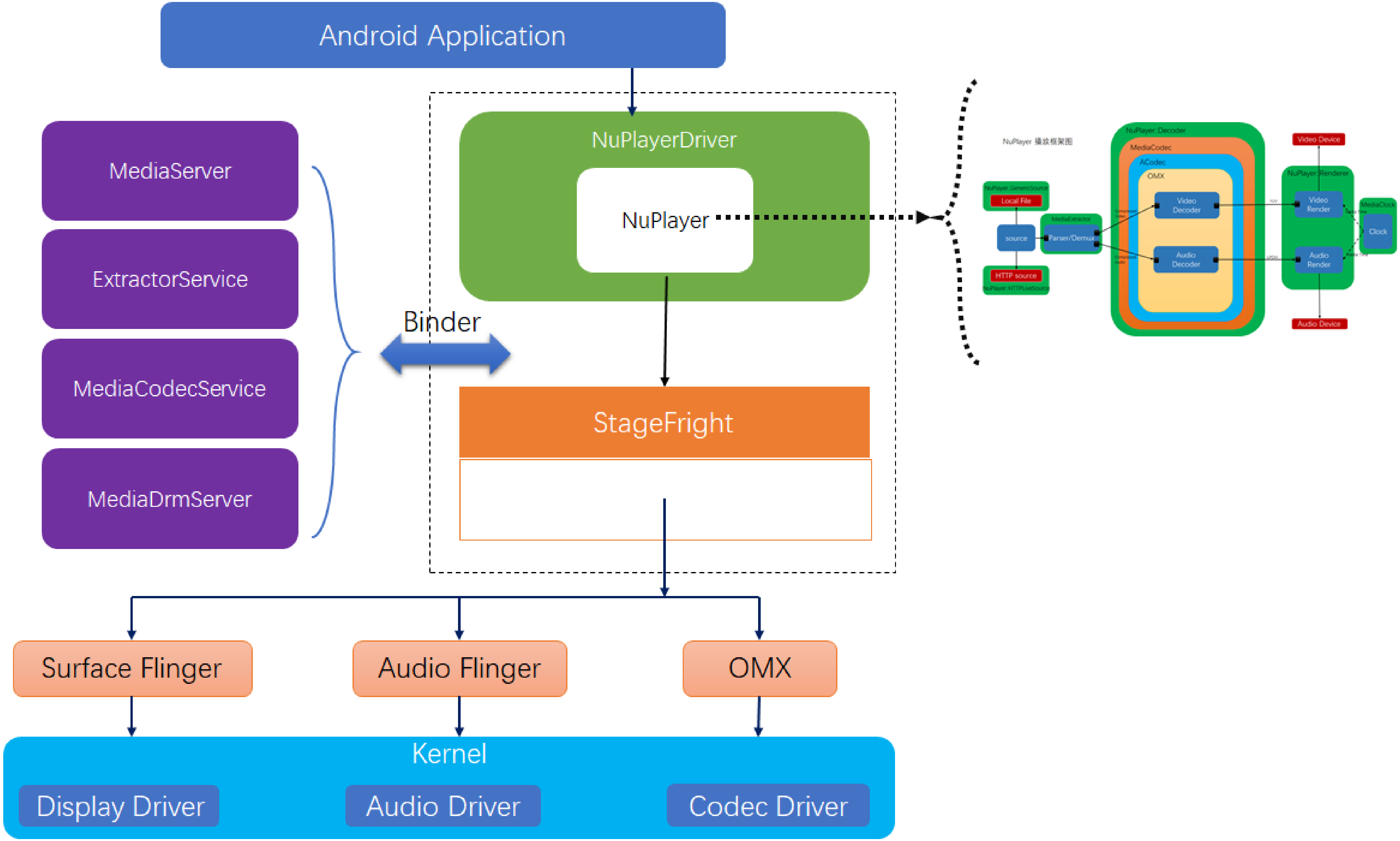
After Android4.0, HttpLive and RTSP protocols began to use the NuPlayer player; (FSL use GMPlayer)

After playing Android5.0 (L version), the local player also started to use the NuPlayer player.

Android7.0 (N version) completely removed Awesomeplayer. Popularly speaking, NuPlayer is a multimedia playback framework provided by AOSP. It can support local files, HTTP (HLS), RTSP and other protocols. It usually supports H.264, H.265/HEVC, AAC encoding formats, and supports MP4 and MPEG. -TS package. NuPlayer is based on the base class of Stagefright. It uses the lower-level ALooper/AHandler mechanism to process requests asynchronously. ALoole queues message requests and processes in AHandler, so there are fewer Mutex/Lock in NuPlayer. Awesomeplayer utilizes omxcodec and NuPlayer utilizes Acodec.

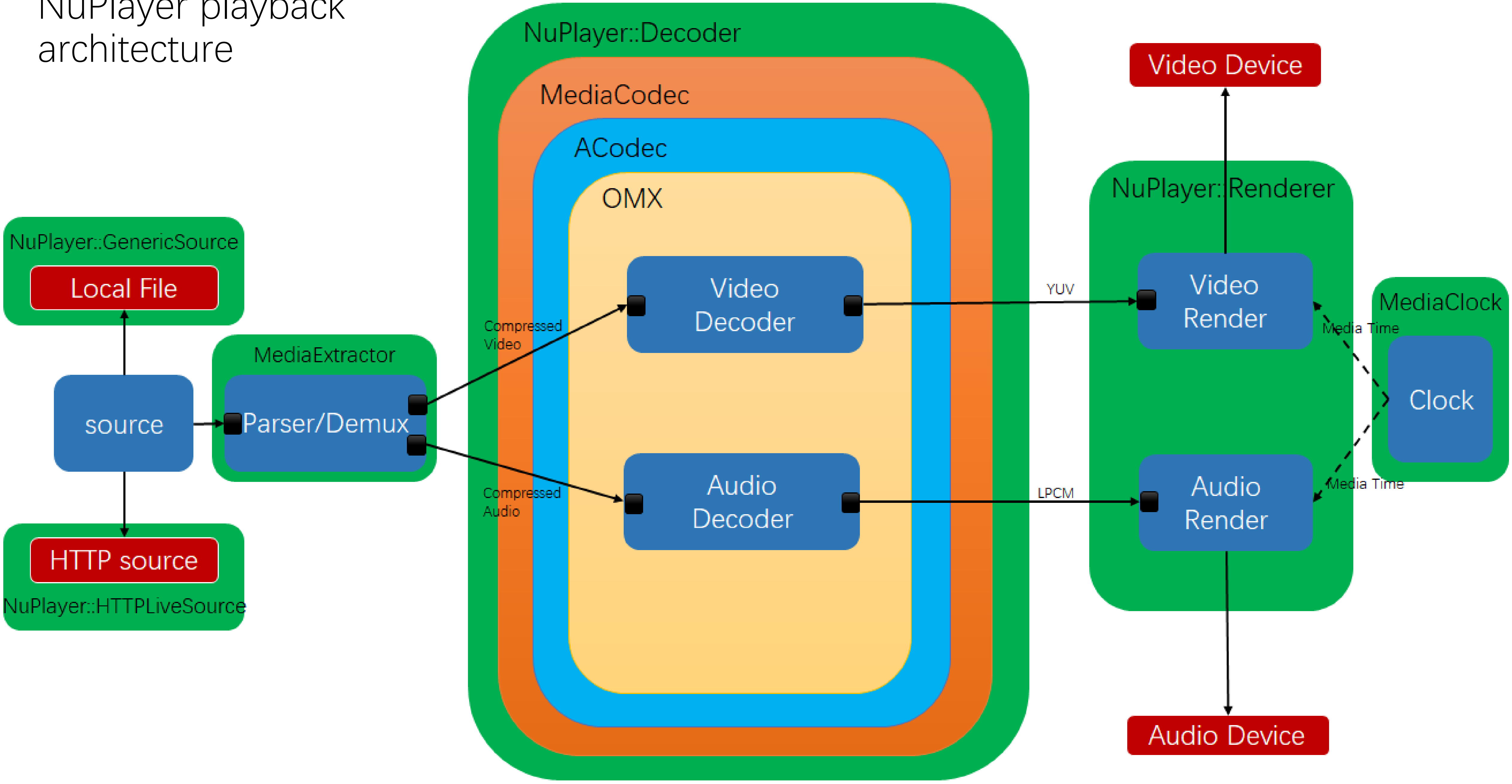
A normal player playback architecture





Android Application
NuPlayerDriver
NuPlayer
MediaServer
ExtractorService
MediaCodecService
MediaDrmServer
Binder
StageFright
Surface Flinger
Audio Flinger
OMX
Kernel
Display Driver
Audio Driver
Codec Driver
NuPlayer 播放框架图
NuPlayer::GenericSource
Local File
source
HTTP source
NuPlayer::HTTPLiveSource
MediaExtractor
Parser/Demux
NuPlayer::Decoder
MediaCodec
ACodec
OMX
Video Decoder
Audio Decoder
Video Device
NuPlayer::Renderer
Video Render
Audio Render
Audio Device
MediaClock
Clock

NuPlayer playback architecture



Function of each module in NuPlayer

（1）Source：the source of the data is not only a local file, but also various protocols on the Internet such as: http, rtsp and so on. The task of source is to abstract the data source to provide a stable data stream for Demux module, main functions: format detection of multimedia files, reading and parsing file.

（2）Parser/Demux： Video files are generally interlaced by the stream of audio and video through some rules. This kind of rule is the container rule. There are now many different container formats. Such as ts, mp4, flv, mkv, avi, rmvb and so on. The function of demux is to strip the stream of audio and video from the container and send it to different decoders. Demux will provides the data stream for decoder to decoding.

（3）Decoder： The core module of the player. Divided into audio and video decoders. The role of the audio and video decoder is to restore these compressed data (including MPEG1 (VCD) \ MPEG2 (DVD) \ MPEG4 \ H.264 and so on.) to the original audio and video data.

（4）Renderer： From a functional view, Renderer mainly has several functions: audio and video raw data buffer operation, audio playback (to the sound card), video display (to the graphics card), audio and video synchronization, and other auxiliary playback control operations.

（5） NuPlayer is the link between Source, Demux，Decoder and Renderer in this playback framework.

ALooper-AHandler-AMessage mechanism

NuPlayer is built on the basis of Stagefright's class, using a lower-level ALooper-AHandler-AMessage mechanism to process messages asynchronously.

AMessage acts as a message carrier and holds information related;

ALooper is a loop that runs a background thread to loop through the received message (transfer the information to the AHandler for processing, which is equivalent to a relay station);

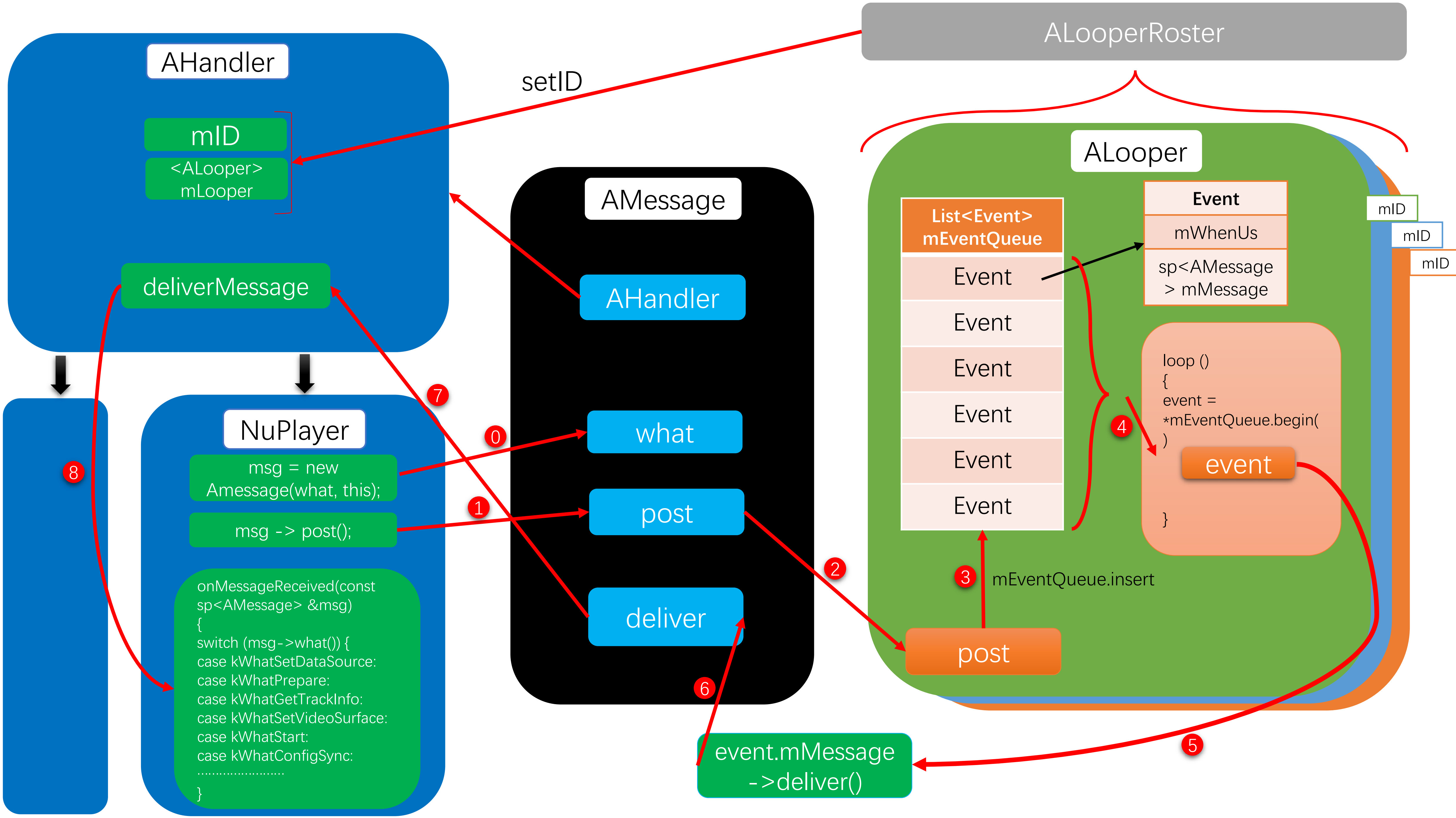
AHandler it is a handler, which is the final processing of the message.

Why is asynchronous mode heavily used in NuPlayer?
Because in the Media-related place, many operations are time-consuming operations,
but the user's tolerance for the smoothness of the picture is very low.

ALooper-AHandler-Amessage usercase

- registerHandler() will binding the ALooper and AHandler.
- AHandler need completed the onMessageReceived function, it include the true operation for each Amessage.
- Please see the next picture for other details.

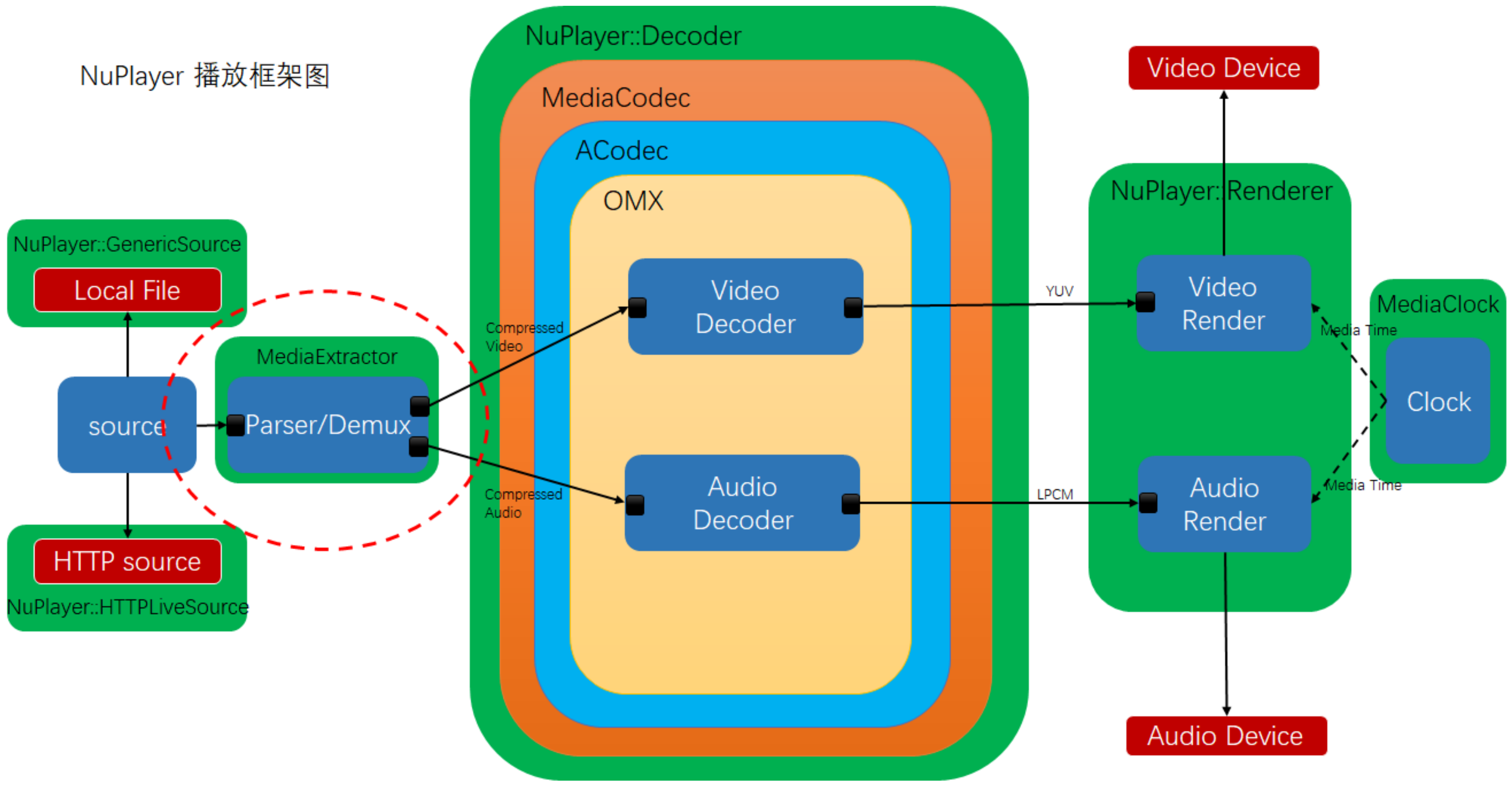
```
mLooper(new ALooper());
mLooper->setName(name);
mLooper->registerHandler(mHandler);
mLooper->start(
            false, // runOnCallingThread
            false, // canCallJava
            ANDROID_PRIORITY_FOREGROUND);


sp<AMessage> notify = dupNotify();
notify->setInt32("what", kWhatPrepared);
notify->setInt32("err", err);
notify->post();
```



ALooperRoster
setID
AHandler
mID
<ALooper> mLooper
deliverMessage
AMessage
AHandler
what
post
deliver
NuPlayer
msg = new Amessage(what, this);
msg -> post();
onMessageReceived(const sp<AMessage> &msg)
{
switch (msg->what()) {
case kWhatSetDataSource:
case kWhatPrepare:
case kWhatGetTrackInfo:
case kWhatSetVideoSurface:
case kWhatStart:
case kWhatConfigSync:
....
}
ALooper
List<Event> mEventQueue
Event
Event
Event
Event
Event
Event
Event
mWhenUs
sp<AMessage> mMessage
loop ()
{
event = *mEventQueue.begin()
event
}
mEventQueue.insert
post
event.mMessage ->deliver()
mID
mID
mID
0
1
2
3
4
5
6
7
8

MediaExtractor & MediaMuxer

- The Native layer in Android abstracts the MediaMuxer class and the MediaExtractor class. The MediaMuxer class is mainly used to mix audio and video data to generate multimedia files (such as mp4 files), while MediaExtractor is just the opposite, mainly used for Demux.



MediaExtractor usercase

- This class is mainly used for the separation of audio and video mixed data. The interface is relatively simple.
- Firstly, set the data source through setDataSource(String path) function. The data source can be a local file or a network stream address of HTTP protocol.
- Here we open a test video in sdcard and then print out its track information. The traversal of the track information can be done by MediaExtractor's getTrackCount and getTrackFormat.
- Track information includes: MimeType, resolution, encoding format, bit rate, frame rate and so on.
- After getting the details of the media file, you can select the specified track, and read the data.

```java
extractor = new MediaExtractor();
extractor.setDataSource("/sdcard/test.mp4");
dumpFormat(extractor);
private void dumpFormat(MediaExtractor extractor) {
    int count = extractor.getTrackCount();
    Log.i(TAG, "playVideo: track count: " + count);
    for (int i = 0; i < count; i++) {
        MediaFormat format = extractor.getTrackFormat(i);
        Log.i(TAG, "playVideo: track " + i + ":" + getTrackInfo(format));

        String mime = format.getString(MediaFormat.KEY_MIME);
        if(mime.startsWith("Video/")){
            videoTrackIndex = i;
        }
        else if(mime.startsWith("audio/")){
            audioTrackIndex = i;
        }
    }
}

mMediaExtractor.selectTrack(videoTrackIndex);
while(true) {
    int sampleSize = mMediaExtractor.readSampleData(buffer, 0);
    if(sampleSize < 0){
        break;
    }
    mMediaExtractor.advance(); //移动到下一帧
}

mMediaExtractor.release(); //读取结束后，要记得释放资源
```

MediaExtractor implementation by FSL

FSL provides different libraries for different formats to parse. The corresponding libraries are selected according to the file format. These libraries have a unified interface function, and these interfaces can be used to obtain relevant information. (But for different formats they have different implementations internally.)

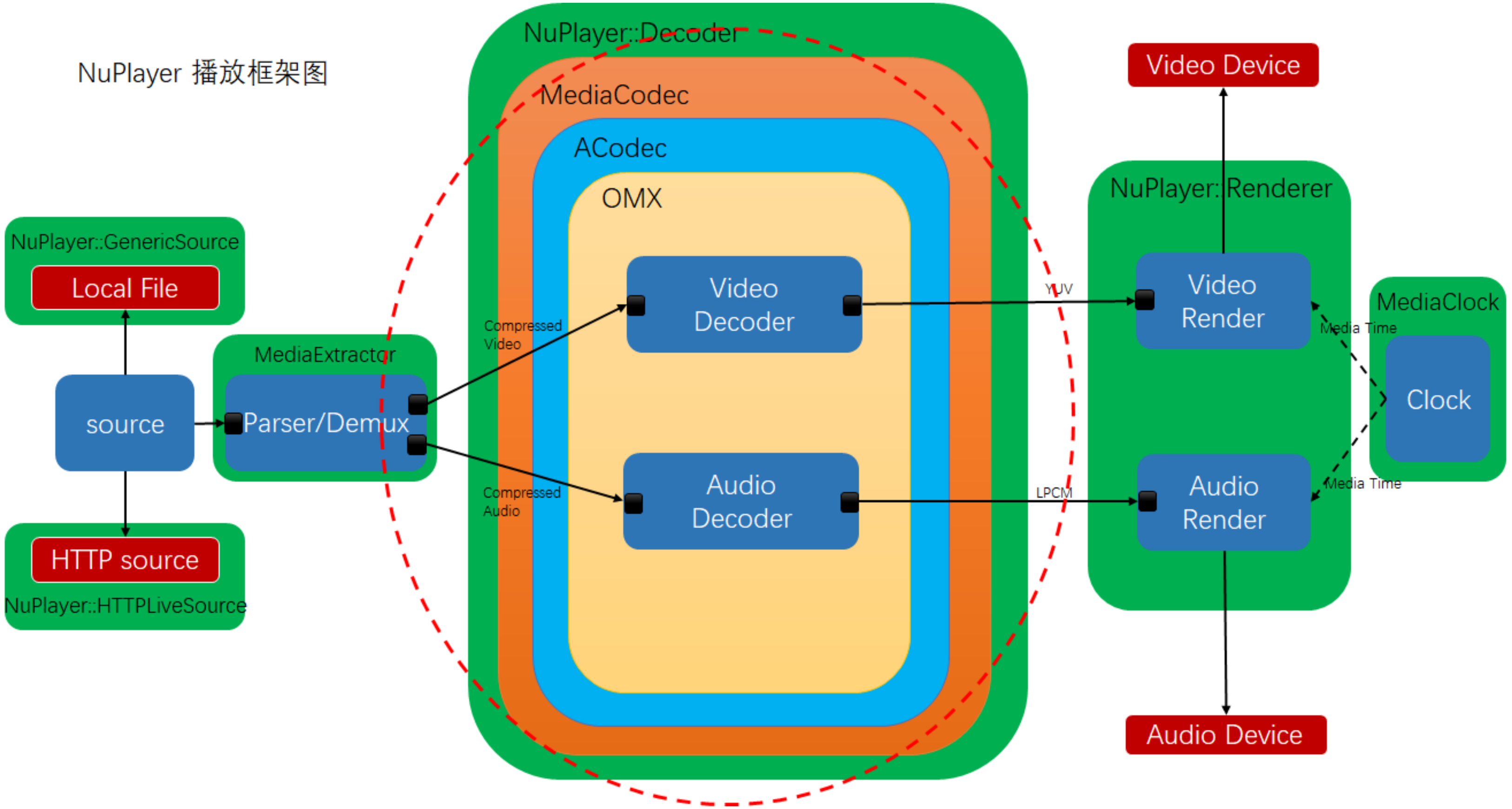
The interface is as follows:

```
 err = IParser->createParser2(flag,
                 &fileOps,
                 &memOps,
                 &outputBufferOps,
                 (void *)mReader,
                 &parserHandle);

 err = IParser->createParser(bLive,
                 &fileOps,
                 &memOps,
                 &outputBufferOps,
                 (void *)mReader,
                 &parserHandle);

err = IParser->setReadMode(parserHandle, mReadMode);
err = IParser->getNumTracks(parserHandle, &trackCnt);
err = IParser->initializeIndex(parserHandle);
err = IParser->isSeekable(parserHandle,(bool *)&bSeekable);
err = IParser->getMovieDuration(parserHandle, (uint64 *)&mMovieDuration);
err = IParser->getMetaData(parserHandle, USER_DATA_CAPTURE_FPS, &userDataFormat, &metaData, &metaDataSize);
err = IParser->getTrackDuration(parserHandle, index,(uint64 *)&duration);
err = IParser->getNumPrograms(parserHandle, &programCount);
err = IParser->seek(parserHandle, i, &sSeekPosTmp, SEEK_FLAG_NO_LATER);
err = IParser->getBitRate(parserHandle, index, &bitrate);
err = IParser->getVideoFrameWidth(parserHandle, index, &width);
err = IParser->getVideoFrameHeight(parserHandle, index, &height);
err = IParser->getVideoFrameRate(parserHandle, index, &rate, &scale);
```

For more details, you can see the: Android/frameworks/av/media/libstagefright/FslExtractor.cpp

MediaCodec

- MediaCodec is a Codec that accelerates decoding and encoding through hardware. It builds a unified interface for chip vendors and application developers. MediaCodec is almost a standard for all Android players. To analyze the source code of a player, such as NuPlayer, ijkplayer, it is necessary to understand the basic usage.



MediaCodec usecase - 1

- Android provides a MediaCodecList for enumerating the names and capabilities of codecs supported by the device to find the appropriate codec.
- The MediaFormat of the target track is retrieved from the MediaExtractor, and then the most suitable decoder can be obtained by codecList.findDecoderForFormat(format), and then the decoder can be created by MediaCodec.createByCodecName.

```
MediaFormat selTrackFmt = chooseVideoTrack(extractor);
codec = createCodec(selTrackFmt, surface);

private MediaFormat chooseVideoTrack(MediaExtractor extractor) {
    int count = extractor.getTrackCount();
    for (int i = 0; i < count; i++) {
        MediaFormat format = extractor.getTrackFormat(i);
        if (format.getString(MediaFormat.KEY_MIME).startsWith("video/")){
            extractor.selectTrack(i);//选择轨道
            return format;
        }
    }
    return null;
}

private MediaCodec createCodec(MediaFormat format, Surface surface) throws IOException{
    MediaCodecList codecList = new MediaCodecList(MediaCodecList.REGULAR_CODECS);
    MediaCodec codec = MediaCodec.createByCodecName(codecList.findDecoderForFormat(format));
    codec.configure(format, surface, null, 0);
    return codec;
}
```

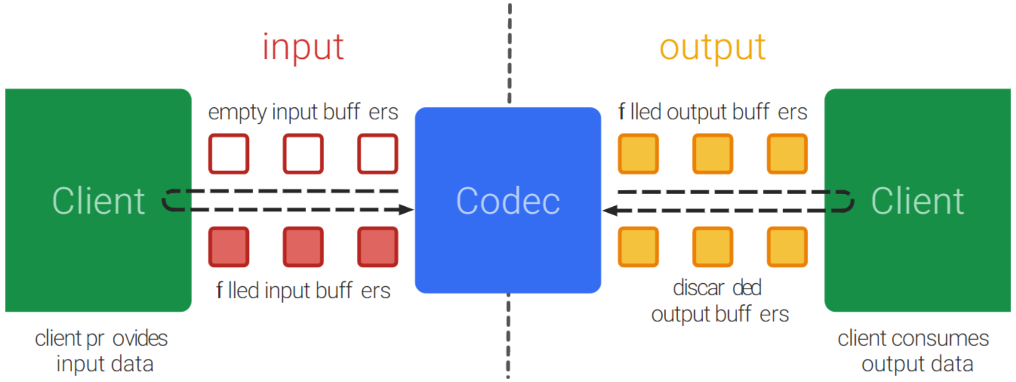
MediaCodec usecase - 2

- There are two ways to use MediaCodec - synchronous and asynchronous.
- The way to play asynchronously is to register a callback to MediaCodec, which is notified by MediaCodec when the input buffer is available, the output buffer is available, and the format is changed.
  What we need to do is to fill the data by MediaExtractor to the specified buffer when the input buffer is available; when the output buffer is available, decide whether to display the frame.

```
codec.setCallback(new MediaCodec.Callback() {
    @Override
    public void onInputBufferAvailable(MediaCodec codec, int index) {
        ByteBuffer buffer = codec.getInputBuffer(index);
        int sampleSize = extractor.readSampleData(buffer, 0);
        if (sampleSize < 0) {
            codec.queueInputBuffer(index, 0, 0, 0, MediaCodec.BUFFER_FLAG_END_OF_STREAM);
        } else {
            long sampleTime = extractor.getSampleTime();
            codec.queueInputBuffer(index, 0, sampleSize, sampleTime, 0);
            extractor.advance();
        }
    }

    @Override
    public void onOutputBufferAvailable(MediaCodec codec, int index, MediaCodec.BufferInfo info) {
        codec.releaseOutputBuffer(index, true);
    }

    @Override
    public void onError(MediaCodec codec, MediaCodec.CodecException e) {
        Log.e(TAG, "onError: "+e.getMessage());
    }

    @Override
    public void onOutputFormatChanged(MediaCodec codec, MediaFormat format) {
        Log.i(TAG, "onOutputFormatChanged: "+format);
    }
});
codec.start();
```

MediaCodec usecase – 3 – buffer sequence

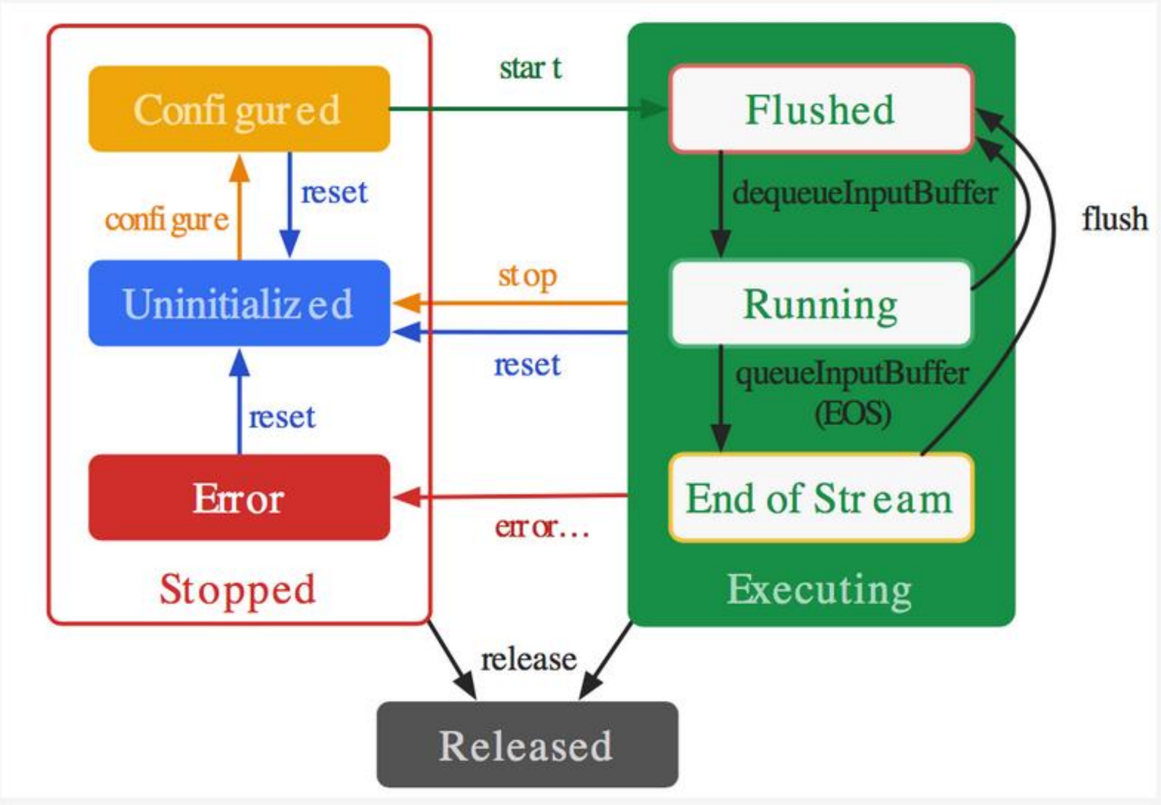
- In broad terms, a codec processes input data to generate output data. It processes data asynchronously and uses a set of input and output buffers. At a simplistic level, you request (or receive) an empty input buffer, fill it up with data and send it to the codec for processing. The codec uses up the data and transforms it into one of its empty output buffers. Finally, you request (or receive) a filled output buffer, consume its contents and release it back to the codec.



MediaCodec usecase – 4 – state mechanism

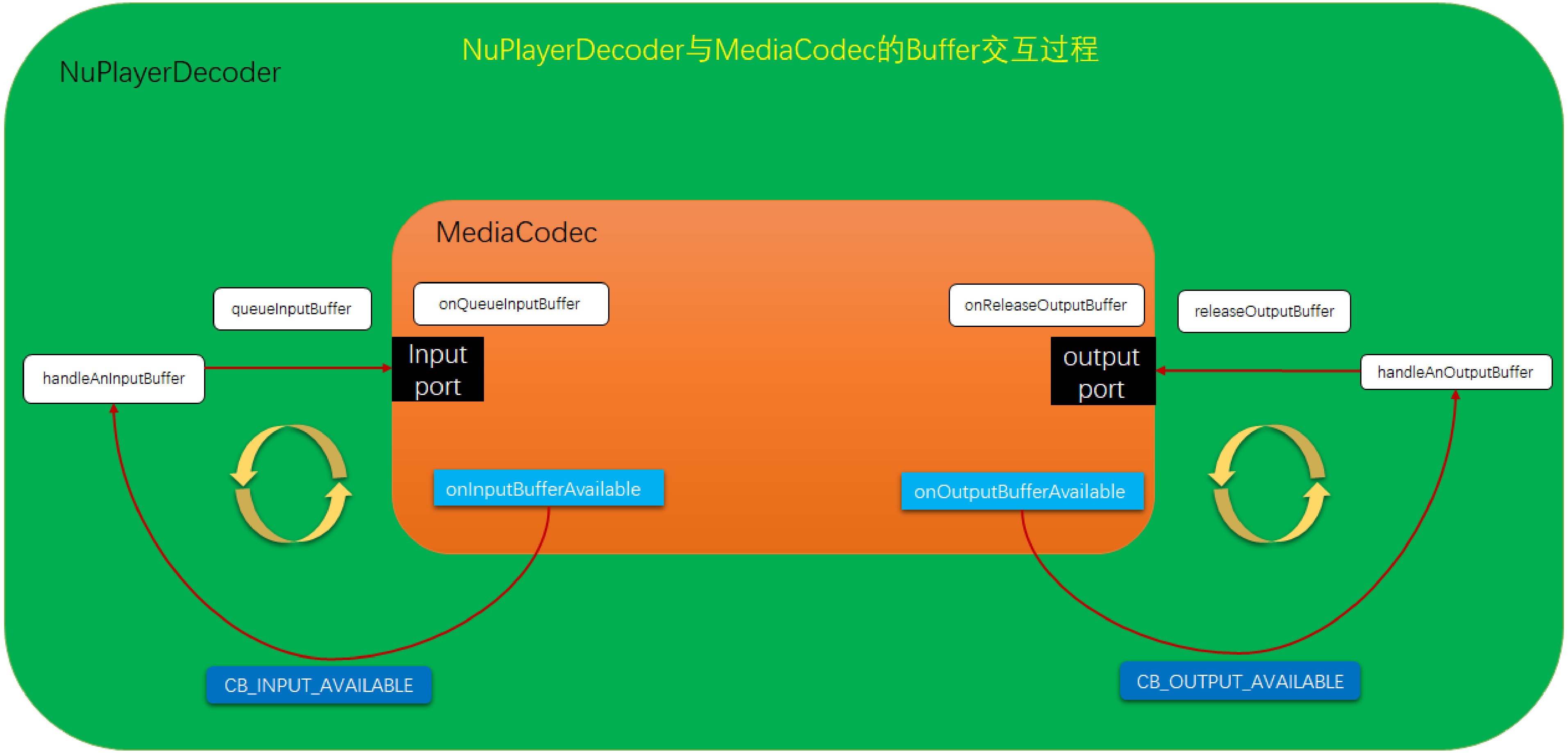
During its life a codec conceptually exists in one of three states: Stopped, Executing or Released. The Stopped collective state is actually the conglomeration of three states: Uninitialized, Configured and Error, whereas the Executing state conceptually progresses through three sub-states: Flushed, Running and End-of-Stream. When you create a codec using one of the factory methods, the codec is in the Uninitialized state. First, you need to configure it via configure(…), which brings it to the Configured state, then call start() to move it to the Executing state. In this state you can process data through the buffer queue manipulation described above.

The Executing state has three sub-states: Flushed, Running and End-of-Stream. Immediately after start() the codec is in the Flushed sub-state, where it holds all the buffers. As soon as the first input buffer is dequeued, the codec moves to the Running sub-state, where it spends most of its life. When you queue an input buffer with the end-of-stream marker, the codec transitions to the End-of-Stream sub-state. In this state the codec no longer accepts further input buffers, but still generates output buffers until the end-of-stream is reached on the output. You can move back to the Flushed sub-state at any time while in the Executing state using flush().



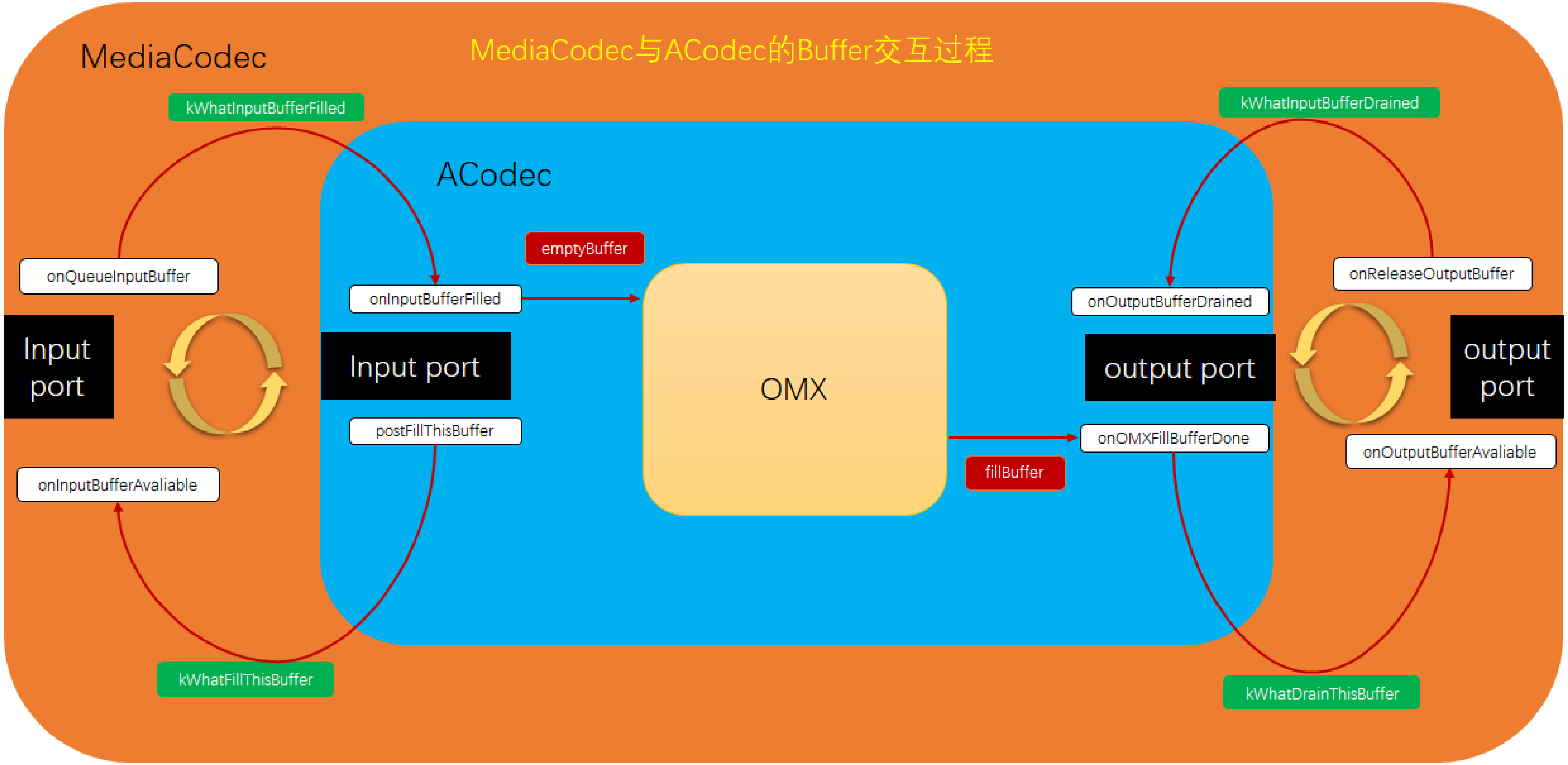
Call stop() to return the codec to the Uninitialized state, whereupon it may be configured again. When you are done using a codec, you must release it by calling release().

The Interaction between NuPlayerDecoder and MediaCodec

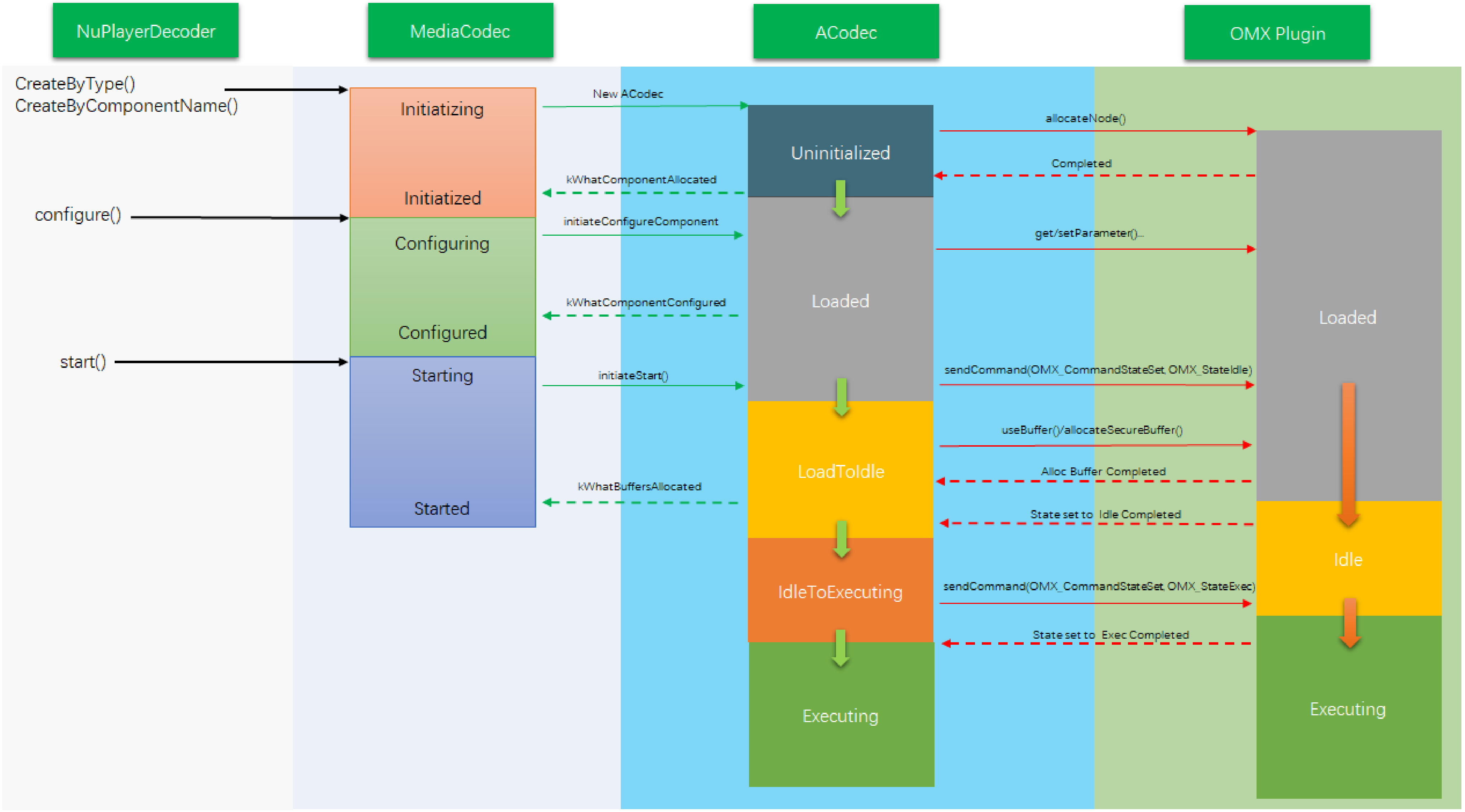


The Interaction between MediaCodec and ACodec

ACodec is used internally by MediaCodec to interact with the underlying OMX plugin and isolate it from MediaCodec.

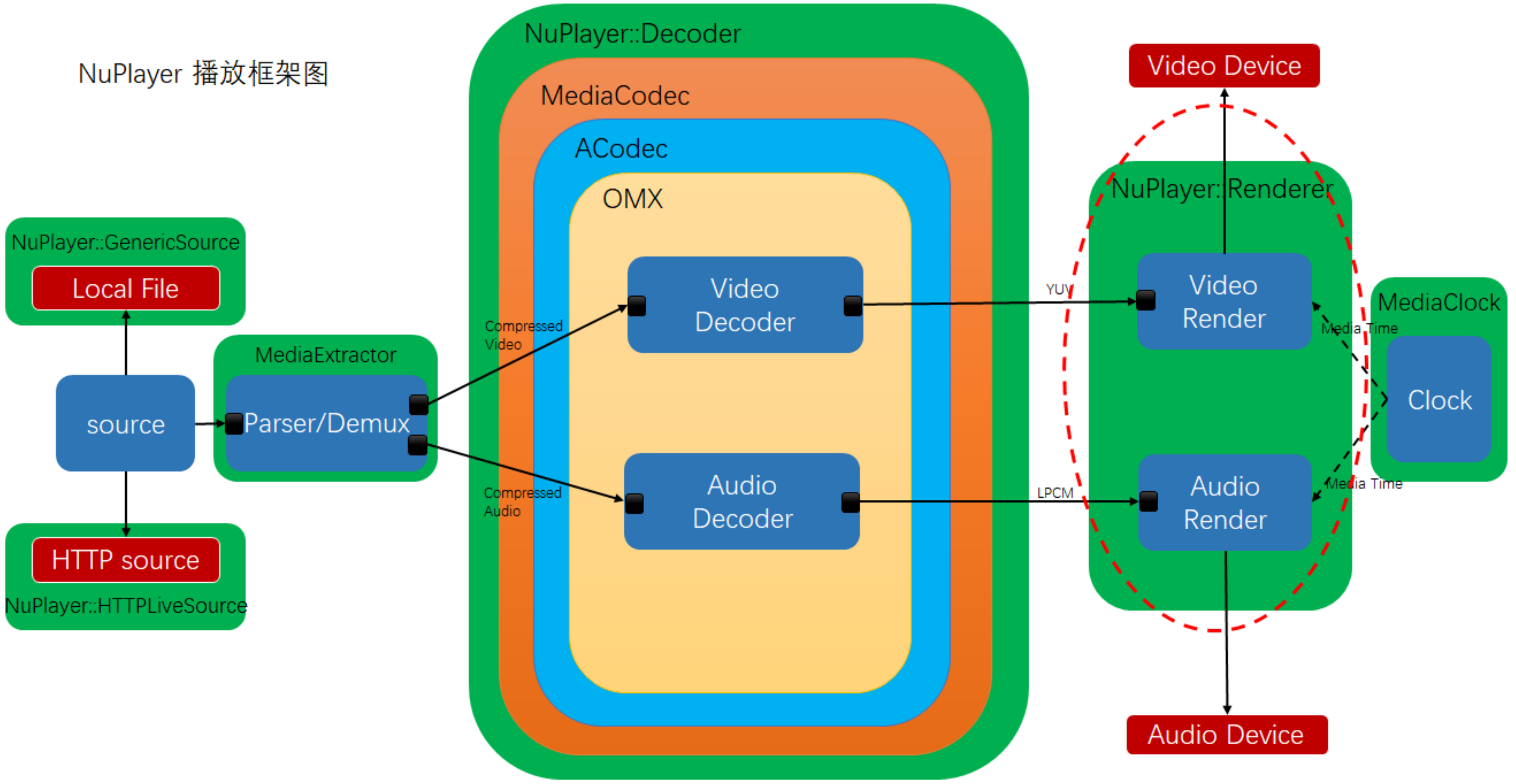


State changing between MediaCodec & Acodec & OMX Plugin

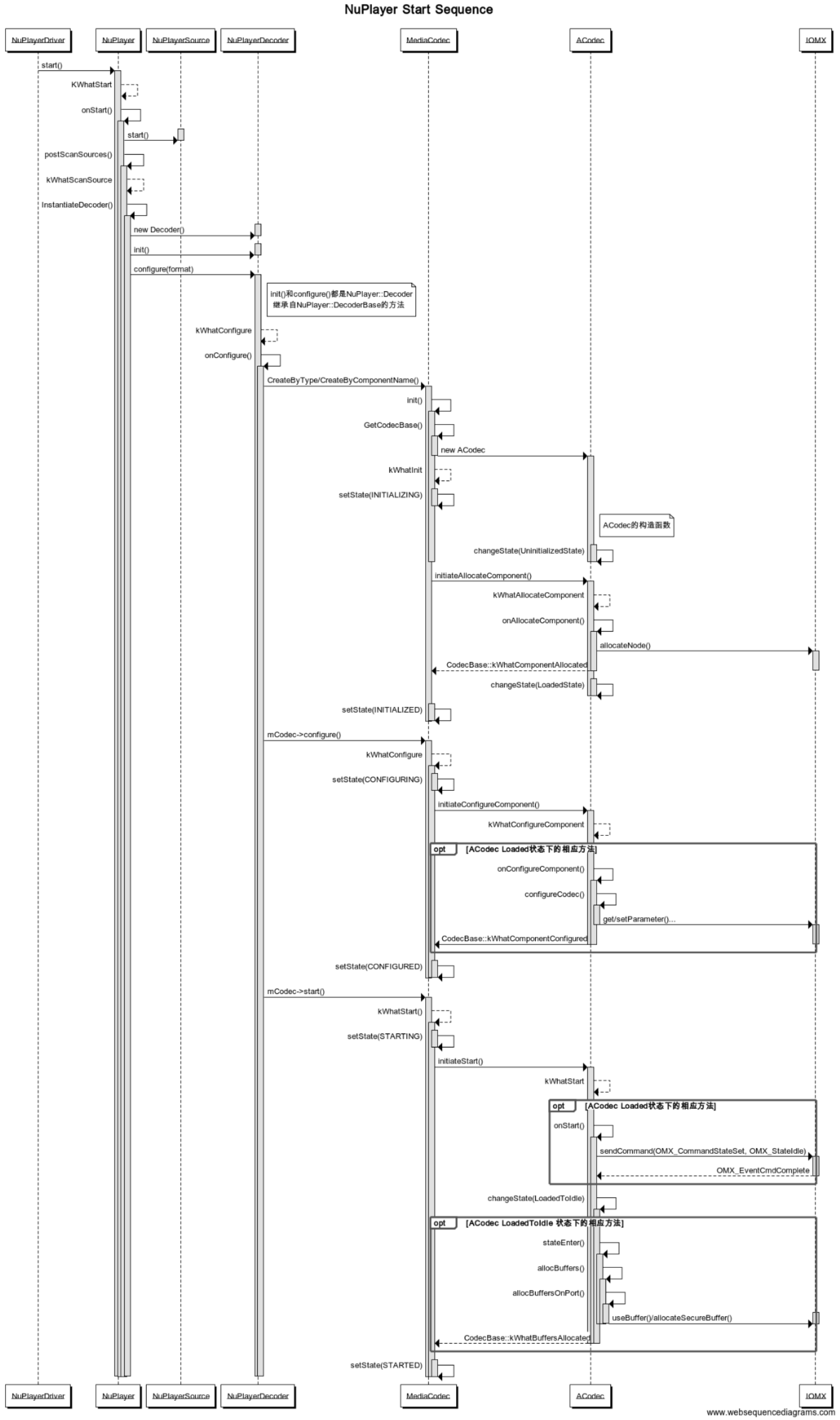


Renderer

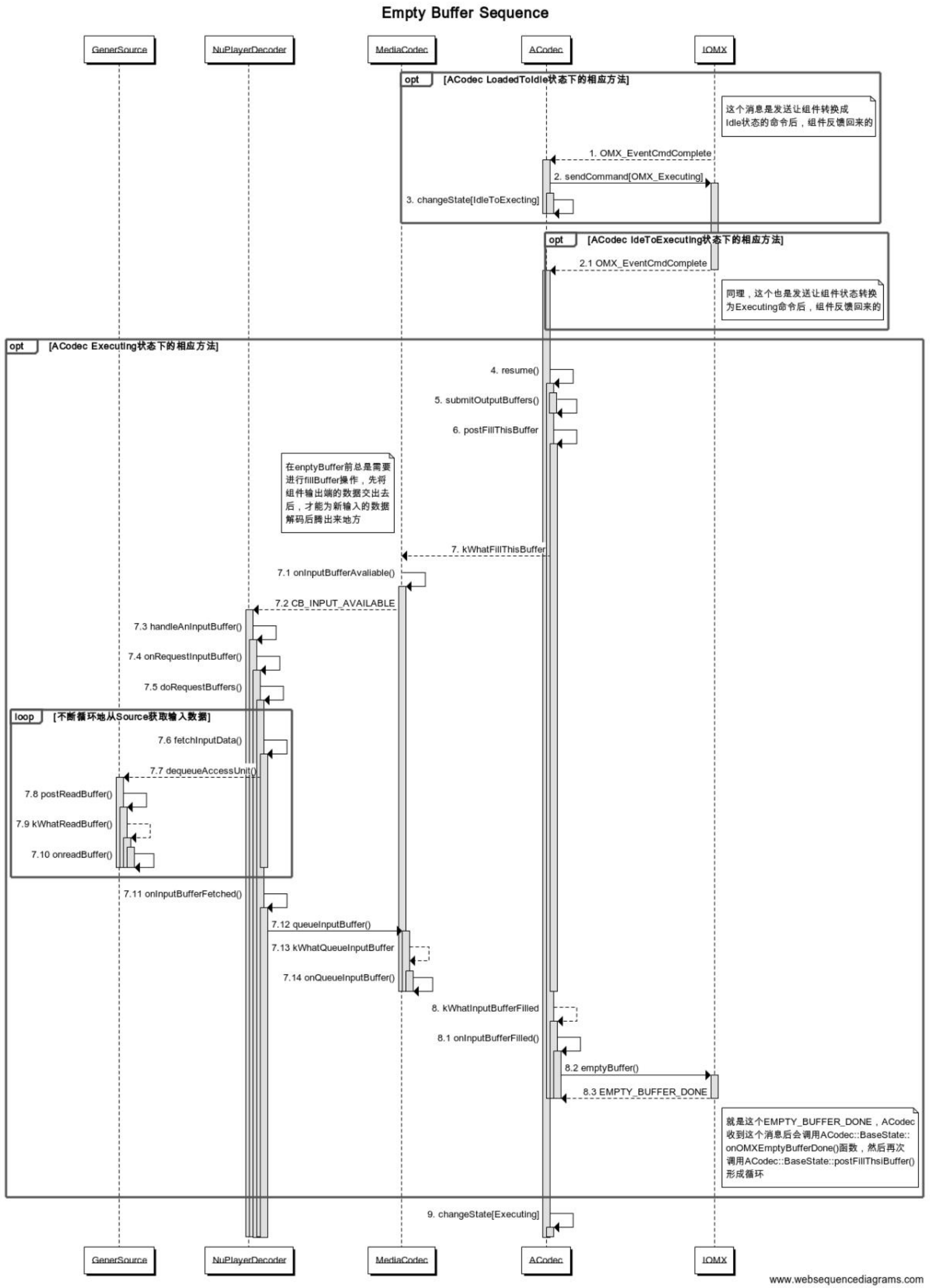
- The role of the Renderer is to determine whether the frame needs to be rendered based on pts of frame, and to synchronize the audio and video. But the real hardware rendering code is in MediaCodec and ACodec.



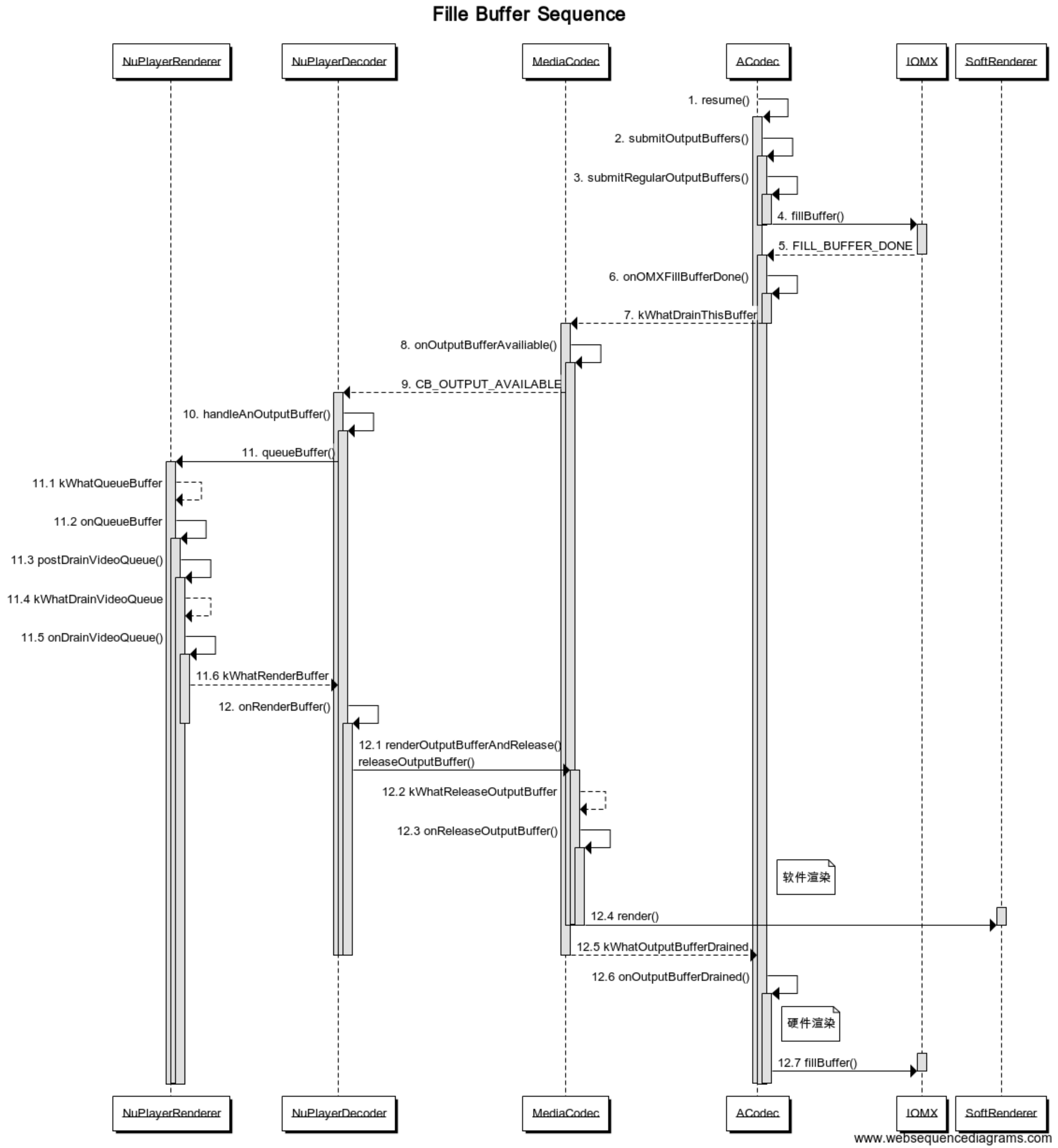
Start Sequence

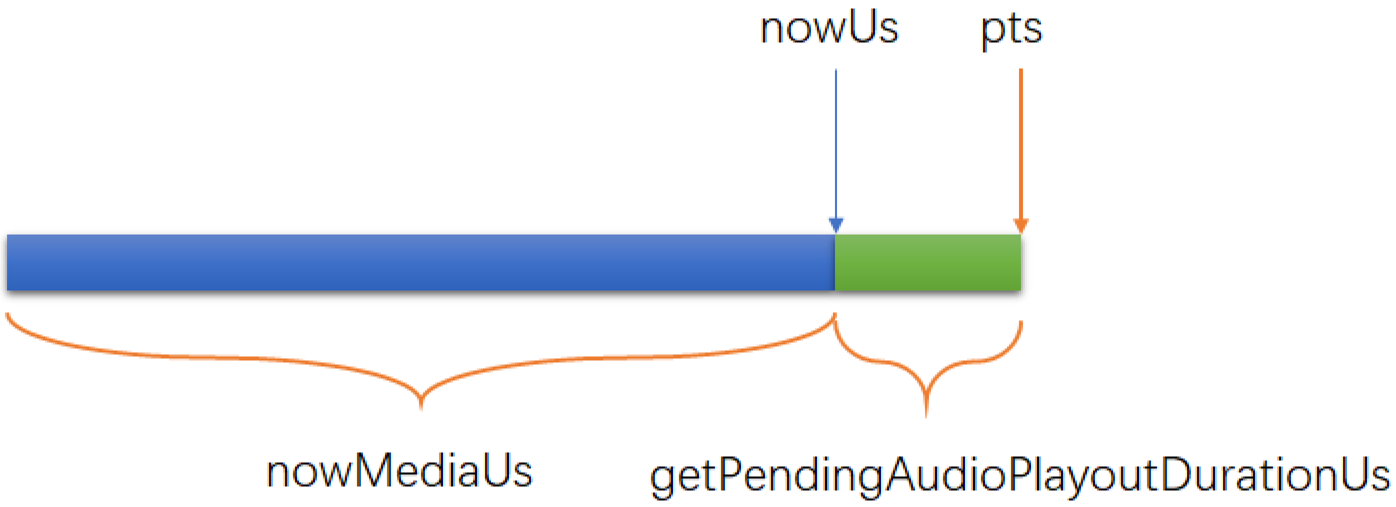


Empty Buffer Sequence



Fill Buffer Sequence



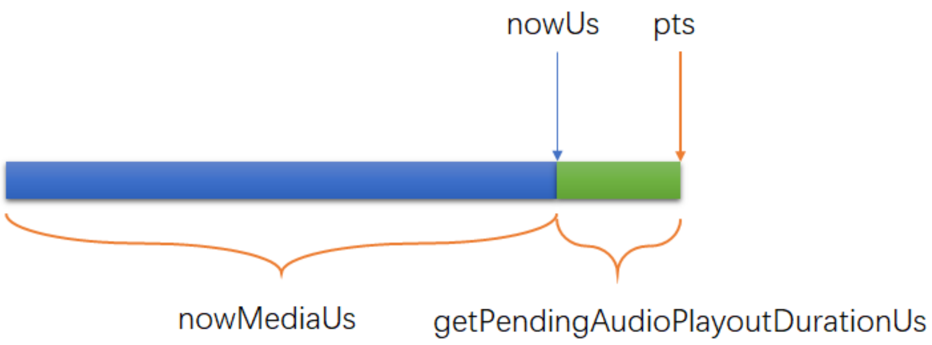


synchronization behavior is required to ensure that the deviation is within a certain range.

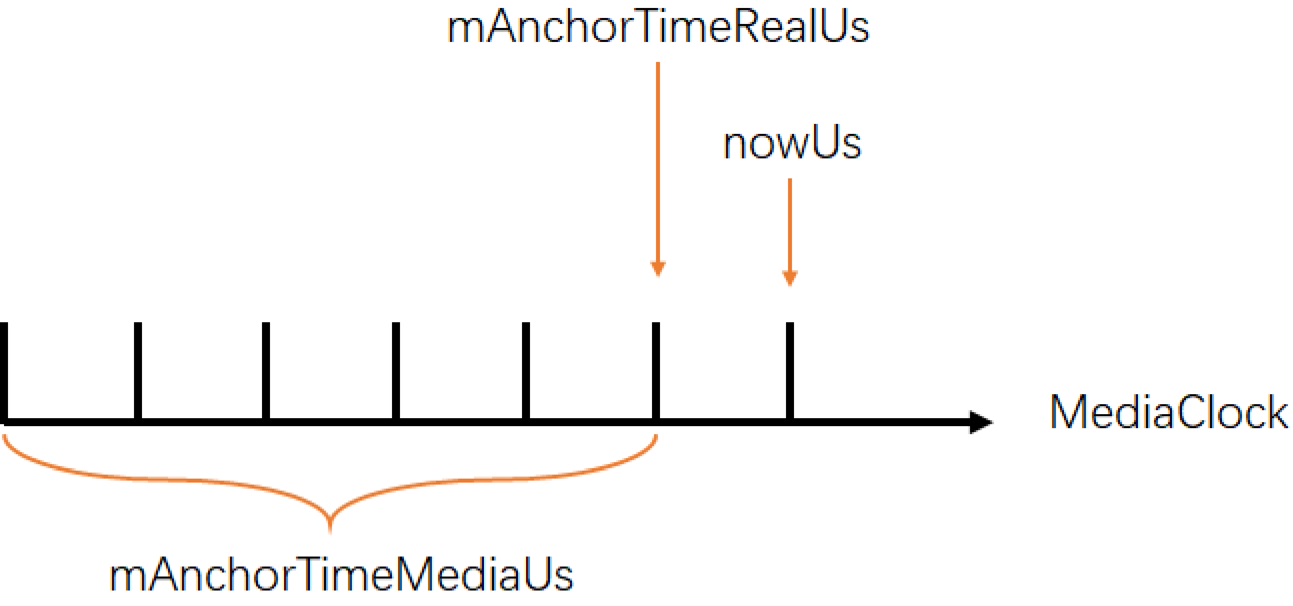
sure continuity during playback. The usual

f audio frame played, the reference external clock(MediaClock) is used to make an anchor point at intervals, and then synchronized the video frame according to the anchor point.

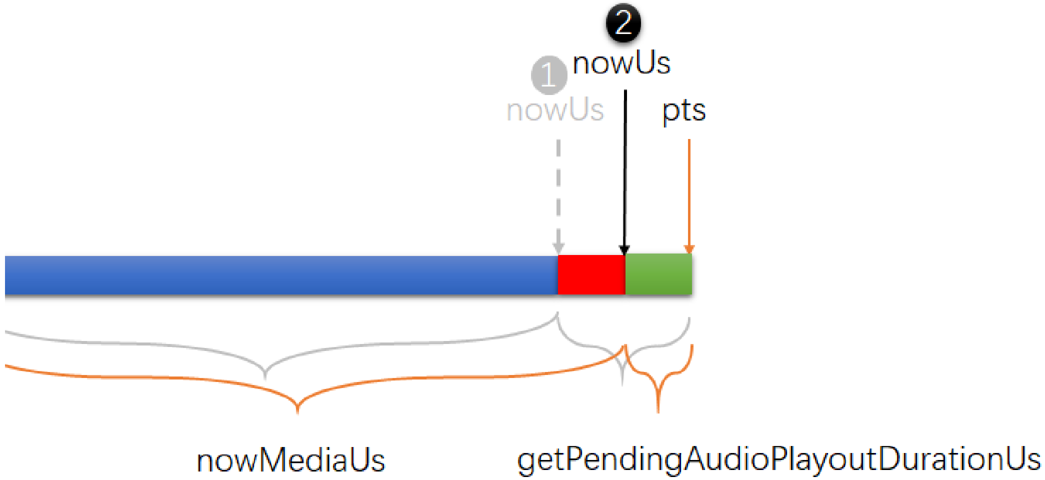
int64_t nowMediaUs = mediaTimeUs - getPendingAudioPlayoutDurationUs(nowUs);



mediaTimeUs : pts of next audio frame;
nowMediaUs : Currently played time;
nowUs : Current system time;
getPendingAudioPlayoutDurationUs : The length of time that has not been played yet



shown in red in the figure:



Audio sampling rate is fixed. At the same time, there are two parameters: one is the number of frames written to audioTrack: mNumFrameWritten, the other is the number of frames played: numplayedFramed:

```
getPendingAudioPlayoutDurationUs = mNumFrameWritten * 1000000LL / sampleRate -
mAudioSink->getPlayedOutDurationUs(nowUs);

mAudioSink->getPlayedOutDurationUs(nowUs) = ((int32_t)numFramesPlayed * 1000000LL /
mSampleRate) + (nowUs - numFramesPlayedAt);
```

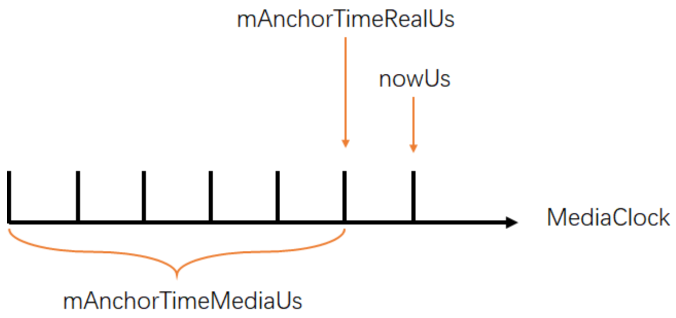
numFramesPlayedAt stands for "system time corresponding to numFramesPlayed"

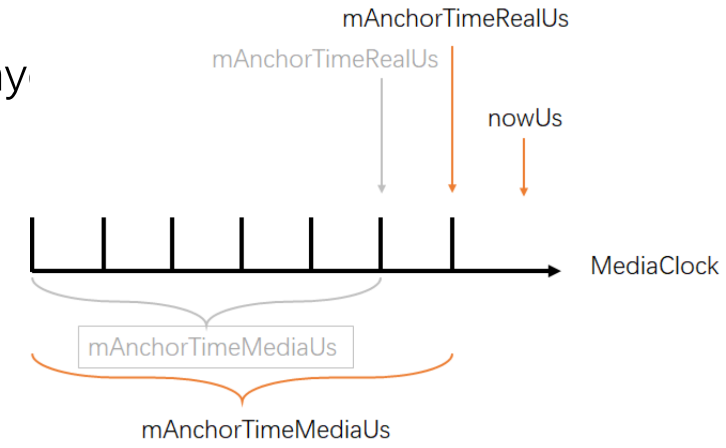
Renderer —— A/V sync - 3

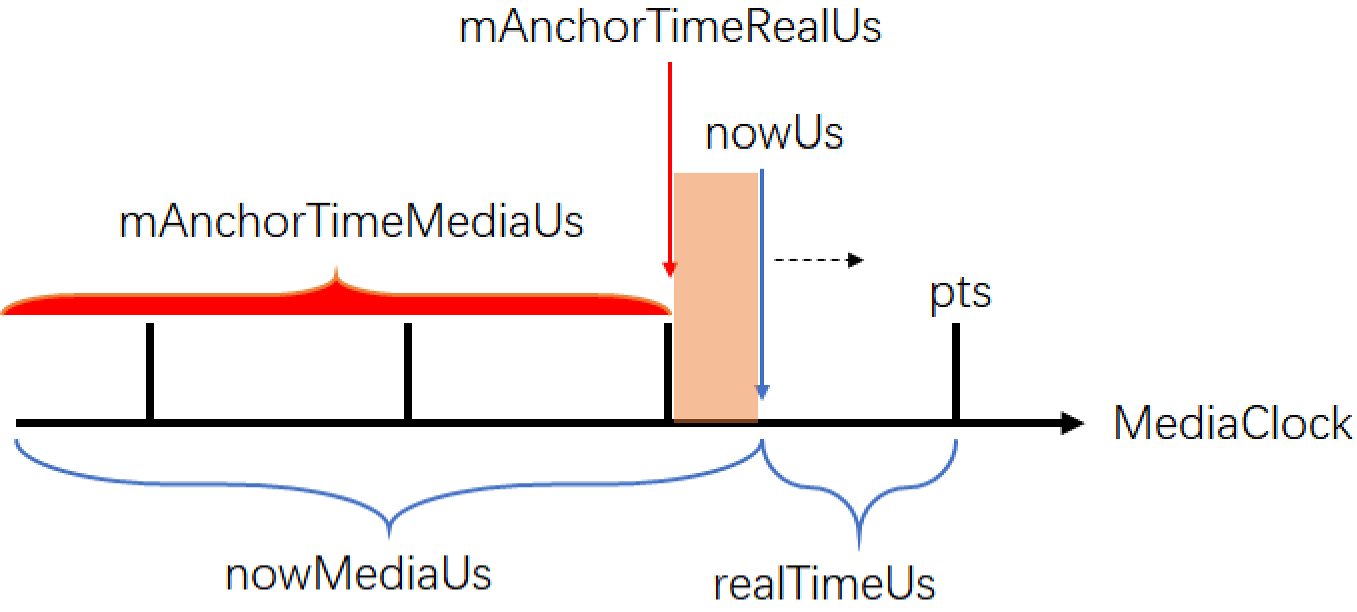
The Renderer will clear the audio frame every once in a while and update the anchor time.

The anchor time is updated as follows:
mAnchorTimeMediaUs: the time since the first frame, after syncing to MediaClock;
mAnchorTimeRealUs: current anchor timestamp under MediaClock;

MediaClock anchor is updated after each audio frame is play



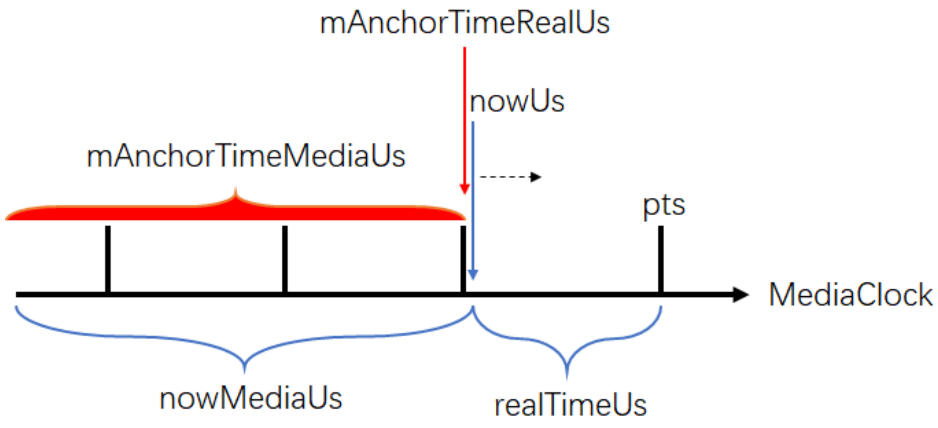


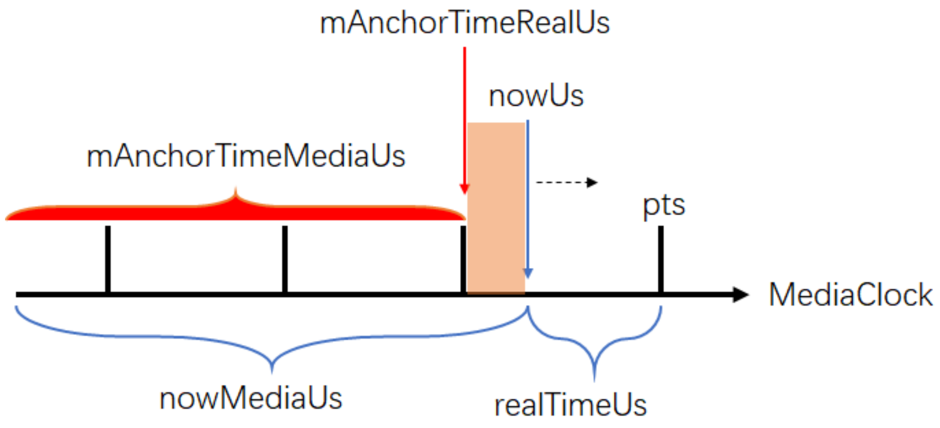


[imeRealUs)) + nowUs;

be played at this point.

Then after realTimeUs, the video frame will be displayed(Also need to consider the vsync mechanism).





OPENMAX

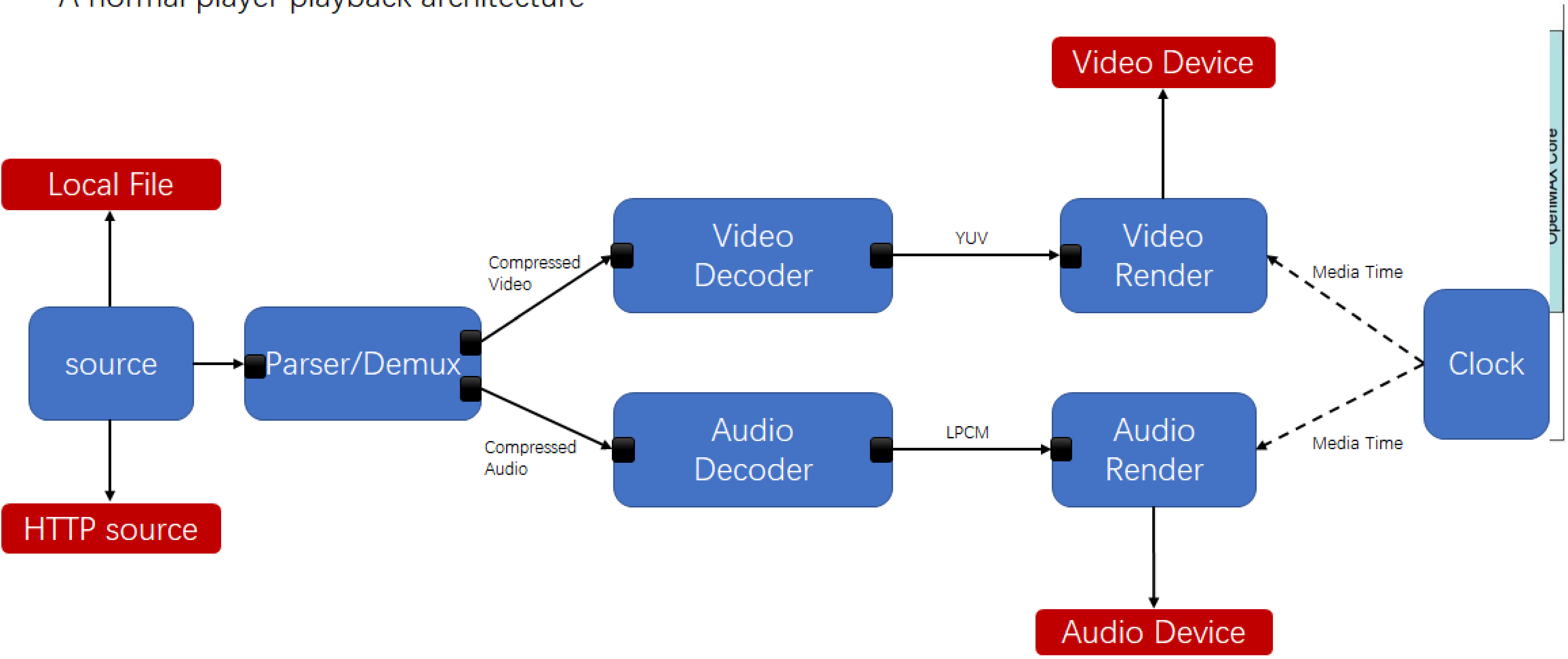
OpenMAX introduction

The OpenMAX IL API is dedicated to building an array of portable media components through the C language. These components can be sources, sinks, codecs, filters, splitters, mixers, or any other operation.

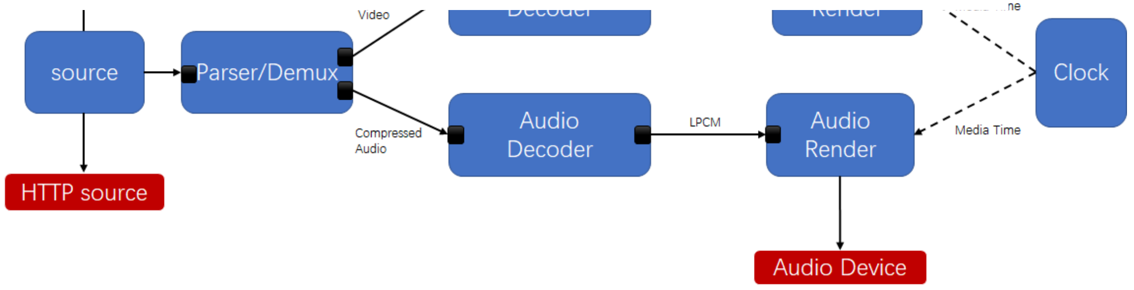
The OpenMAX IL API allows users to load, control, connect and uninstall individual components. Android's main multimedia engine, StageFright, uses OpenMax through IBinder for codec processing. According to the abstraction of OpenMAX, Android itself does not care whether the constructed Codec is hardware decoding or software decoding.

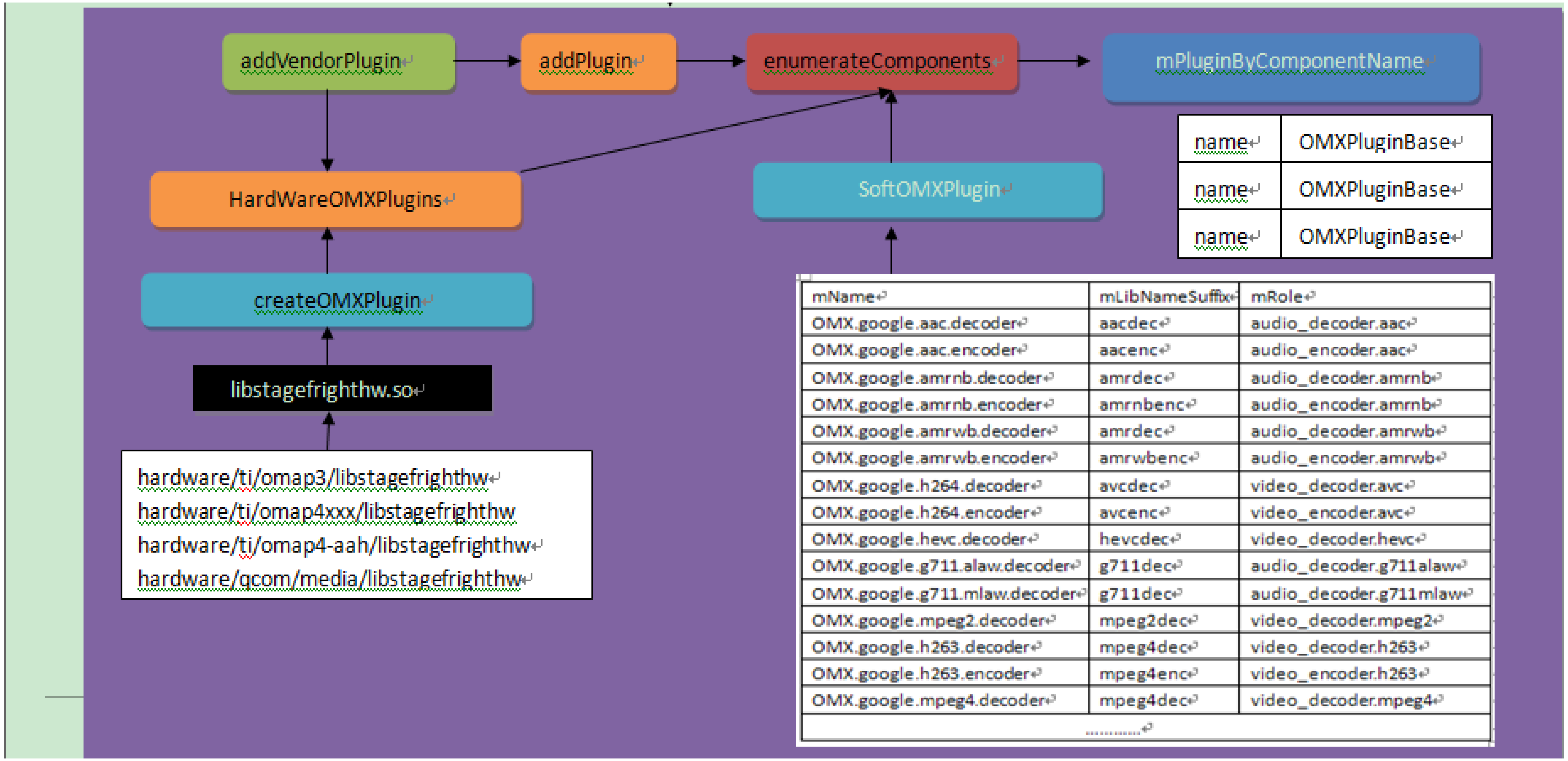
- The official website of OpenMAX is as follows:: http://www.khronos.org/openmax/

A normal player playback architecture



components are the Source component, the Host component, the Accelerator component, and the Sink component.

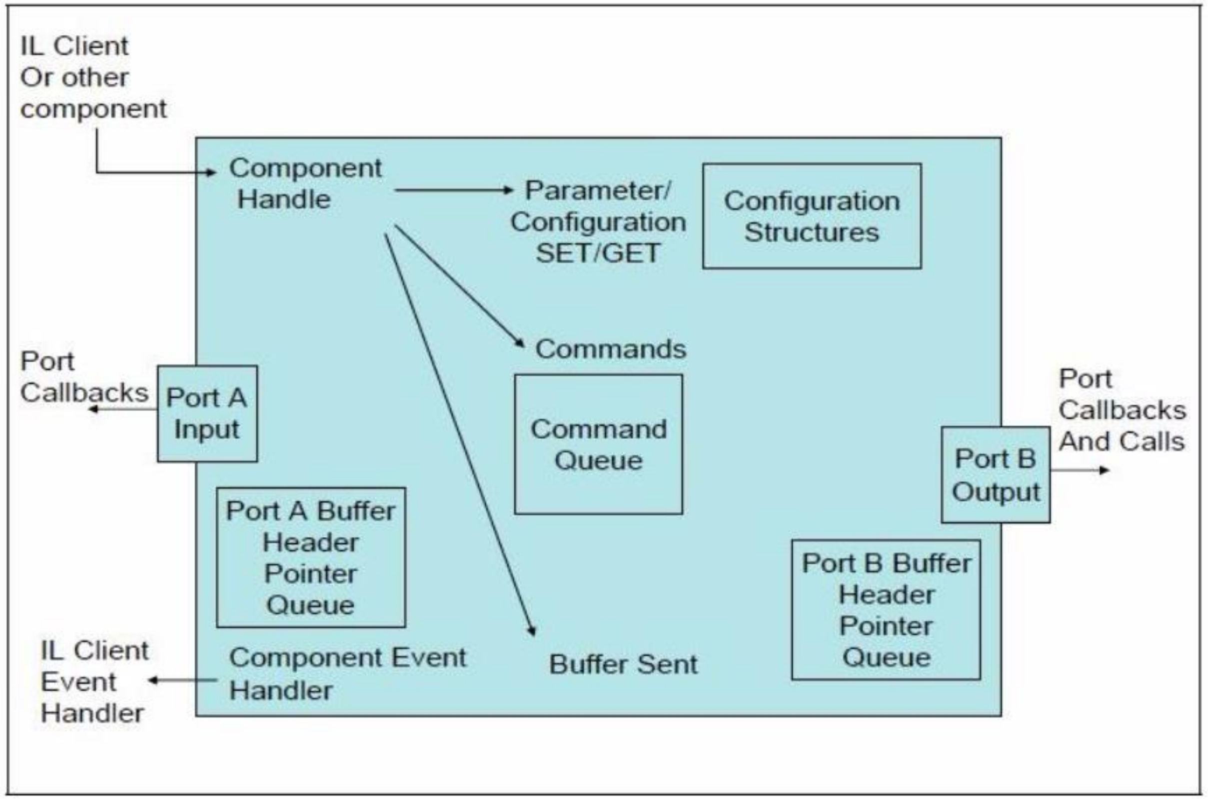



addVendorPlugin
addPlugin
enumerateComponents
mPluginByComponentName
name | OMXPluginBase
name | OMXPluginBase
name | OMXPluginBase
HardWareOMXPlugins
SoftOMXPlugin
createOMXPlugin
libstagefrighthw.so
hardware/ti/omap3/libstagefrighthw
hardware/ti/omap4xxx/libstagefrighthw
hardware/ti/omap4-aah/libstagefrighthw
hardware/qcom/media/libstagefrighthw
mName | mLibNameSuffix | mRole
OMX.google.aac.decoder | aacdec | audio_decoder.aac
OMX.google.aac.encoder | aacenc | audio_encoder.aac
OMX.google.amrnb.decoder | amrdec | audio_decoder.amrnb
OMX.google.amrnb.encoder | amrnbenc | audio_encoder.amrnb
OMX.google.amrwb.decoder | amrdec | audio_decoder.amrwb
OMX.google.amrwb.encoder | amrwbenc | audio_encoder.amrwb
OMX.google.h264.decoder | avcdec | video_decoder.avc
OMX.google.h264.encoder | avcenc | video_encoder.avc
OMX.google.hevc.decoder | hevcdec | video_decoder.hevc
OMX.google.g711.alaw.decoder | g711dec | audio_decoder.g711alaw
OMX.google.g711.mlaw.decoder | g711dec | audio_decoder.g711mlaw
OMX.google.mpeg2.decoder | mpeg2dec | video_decoder.mpeg2
OMX.google.h263.decoder | mpeg4dec | video_decoder.h263
OMX.google.h263.encoder | mpeg4enc | video_encoder.h263
OMX.google.mpeg4.decoder | mpeg4dec | video_decoder.mpeg4
…………

OMX Plugin/component

The components are the core of the OpenMax IL implementation. One component is interfaced to the input and output ports, and the port can be connected to another component. The external pair component can send commands, and also set/get parameters, configuration, and so on. The component's port can contain a queue of buffers.

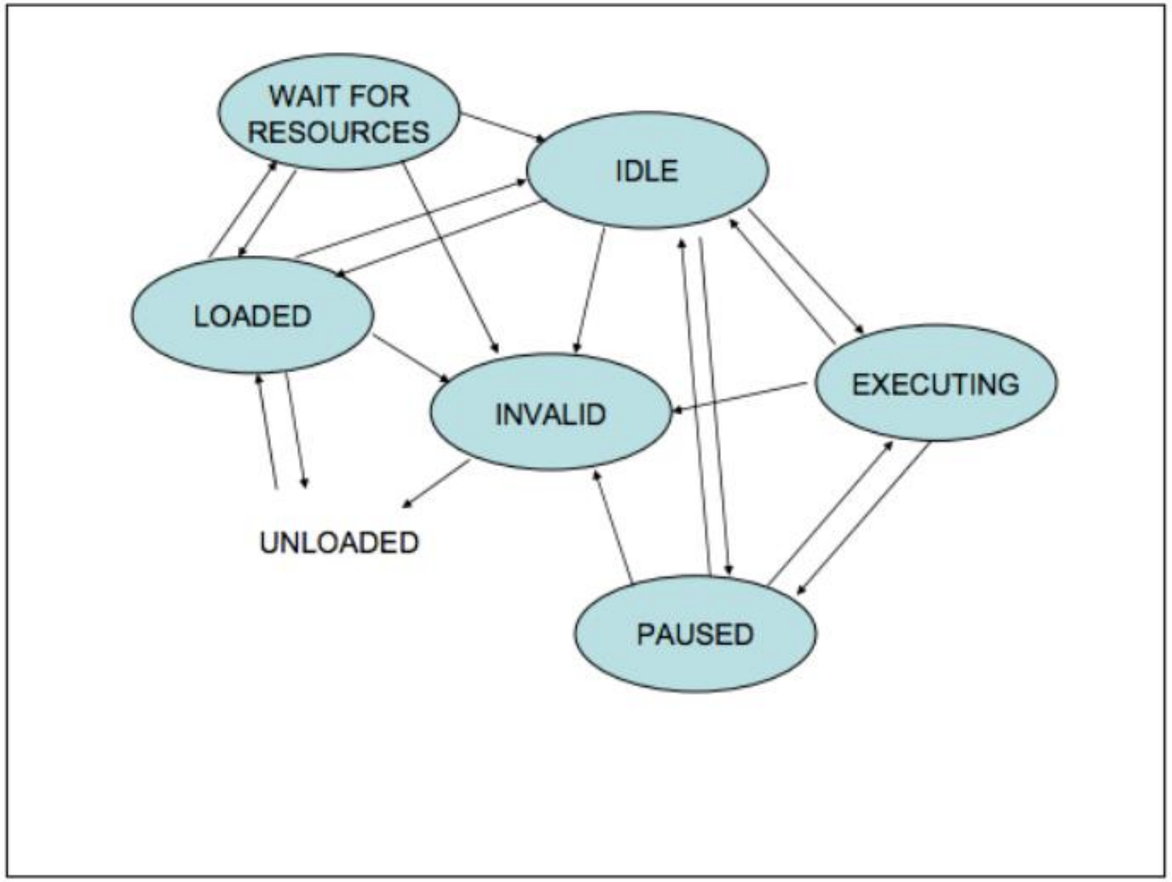
The core content of the processing of the component is: the Buffer is consumed through the input port, and the Buffer is filled through the output port, so that the multi-components can be connected to form a streaming process. The structure of a component in OpenMAX IL is shown below:



OMX Plugin State mechanism

Loaded -> Idle
1) Send the command to the OMX component via the "OMX_SendCommand" call, changing the state from OMX_StateLoaded to OMX_StateIdle
2) Call a series of "OMX_UseBuffer" or "OMX_AllocateBuffer" to notify the OMX component. These calls use NumInputBuffer to record the number of input and input ports, and NumOutputBuffer to record the number of input and input ports.
3) Wait for the EventHandler event callback of the OMX component to notify the framework state transformation completion (OMX_EventCmdComplete)

Idle -> Executing
1) Send the command to the OMX component via the "OMX_SendCommand" call, changing the state from OMX_StateIdle to OMX_StateExecuting.
2) Wait for the EventHandler event callback of the OMX component to notify the framework state transformation completion (OMX_EventCmdComplete).
3) The input buffer is sent to the OMX component via the OMX_EmptyThisBuffer call, and the output buffer is sent to the OMX component via the OMX_FillThisBuffer call, and the component returns the buffer using the appropriate callback function.

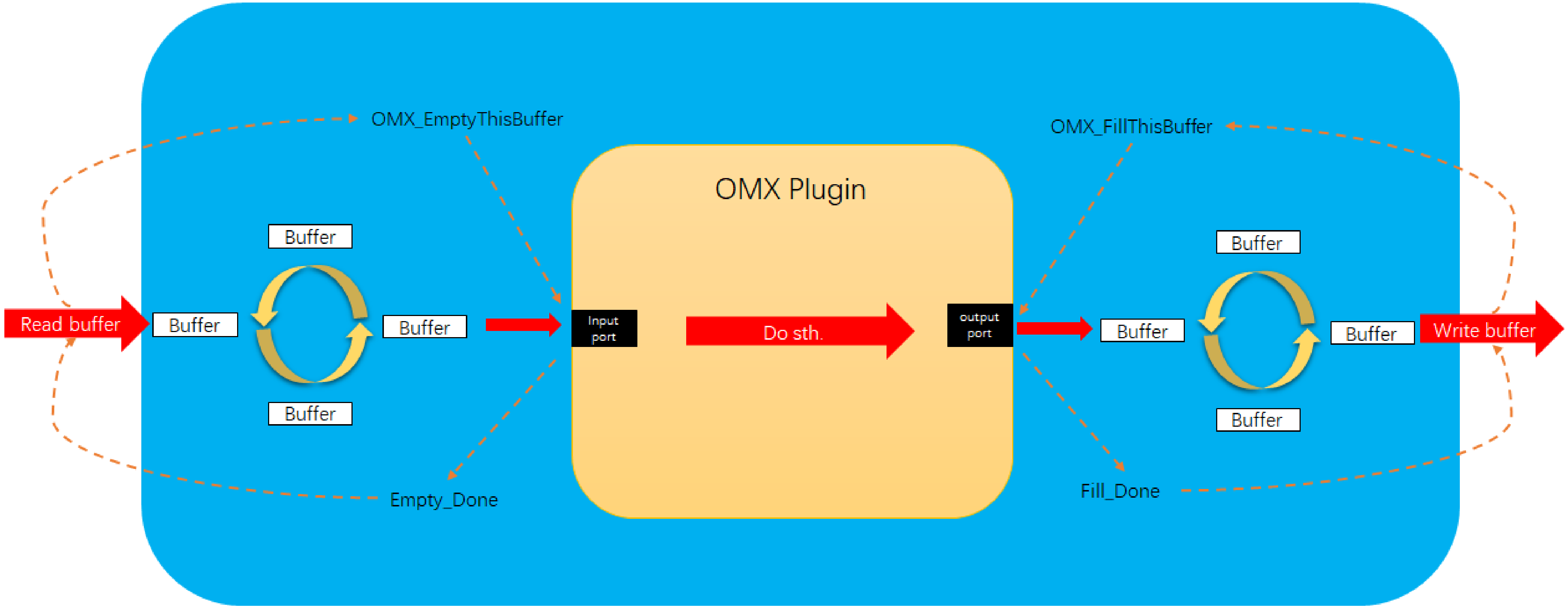


How to use an OMX plugin

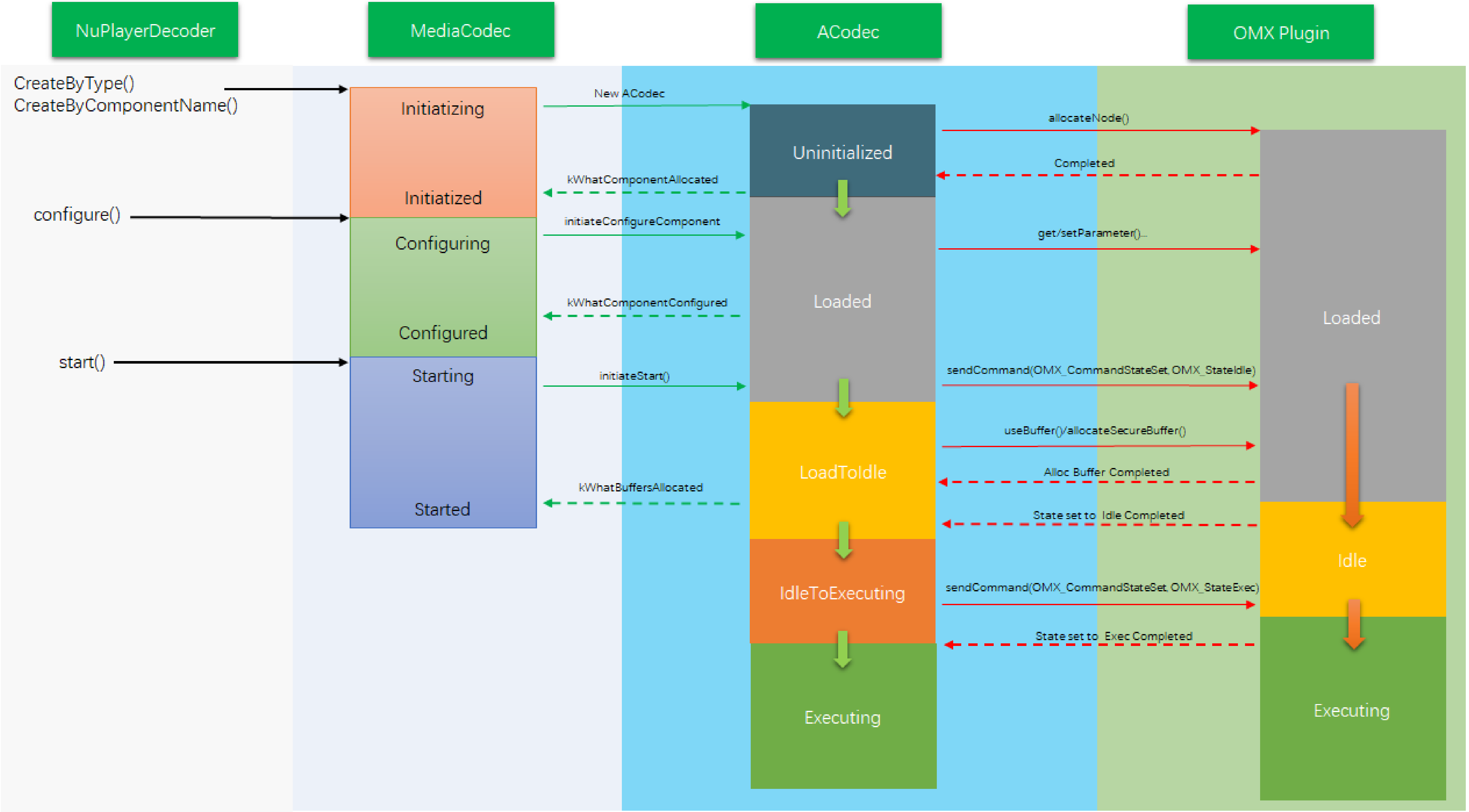
ret = OMX_GetHandle(&hComponent, hTest->name, hTest, &gCallBacks);
ret = OMX_GetParameter(hComponent, OMX_IndexParamPortDefinition, &sPortDef);
ret = SendCommand(hTest, OMX_CommandStateSet,eState,NULL, OMX_FALSE);
ret = WaitCommand(hTest, OMX_CommandStateSet, eState, NULL);

```
OMX_ERRORTYPE start_data_process(HTEST *hTest)
{
    OMX_ERRORTYPE ret = OMX_ErrorNone;
    OMX_U32 i;

    hTest->bHoldBuffers = OMX_FALSE;

    /* Send output buffers */
    for(i=0; i<hTest->nBufferHdr[1]; i++) {
        hTest->pBufferHdr[1][i]->nFilledLen = 0;
        hTest->pBufferHdr[1][i]->nOffset = 0;
        OMX_FillThisBuffer(hTest->hComponent, hTest->pBufferHdr[1][i]);
    }

    /* Send input buffers */
    for(i=0; i<hTest->nBufferHdr[0]; i++) {
        read_data(hTest, hTest->pBufferHdr[0][i]);
        OMX_EmptyThisBuffer(hTest->hComponent, hTest->pBufferHdr[0][i]);
    }

    return ret;
}
```

Omx plugin buffer sequence



State switching diagram



How to Implement an omx plugin - 1

Stagefright comes with built-in software codecs for common media formats, but you can also add your own custom hardware codecs as OpenMAX components. To do this, you must create the OMX components and an OMX plugin that hooks together your custom codecs with the Stagefright framework.

To add your own codecs:
1. Create your components according to the OpenMAX IL component standard. The component interface is located in the frameworks/native/include/media/OpenMAX/OMX_Component.h file.
2. Create a OpenMAX plugin that links your components with the Stagefright service. For the interfaces to create the plugin, see frameworks/native/include/media/hardware/OMXPluginBase.h and HardwareAPI.h header files.
3. Build your plugin as a shared library with the name libstagefrighthw.so in your product Makefile. For example:

   In your device's Makefile, ensure you declare the module as a product package:

```
LOCAL_MODULE := libstagefrighthw
PRODUCT_PACKAGES += \
  libstagefrighthw \
  ...
```

How to Implement an omx plugin - 2

Exposing codecs to the framework

The Stagefright service parses the system/etc/media_codecs.xml and system/etc/media_profiles.xml to expose the supported codecs and profiles on the device to app developers via the android.media.MediaCodecList and android.media.CamcorderProfile classes. You must create both files in the device/<company>/<device>/ directory and copy this over to the system image's system/etc directory in your device's Makefile. For example:

```
PRODUCT_COPY_FILES += \
device/fsl-proprietary/media-profile/media_profiles.xml
device/fsl-proprietary/media-profile/media_codecs.xml
device/fsl-proprietary/media-profile/media_codecs_vpu.xml
device/fsl-proprietary/media-profile/media_codecs_libav.xml
```

How to Implement an omx plugin - 3

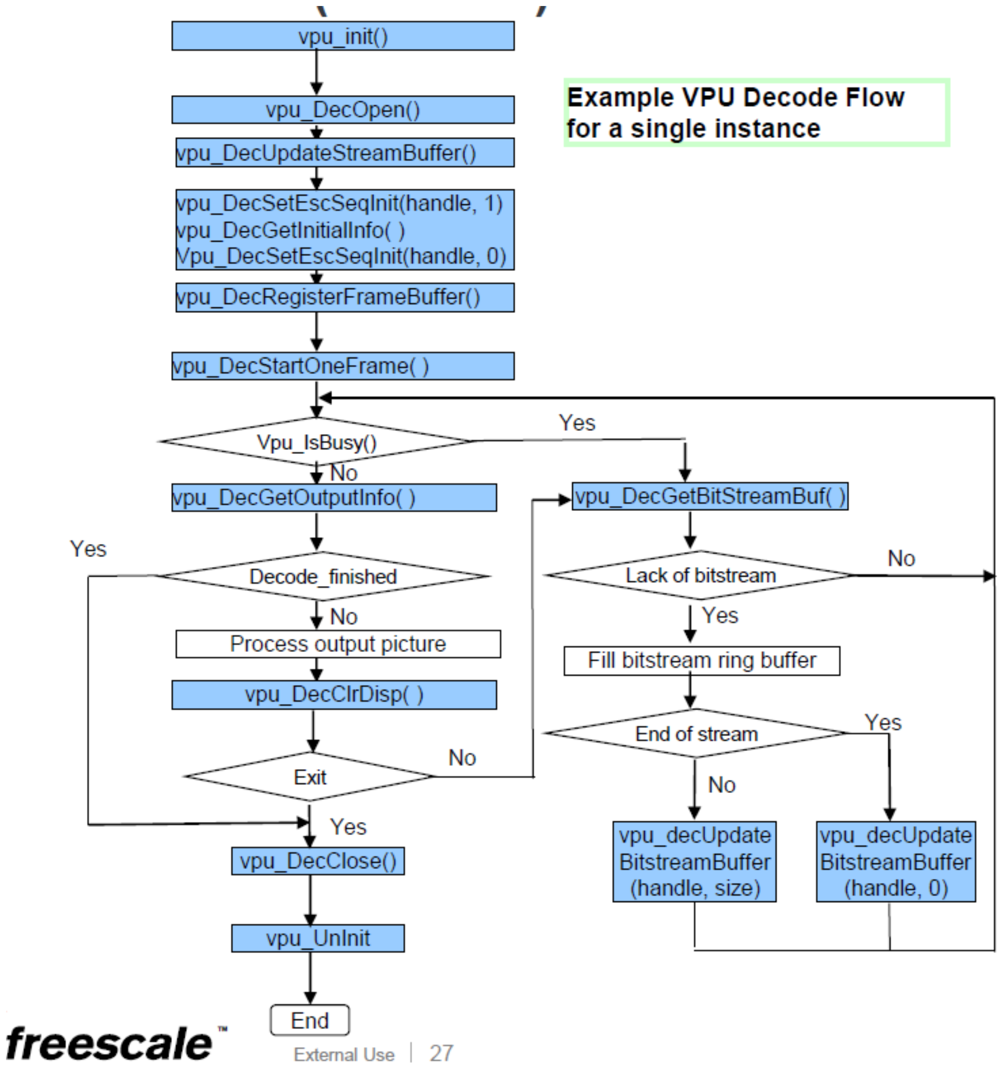
Add Entry function in external/fsl_imx_omx/OpenMAXIL/release/registry/component_register

```
@
component_name=OMX.Freescale.std.video_decoder.avc.v3.hw-based;
library_path=lib_omx_vpu_dec_v2_arm11_elinux.so;
component_entry_function=VpuDecoderInit;
component_role=video_decoder.avc;
role_priority=3;
$

@
component_name=OMX.Freescale.std.video_decoder.avc.sw-based;
library_path=lib_omx_libav_video_dec_arm11_elinux.so;
component_entry_function=LibavVideoDecoderInit;
component_role=video_decoder.avc;
role_priority=2;
$

@
component_name=OMX.Freescale.std.video_decoder.soft_hevc.sw-based;
library_path=lib_omx_soft_hevc_dec_arm11_elinux.so;
component_entry_function=SoftHevcDecoderInit;
component_role=video_decoder.hevc;
role_priority=3;
$

```

```
extern "C"
{
    OMX_ERRORTYPE VpuDecoderInit(OMX_IN OMX_HANDLETYPE pHandle)
    {
        OMX_ERRORTYPE ret = OMX_ErrorNone;
        VpuDecoder *obj = NULL;
        ComponentBase *base = NULL;
        VPU_COMP_API_LOG("%s: \r\n",__FUNCTION__);

        obj = FSL_NEW(VpuDecoder, ());
        if(obj == NULL)
        {
            VPU_COMP_ERR_LOG("%s: vpu decoder new failure: ret=0x%X \r\n",__FUNCTION__,ret);
            return OMX_ErrorInsufficientResources;
        }

        base = (ComponentBase*)obj;
        ret = base->ConstructComponent(pHandle);
        if(ret != OMX_ErrorNone)
        {
            VPU_COMP_ERR_LOG("%s: vpu decoder construct failure: ret=0x%X \r\n",__FUNCTION__,ret);
            return ret;
        }
        return ret;
    }
}
```

How to Implement an omx plugin – 4

- Integrate these hardware decoding code into the omx framework:



vpu_Init()
vpu_Deinit()
vpu_IsBusy(), jpu_IsBusy()
vpu_SleepWake()
vpu_GetVersionInfo();
vpu_WaitForInt()
vpu_SWReset()

IOGetPhyMem(),IOFreePhyMem()
IOGetVirtMem(), IOFreeVirtMem()
IOGetIramBase()

vpu_DecOpen();
vpu_DecClose();
vpu_DecSetEscSeqInit();
vpu_DecGetInitialInfo();
vpu_DecRegisterFrameBuffer();
vpu_DecGetBitstreamBuffer();
vpu_DecUpdateBitstreamBuffer();
vpu_DecStartOneFrame();
vpu_DecGetOutputInfo();
vpu_DecBitBufferFlush();
vpu_DecGiveCommand()
vpu_DecClrDispFlag()

THANKS