Rockchip Android Multimedia FAQ

ID: RK-PC-YF-E07

Release Version: V1.0.3

Release Date: 2025-02-07

Security Level: □Secret □Internal □Public ■Public

Rockchip Electronics Co., Ltd.

No.18 Building, A District, No.89, software Boulevard Fuzhou, Fujian,PRC

Website: www.rock-chips.com

Customer service Tel: +86-4007-700-590

Customer service Fax: +86-591-83951833

Customer service e-Mail: fae@rock-chips.com

Preface

Overview

This document primarily introduces common debugging methods for the Rockchip multimedia platform and frequently encountered issues during the development process. Engineers encountering issues related to those covered in this document are advised to attempt debugging using the provided methods, converge on the problem, enhance troubleshooting efficiency, and proceed to resolve the issues.

Chip Name	Kernel Version
Support all chipsets	Linux-4.19, Linux-5.10

Intended Audience

This document (this guide) is mainly intended for:

Technical Support Engineer

Software Development Engineer

Revision History

Version	Author	Date	Change Description
V1.0.0	Chen Jinsen	2022-03-03	Initial version release
V1.0.1	Chen Jinsen	2023-06-04	Updated format, added common issues
V1.0.2	Chen Jinsen	2023-03-19	Added support for RK3576
V1.0.3	Chen Jin Sen	2025-02-07	Added some common issues

Contents

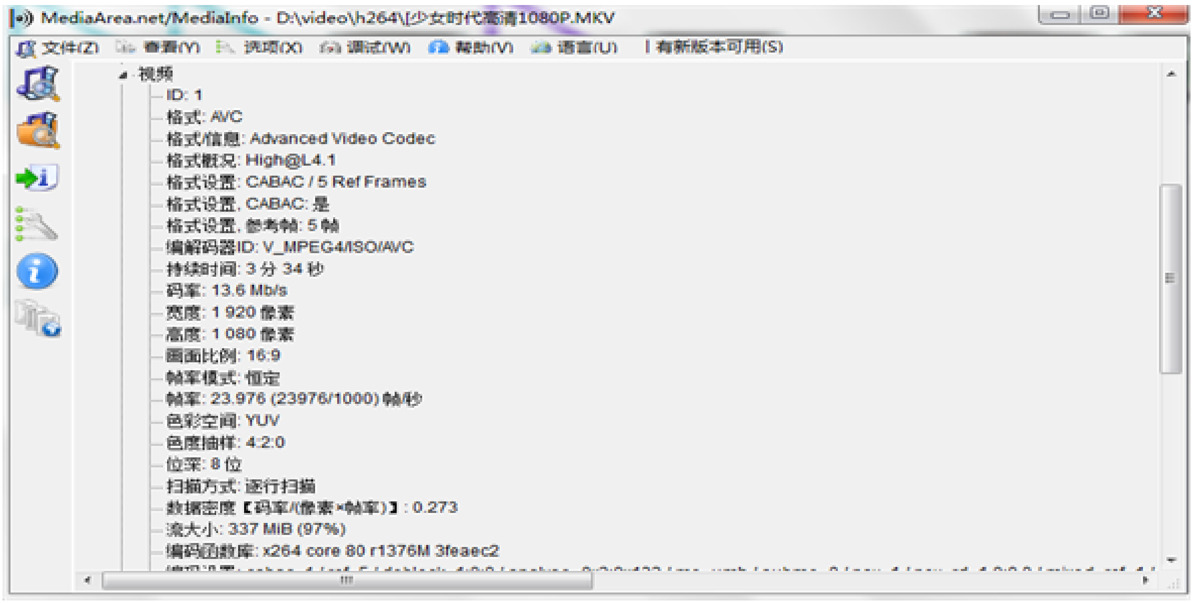
1. General Media Category

1.1 Media Source Unable to Play

1.1.1 Check if the media sources are within the supported specifications of the chip

Using the MediaInfo tool provided by Windows or Linux allows querying the media parameters of the source material, including encoding format, resolution, bitrate, scan method, bit depth, and other basic information. Comparing with the platform-provided chip datasheet manuals or Codec Benchmark can preliminarily determine if the source material is supported.

> Note: Datesheet manuals and Codec Benchmark calibrated chip codec capabilities are essential prerequisites for troubleshooting chip support issues. To obtain these documents, please submit a request via email (sw.fae@rock-chips.com).


MediaArea.net/MediaInfo - D:\video\h264\[少女时代高清1080P.MKV
文件(Z)
查看(Y)
选项(X)
调试(W)
帮助(V)
语言(U)
有新版本可用(S)

```
视频
  ID: 1
  格式: AVC
  格式/信息: Advanced Video Codec
  格式概况: High@L4.1
  格式设置: CABAC / 5 Ref Frames
  格式设置, CABAC: 是
  格式设置, 参考帧: 5 帧
  编解码器ID: V_MPEG4/ISO/AVC
  持续时间: 3 分 34 秒
  码率: 13.6 Mb/s
  宽度: 1 920 像素
  高度: 1 080 像素
  画面比例: 16:9
  帧率模式: 恒定
  帧率: 23.976 (23976/1000) 帧/秒
  色彩空间: YUV
  色度抽样: 4:2:0
  位深: 8 位
  扫描方式: 逐行扫描
  数据密度【码率/(像素×帧率)】: 0.273
  流大小: 337 MiB (97%)
  编码函数库: x264 core 80 r1376M 3feaec2
```

1.1.2 Check Whether Audio/Video Formats Involve Copyright Issues

The audio/video formats that the platform is explicitly known to not support due to copyright reasons include:

```
Audio: mlp, ac3, eac3, dts, dsp, heaac, other Dolby-related
Video: DivX, Xvid, RMVB, VP6, VC-1, SVQ, ISO Blue-ray
```

1.2 Common Chip Codec Capabilities Specification Table

The codec specifications for chips can be queried and obtained in Datasheet or Codec Benchmark. The following section includes the calibrated specification tables for common chip codec capabilities on the platform, facilitating quick query and confirmation.

> The actual measured encoding and decoding capabilities are correlated with the system load at the time of testing. For instance, when the CPU\DDR load is relatively high, the encoding and decoding capabilities may slightly fall below the specifications stated in the Datasheet.

1.2.1 Decoding Capability Specification Table

	H264	H265	VP9	JPEG
RK3588	7680x4320@30f	7680x4320@60f	7680x4320@60f	1920x1080@280f
RK3576	4096x2304@60f	7680x4320@30f	7680x4320@30f	3840x2160@90f
RK3562	1920x1080@60f	4096x2304@30f	4096x2304@30f	1920x1080@60f
RK3528	4096x2304@30f	4096x2304@60f	4096x2304@60f	1920x1080@120f
RK356X	4096x2304@30f	4096x2304@60f	4096x2304@60f	1920x1080@80f
RK3399	4096x2304@30f	4096x2304@60f	4096x2304@60f	1920x1080@30f
RK3328	4096x2304@30f	4096x2304@60f	4096x2304@60f	1920x1080@30f
RK3288	3840x2160@30f	4096x2304@60f	N/A	1920x1080@30f
RK3368/PX5	4096x2160@25f	4096x2304@60f	N/A	1920x1080@30f
RK3326/PX30	1920x1080@60f	1920x1080@60f	N/A	1920x1080@30f
RK312X	1920x1080@60f	1920x1080@60f	N/A	1920x1080@30f

Other points to note:

1. RK3588 supports AVS2(7680x4320@60f) and AV1(3840x2160@60f) decoding.
2. RK3528 supports AVS2 (4096x2160@60f) decoding.
3. RK3562 does not support decoding formats such as mpeg1/2/4, vp8, h263, etc.
4. Except for RK3562, other chips support MPEG1/2/4, VP8, H263 decoding, with maximum specification of 1080P.

1.2.2 Encoding Capability Specification Table

	H264	H265	VP8	JPEG
RK3588	7680x4320@30f	7680x4320@30f	1920x1080@30f	3840x2160@30f
RK3576	3840x2160@60f	3840x2160@60f	N/A	3840x2160@160f
RK3562	1920x1080@30f	N/A	N/A	N/A
RK3528	1920x1080@60f	1920x1080@60f	N/A	3840x2160@30f
RK356X	1920x1080@60f	1920x1080@60f	1920x1080@30f	1920x1080@60f
RK3399	1920x1080@30f	N/A	1920x1080@30f	1920x1080@30f
RK3328	1920x1080@30f	1920x1080@30f	1920x1080@30f	1920x1080@30f
RK3288	1920x1080@30f	N/A	1920x1080@30f	1920x1080@30f
RK3368/PX5	1920x1080@30f	N/A	1920x1080@30f	1920x1080@30f
RK3326/PX30	1920x1080@30f	N/A	1920x1080@30f	1920x1080@30f
RK312X	1920x1080@30f	N/A	1920x1080@30f	1920x1080@30f

1.3 How to Increase VPU Frequency

VPU (Video Processing Unit) is a hardware video processing unit. Evaluating the hardware codec performance issues often requires adjusting the VPU frequency.

> It is generally considered that when the performance reaches a bottleneck, increasing the frequency of VPU and DDR can improve the hardware's codec capabilities. However, overloaded frequencies may impact the system stability of the device. It is recommended that customers conduct thorough testing prior to integration.

To increase DDR frequency please refer to the documentation under SDK RKDocs/common/DDR/Rockchip-Developer-Guide-DDR/ directory. This section primarily introduces VPU frequency operations based on the platform.

1.3.1 VPU Frequency Query

The VPU frequency can be queried through the system clock tree table (cat /d/clk/clk_summary). The following lists the clock tree frequency node names for common codecs, which can be cross-referenced in the clk_summary output.

	h264_dec	h265_dec	vp9_dec	jpeg_dec	h264_enc	h265_enc
RK3588	rkvdec	rkvdec	rkvdec	jpeg_decoder	rkvenc	rkvenc
RK3576	rkvdec	rkvdec	rkvdec	jpeg	vepu	vepu
RK3562	rkvdec	rkvdec	rkvdec	jdec	rkvenc	×
RK3528	rkvdec	rkvdec	rkvdec	jpeg_decoder	rkvenc	rkvenc
RK356X	rkvdec	rkvdec	rkvdec	jdec	rkvenc	rkvenc
RK3399	vdu	vdu	vdu	vcodec	vcodec	×
RK3328	rkvdec	rkvdec	rkvdec	vpu	h264	h265
RK3288	vcodec	hevc	×	×	vcodec	×
RK3368/PX5	video	video	×	video	video	×
RK3326/PX30	vpu	vpu	×	vpu	vpu	×
RK312X	vdpu	vdpu	×	vdpu	vdpu	×

Examples - Required frequency for the following scenarios:

1. RK3588 H264 Decoding

```
cat /d/clk/clk_summary | grep rkvdec     // <aclk_rkvdec0> <aclk_rkvdec1>
```

2. RK3288 h264 Decoding

```
cat /d/clk/clk_summary | grep vcodec  // <aclk_vcodec>
```

3. RK3328 H.265 Encoding

```
cat /d/clk/clk_summary | grep h265    // <aclk_h265>
```

1.3.2 VPU Frequency Modification

4.4 Kernel (Android 7.1 ~ 9.0) frequency modification can be referenced as follows: configure VPU frequency to run at 500MHz for testing. The 4.4 kernel driver version does not support to configure individual IP frequency.

```
--- a/drivers/video/rockchip/vcodec/vcodec_service.c
+++ b/drivers/video/rockchip/vcodec/vcodec_service.c
@@ -2307,6 +2307,7 @@ static void vcodec_set_freq_default(struct vpu_service_info
*pservice,
 {
        enum VPU_FREQ curr = atomic_read(&pservice->freq_status);

+       reg->freq = VPU_FREQ_500M;
        if (curr == reg->freq)
                return;
```

4.19/5.10 kernel (>=Android 10.0) frequency configuration can be referenced as follows: configure the rkvdec frequency to 500M for testing.

1. mpp_service driver uses dtsi format to configure frequency information

The chip codec configuration can be queried in the dtsi file. The following example is to configure the RK3399 decoder to boost the frequency to 500M when decoding resolutions over 4K.

```
<rk3399.dtsi>

rkvdec: rkvdec@ff660000{
    clock-names = "aclk_vcodec", "hclk_vcodec",
            "clk_cabac", "clk_core",;
    rockchip,normal-rates = <297000000>, <0>,
            <297000000>, <297000000>, ;
    rockchip,advanced-rates = <500000000>, <0>,
            <500000000>, <500000000>;
    rockchip,default-max-load = <2088960>;  // 1920x1088
};

rockchip,normal-rates is the clock rate set when the resolution is less than
1920x1088.
rockchip,advanced-rates is the clock rate set when the resolution exceeds
1920x1088.
```

2. Using debugfs node to modify frequency

For temporary testing to evaluate the impact of frequency on the codec, the following approach can be used:

```
echo 500000000 > /proc/mpp_service/rkvdec/aclk
echo 500000000 > /proc/mpp_service/rkvdec/clk_core
echo 500000000 > /proc/mpp_service/rkvdec/clk_cabac
```

1.4 How to Capture Codec Input and Output

For flowering screen, green screen, and similar issues, capturing the input and output of encoding and decoding facilitates rapid problem localization and narrowing down the problem scope.

The following introduces the switch for saving decoding input and output on the platform.

[Application Layer]

```
1. Using the MediaCodec API
    a) Encoding Status
       - Input: queueInputBuffer Save file before input buffer
       - Output: After dequeueOutputBuffer, the encoded output buffer can be read
and written.
    b) Decoding Status
       - Input: queueInputBuffer Save file before input buffer
       - Output: When the Surface is unconfigured, the decoded output buffer can
be read and written after dequeueOutputBuffer.
```

[Framework Layer]

```
setenforce 0
mkdir /data/video/
```

```
1) Android 12 and above versions use the Codec2 framework. Follow the commands
below to capture.

setprop vendor.dump.c2.log 0x000000f0

2) Android 11 and earlier versions use the OMX framework. Execute the following
command(s) to capture.

// Decode dec_in*.bin
setprop vendor.omx.vdec.debug 0x01000000
setprop record_omx_dec_in 1

// Encode enc_in*.bin enc_out*.bin
setprop vendor.omx.venc.debug 0x03000000
setprop record_omx_enc_in 1
setprop record_omx_enc_out 1
```

If the above command fails to generate files in the /data/video/ path, you can use the switch of the underlying system codec library to capture.

```
setenforce 0
mkdir /data/video/

setprop mpp_dump_in /data/video/mpp_dec_in.bin
setprop mpp_dump_out /data/video/mpp_dec_out.bin
setprop vendor.mpp_dump_in /data/video/mpp_dec_in.bin
setprop vendor.mpp_dump_out /data/video/mpp_dec_out.bin
setprop mpp_debug 0x600 && setprop vendor.mpp_debug 0x600
```

1.5 Multi-channel Encoding and Decoding Supported Channel Calculation

The calculation of the maximum supported channel count for the chip hardware codec involves hardware pixel computing capability, illustrated with a specific example:

```
Question: What is the maximum number of H264 1080P@30fps decoding streams
supported by RK3399?

According to the RK3399 specifications, the H.264 decoding capability is:
4096x2304@30fps.
1) Hardware pixer computing capability: 4096x2304x30f per second
2) Conversion of 1080P@30fps, calculation method:
                 (4096x2304x30) / (1920x1088x30) = 4.5
3) The greater the number of channels, the larger the calculated loss. Generally,
the final value is rounded down to the nearest integer, hence supporting 4
channels of 1080P@30fps
```

Other codecs may refer to similar calculations. It should be noted that the above calculations are based on high-bitrate extreme source materials, thus ensuring support for H264 4-channel 1080P@30fps decoding under any circumstances.

RK3588, RK356X, RK3399, RK3328 built with high-performance decoder rkvdec (H264, H265, VP9), decoding H264\H265\VP9, under extreme conditions, ordinary test source materials have the opportunity to exceed the computing limits.

As **the RK3399 may have the potential to achieve 1080P@30fps 8-channel decoding**, the specific evaluation methods are as follows:

Prerequisite: Non-high bitrate video source

1) Decoding Performance Metrics: VPU-driven kernel single-frame decoding time

```
4.19/5.10 Kernel (Android 10.0 and above versions)
$ echo 0x0100 > /sys/module/rk_vcodec/parameters/mpp_dev_debug
$ cat /proc/kmsg

4.4 Kernel (Android 7.1 to 9.0 versions)
$ echo 0x0100 > /sys/module/rk_vcodec/parameters/debug
$ cat /proc/kmsg
```

The required single-frame decoding time for supporting 8-channel 1080P@30fps is (time / total frames):

```
   ->  (1 x 1000) / (8 * 30 ) ≈ 4.16 ms
```

Ignoring time overhead from the decoding pipeline, a decoding time within 4ms per frame meets the requirement. If current testing shows the single-frame decoding time does not meet the requirement, evaluate and improve the frequency-related parameters as follows: increasing VPU frequency or DDR frequency typically provides some performance improvement for the hardware codec.

2) Frequency Information

It is commonly believed that increasing the frequency of VPU and DDR can improve the hardware codec capabilities when the performance reaches a bottleneck. Therefore, check the VPU frequency and DDR frequency during testing. If the frequencies still have room for improvement, it is recommended to attempt increasing the frequencies and then proceed to step 2 to check whether the kernel decoding time meets the requirements.

```
/* VPU Frequency */
$ cat /d/clk/clk_summary | grep vdu     <aclk_vdu>    rk3399
$ cat /d/clk/clk_summary | grep rkvdec  <aclk_rkvdec> rk3588\rk3328\rk356x

/* DDR Frequency */
$ cat /sys/class/devfreq/dmc/cur_freq
```

3) Increasing VPU Frequency

Refer to the introduction in Section 1.3 to increase the VPU frequency.

4) Increasing DDR Frequency

```
echo performance > /sys/class/devfreq/dmc/governor   // Set DDR frequency to
performance
```

For specific instructions on increasing the DDR frequency, refer to the documentation in RKDocs/common/DDR/Rockchip-Developer-Guide-DDR/.

1.6 High CPU Usage in Multimedia Applications

[Problem Description]

The customer's customized multimedia codecs application experiences high overall CPU usage during runtime, leading to system unresponsiveness.

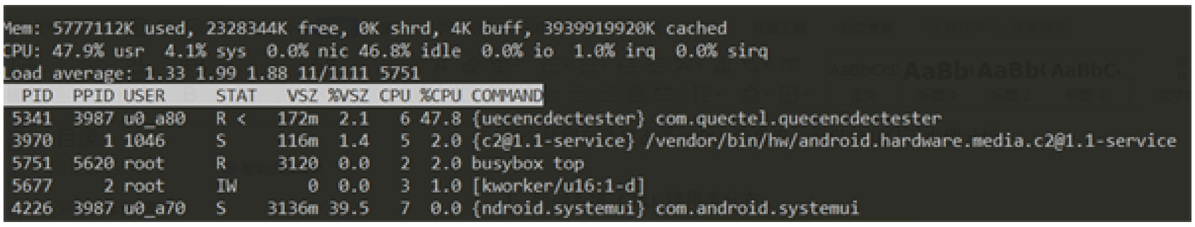
[Problem Analysis]

Media applications involve extensive read/write operations that also consume CPU resources. Therefore, the first step is to decouple responsibilities, requiring a breakdown to identify exactly which process and which thread is abnormally occupying CPU. The following steps can be referenced for analysis, progressively narrowing the scope from **process->thread->function** step by step.

1. Determine the distribution of system CPU usage

In problem scenarios, use the system **adb shell busybox top** to real-time display the CPU usage of each process.

As shown in the figure below, the application process com.quectel.quecencdectester occupies 47.8% of CPU usage and is the process requiring focused analysis.

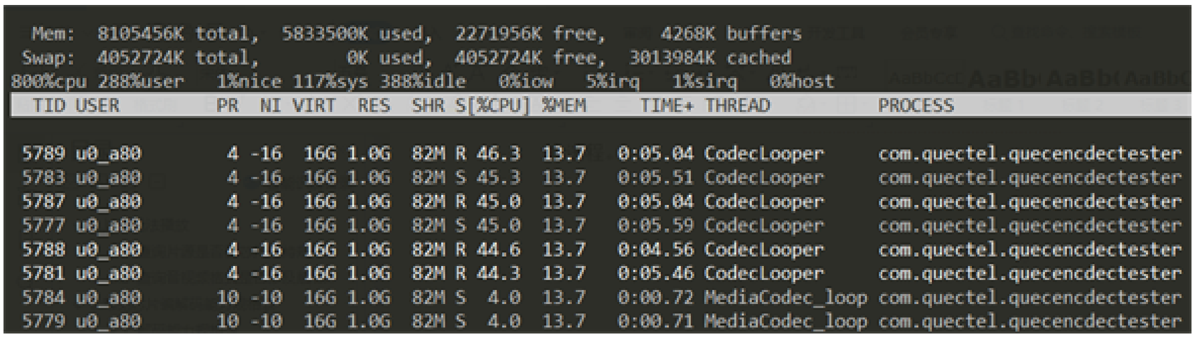


```
Mem: 5777112K used, 2328344K free, 0K shrd, 4K buff, 3939919920K cached
CPU: 47.9% usr  4.1% sys  0.0% nic 46.8% idle  0.0% io  1.0% irq  0.0% sirq
Load average: 1.33 1.99 1.88 11/1111 5751
  PID  PPID USER     STAT   VSZ %VSZ CPU %CPU COMMAND
 5341  3987 u0_a80   R <   172m  2.1   6 47.8 {uecencdectester} com.quectel.quecencdectester
 3970     1 1046     S     116m  1.4   5  2.0 {c2@1.1-service} /vendor/bin/hw/android.hardware.media.c2@1.1-service
 5751  5620 root     R     3120  0.0   2  2.0 busybox top
 5677     2 root     IW       0  0.0   3  1.0 [kworker/u16:1-d]
 4226  3987 u0_a70   S    3136m 39.5   7  0.0 {ndroid.systemui} com.android.systemui
```

2. Verify CPU Usage Distribution Among Threads in Abnormal Processes

In problem scenarios, use the system command **top -H -p $(pidof com.quectel.quecencdectester)** to view the CPU usage of individual threads within the application process. If the CPU usage is primarily from threads within the application, the customer should conduct their own analysis. If the CPU usage is from system framework threads, it should be reported to Redmine for RK handling.

As shown in the figure below, CodecLooper is the encoding and decoding processing thread for the MediaCodec framework. During multi-decoding tests conducted by the Sample program, this framework's encoding and decoding thread consumes the most of CPU usage.



```
  Mem:  8105456K total,  5833500K used,  2271956K free,     4268K buffers
 Swap:  4052724K total,        0K used,  4052724K free,  3013984K cached
800%cpu 288%user   1%nice 117%sys 388%idle   0%iow   5%irq   1%sirq   0%host
  TID USER         PR  NI VIRT  RES  SHR S[%CPU] %MEM     TIME+ THREAD          PROCESS

 5789 u0_a80        4 -16  16G 1.0G  82M R 46.3  13.7   0:05.04 CodecLooper     com.quectel.quecencdectester
 5783 u0_a80        4 -16  16G 1.0G  82M S 45.3  13.7   0:05.51 CodecLooper     com.quectel.quecencdectester
 5787 u0_a80        4 -16  16G 1.0G  82M R 45.0  13.7   0:05.04 CodecLooper     com.quectel.quecencdectester
 5777 u0_a80        4 -16  16G 1.0G  82M S 45.0  13.7   0:05.59 CodecLooper     com.quectel.quecencdectester
 5788 u0_a80        4 -16  16G 1.0G  82M R 44.6  13.7   0:04.56 CodecLooper     com.quectel.quecencdectester
 5781 u0_a80        4 -16  16G 1.0G  82M R 44.3  13.7   0:05.46 CodecLooper     com.quectel.quecencdectester
 5784 u0_a80       10 -10  16G 1.0G  82M S  4.0  13.7   0:00.72 MediaCodec_loop com.quectel.quecencdectester
 5779 u0_a80       10 -10  16G 1.0G  82M S  4.0  13.7   0:00.71 MediaCodec_loop com.quectel.quecencdectester
```

3. Analysis of CPU Usage for Functions

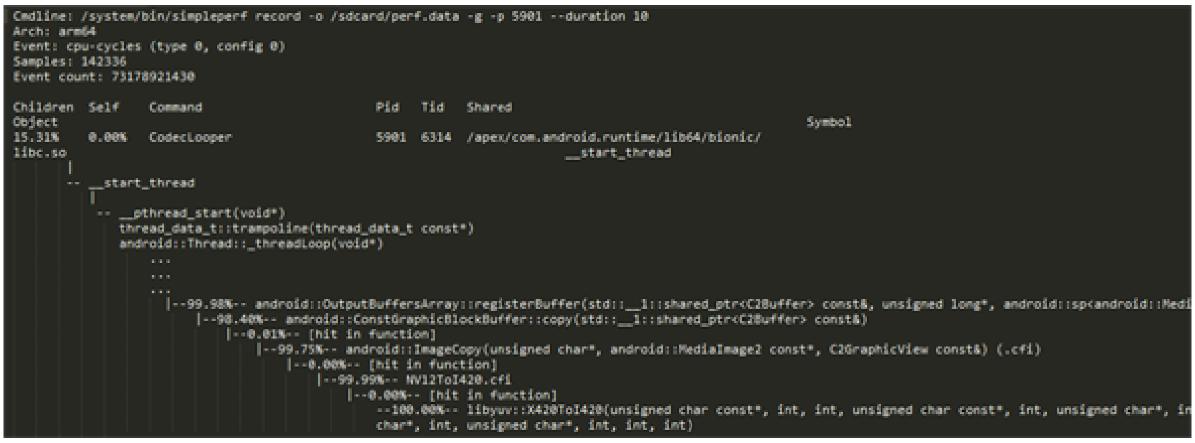
Android CPU performance analysis tools are numerous, with integrated tools such as TraceView\Systrace available via the Android Studio IDE. This chapter introduces the simpleperf tool bundled with the SDK to perform statistical analysis of CPU usage.

Following the analysis of the Sample program above, in multi-decoding applications the framework thread CodecLooper exhibits relatively high CPU usage. We can use the simpleperf tool to determine CPU usage percentage distribution across individual functions.

```
1. Use the following command to record CPU events in real-time, where 1349 is
the process PID, and duration 10 specifies a recording duration of 10 seconds
simpleperf record -o /sdcard/perf.data -g -p 1349 --duration 10

2. After recording ends, use the following command to generate the analysis
report
simpleperf report -i /sdcard/perf.data -g caller > /sdcard/perfdata
```

/sdcard/perfdata contains the generated analysis report, as shown in the figure below. CodecLooper is the thread with the highest CPU usage, accounting for 15.31% of CPU processing time. Within this thread, the majority of time is concentrated on processing the libyuv::X420ToI420 function.



```
Cmdline: /system/bin/simpleperf record -o /sdcard/perf.data -g -p 5901 --duration 10
Arch: arm64
Event: cpu-cycles (type 0, config 0)
Samples: 142336
Event count: 73178921430

Children  Self    Command                     Pid   Tid   Shared
Object                                                                                  Symbol
15.31%    0.00%   CodecLooper                 5901  6314  /apex/com.android.runtime/lib64/bionic/
libc.so                                                              __start_thread
        |
        -- __start_thread
           |
            -- __pthread_start(void*)
               thread_data_t::trampoline(thread_data_t const*)
               android::Thread::_threadLoop(void*)
                     ...
                     ...
                     ...
                      |--99.98%-- android::OutputBuffersArray::registerBuffer(std::__1::shared_ptr<C2Buffer> const&, unsigned long*, android::sp<android::Medi
                          |--98.40%-- android::ConstGraphicBlockBuffer::copy(std::__1::shared_ptr<C2Buffer> const&)
                              |--0.01%-- [hit in function]
                                  |--99.75%-- android::ImageCopy(unsigned char*, android::MediaImage2 const*, C2GraphicView const&) (.cfi)
                                      |--0.00%-- [hit in function]
                                          |--99.99%-- NV12ToI420.cfi
                                              |--0.00%-- [hit in function]
                                                  --100.00%-- libyuv::X420ToI420(unsigned char const*, int, int, unsigned char const*, int, unsigned char*, in
                                                  char*, int, unsigned char*, int, int, int)
```

This function serves to copy the framework's decoded output to the application's external space under the MediaCodec Buffer Mode decoding mode. Since the application cannot process the framework's DMA decoded output, this CPU copy operation is unavoidable. In the chapter `<MediaCodec BufferMode Decoding Efficiency Improvement>` of this document, replacing the CPU copy with platform hardware RGA copy can improve the CPU usage issue in this scenario.

> The same analytical steps can be used to troubleshoot other CPU usage issues in the application.

1.7 Analysis of Memory Leak Issues in Multimedia Applications

[Problem Description]

Multimedia applications running process leads to gradually decreasing system available memory, eventually causing Out-of-Memory (OOM) and system reboot.

[Problem Analysis]

Memory leak issues can be troubleshooted following the steps below:

1. Confirm which process is experiencing a memory leak.

The first step is to determine whether the leak originates from a customized application or a system process. This can be achieved by verifying the meminfo information before and after the issue occurs, comparing the RSS\PSS memory usage across all processes to identify which process exhibits consistently increasing memory consumption, thereby indicating a potential memory leak in that process.

```
dumpsys meminfo
```

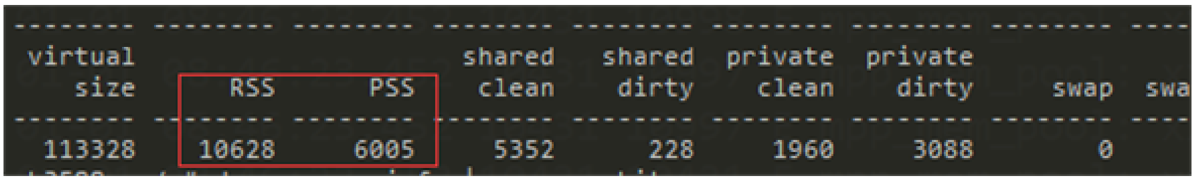
2. Identify the type of memory leak

Identifying the leak type helps better locate the issue. The primary potential leak types for video playback are malloc or dmabuf. Dmabuf is used for decoding output and display, serving as the foundation of zero-copy video playback, allocated through dma buffer interfaces (ION\DRM\dmaBufferHeap). Malloc refers to memory allocated via alloc or malloc functions.

The Linux showmap command is used to locate the memory map allocated by a process.

```
showmap $(pidof xxx)
```

Among these, RSS\PSS represents the memory occupied by the process, in units of K. Identify larger memory blocks in use and determine the memory leak type. If the leak is of the dmaBuf type, primarily inspect buffer allocation and release during video playback initialization, destruction, and info-change events. If it is a malloc-type memory leak, the Android Malloc Debug tool can be used to locate the leaking stack trace.



```
-------- -------- -------- -------- -------- -------- -------- -------- ----
 virtual                     shared   shared  private  private
    size      RSS      PSS    clean    dirty    clean    dirty     swap  swa
-------- -------- -------- -------- -------- -------- -------- -------- ----
  113328    10628     6005     5352      228     1960     3088        0
```

```
Map Memory Object Type:
    - /dmabuf:           DMA buffer type memory
    - malloc or anon:scudo  malloc-type memory
```

3. Locating the Stack of Memory Leak Growth Points with Android Malloc Debug

Malloc Debug is a tool natively provided by Android for debugging native memory issues such as memory leaks\memory corruption\memory release issues. The relevant instruction documentation is located in the SDK bionic/libc/malloc_debug/README.md.

The following is an example, assuming the program name for memory detection is myTest.

```
adb shell stop
adb shell setprop libc.debug.malloc.program myTest
adb shell setprop libc.debug.malloc.options backtrace
adb shell start
# Start Testing and Reproducing Issues...
kill -9 $(pidof myTest)
```

The default captured memory snapshot is saved at the path: /data/local/tmp/backtrace_heap.3302.txt, where 3302 is the process pid of the myTest program.

The generated memory snapshot can be parsed using the SDK native_heapdump_viewer.py tool to generate leak stacks.

```
python development/scripts/native_heapdump_viewer.py --symbols $(OUT)/symbols b
acktrace_heap.3378.txt > out_heap.txt
```

Fix the memory leak points based on the code indicated by the out_heap memory stack information.

1.8 Platform JPEG Hardware Codec Reference Demo

Google's native MediaCodec pipeline does not support JPEG encoding and decoding. The MPP JPEG encapsulation library libhwjpeg can be used to integrate JPEG hardware encoding and decoding functionality into system multimedia applications.

> Related code is located in the SDK path: hardware/rockchip/libhwjpeg

libhwjpeg is used to support JPEG hardware encoding and decoding on the Rockchip platform and serves as the encapsulation of the JPEG encoding and decoding logic within the platform's MPP (Media Process Platform) library.

The MpiJpegEncoder class encapsulates hardware encoding-related operations, while the MpiJpegDecoder class encapsulates hardware decoding-related operations, supporting image or MJPEG stream decoding. The main directories of the project:

```
- src: Library Implementation Code
- inc: application interface header file
- test: User test instance
```

Specific usage instructions can be found in the readme.txt file located in the directory.

2. Video Encoding and Decoding Category

2.1 Media Source Stuttering\Audio-Video Synchronization Issues

Video playback stuttering, audio stuttering, audio-video desynchronization, etc., can be categorized as smoothness issues. The analysis of smoothness issues relies on display frame rate (FPS) and kernel single-frame decoding time.

```
// Display Frame Rate FPS
setprop debug.sf.fps 1
logcat -c ;logcat | grep mFps

// Kernel single-frame decoding time
4.19/5.10 Kernel (Android 10.0 and above)
echo 0x0100 > /sys/module/rk_vcodec/parameters/mpp_dev_debug
cat /proc/kmsg

4.4 Kernel (Android 7.1 to 9.0 versions)
echo 0x0100 > /sys/module/rk_vcodec/parameters/debug
cat /proc/kmsg
```

For video playback not being smooth, the main reasons can be attributed to the following points, and engineers encountering related issues can refer to the following methods for troubleshooting:

1. Verify if platform hardware decoding is being used

Some video websites or applications may use software decoding to decode video formats due to unknown code and API usage. Therefore, if stuttering occurs in such cases, the corresponding video format should first be identified in the Logcat logs. If the video format and specifications are confirmed to be within the chip's decoding capabilities but platform hardware decoding is not being used, please first check whether the program or website contains the configurations that control whether hardware decoding is used.

Determine whether platform hardware decoding is used by entering a command to query the kernel's single-frame decoding time. If processed by kernel hardware, a value will be printed, indicating hardware decoding is currently in use, otherwise, software decoding is being used.

2. Confirm whether the issue lies in decoding or display

Video playback involves two operations: decoding and rendering. The decoded output is submitted to the display component for rendering. Therefore, please first refer to the chip codec capability specification table in Section 1.2 to determine whether the video sources fall within the decoding capability range. For such sources, use the commands mentioned above to query the per-frame decoding time of the kernel and the display frame rate.

If the single-frame decoding time is sufficient (30 fps source within 33 ms / 60 fps source within 16 ms), it can first be categorized as insufficient display synthesis efficiency, and troubleshooting should refer to Section 3.

3. Insufficient Display Synthesis Efficiency

Abnormal configuration of the display refresh rate can lead to inefficient HWC synthesis efficiency, therefore, the first step is to verify whether the display refresh rate is correctly configured in the DTS.

```
rk3566_r:/ # cat /d/dri/0/summary
Video Port0: DISABLED
Video Port1: ACTIVE
    Connector: DSI-1
        bus_format[100a]: RGB888_1X24
        overlay_mode[0] output_mode[0] color_space[0]
    Display mode: 1080x1920p60
        clk[132000] real_clk[132000] type[48] flag[a]
        H: 1080 1095 1097 1127
```

The screen refresh rate is generally configured to 60 fps. If issues occur, correct the panel-timing settings corresponding to the display panel in the DTS configuration using the following formula.

```
clock-frequency = (hactive + hback-porch + hfront-porch + hsync-len) * (vactive +
vback-porch + vfrontporch + vsync-len) * fps
```

For Non-screen parameter configuration issues, the common causes are typically the following:

- The video display output has an angular orientation, i.e., rotated screen output.
- Scenario has excessive Surface layer count with multiple layers present.
- Video format is not supported.

The following logs can be captured and submitted to Redmine for assignment to the corresponding engineer(s) to handle.

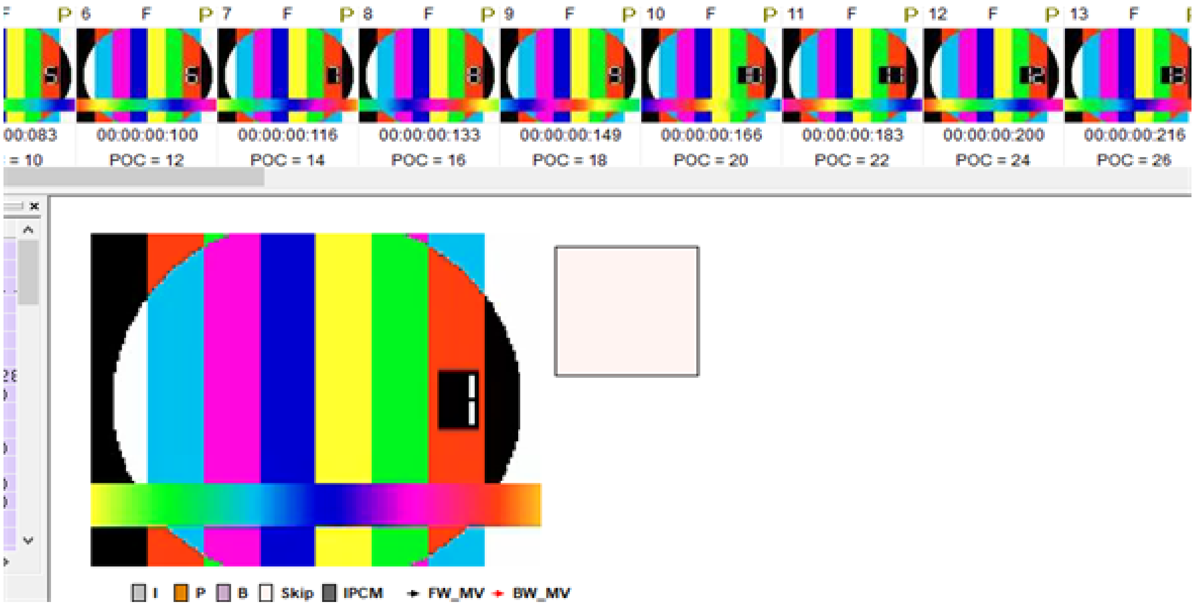
```
// Check if the synthesis strategy is functioning properly via SurfaceFlinger
Services.
dumpsys SurfaceFlinger

// If abnormal, print the HWC log to identify the cause of the issue
adb shell "setprop sys.hwc.log 51”
adb shell "logcat -c ;logcat" > hwc
```

4. Decoding Errors in the Video Decoding Process

If logs contain prints related to "error frame", it indicates that the hardware decoding process has failed and dropped frames have occurred. For such issues, first verify that the bitstream provided to the decoder is correct and contains no dropped frames. The media framework provides several dump methods for decoding inputs. Please refer to the commands in Chapter 4 for operations.

The captured input is a pure video track. PC tools can be used to view and analyze the stream. For H264 format, use eseye or Vega H264 Analyzer. For H265 format, use HEVCAnalyzer. These tools typically can determine whether the bitstream itself has issues or frame dropping. As shown in the following Vega analysis tool preview, if the POC values are consecutive and the preview shows no errors, it indicates the bitstream is normal and no frames are dropped.



After confirming the decoding input is correct, it indicates that the current decoding framework has compatibility issues with the specific bitstream. You may submit the corresponding bitstream along with the Logcat logs to Redmine for assignment to the relevant engineer for handling.

2.2 Video/Screen Recording Output Blurry or Mosaic

[Problem Description]

Video/Screen recording output file blurred\mosaic\unclear

[Problem Analysis]

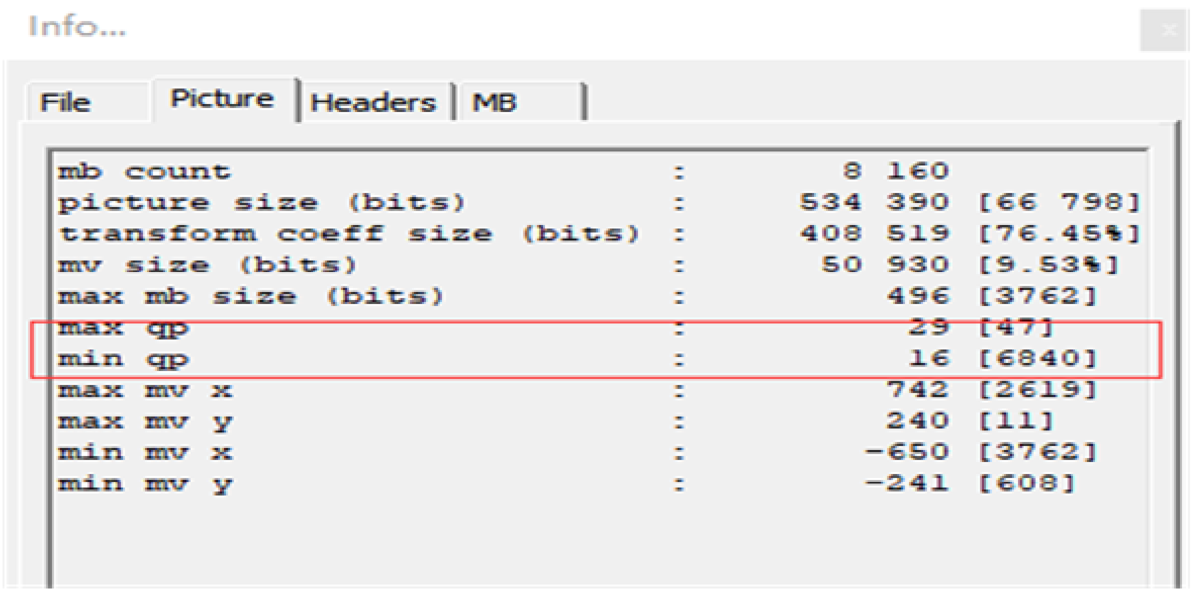
1. Platform encoding falls under lossy compression encoding. During the encoding process, there is a loss of image data, thus resulting in differences between the encoded output and

the original image.
2. Blurry or mosaic artifacts are usually caused by low local encoding quality. QP is the quantization parameter for encoding quality, with a range from 1 to 51. The lower the QP value, the higher the encoding quality.

In cases where the user interface does not restrict the QP range, the QP range is determined by the user-set bitrate. The higher the bitrate, the higher the encoding quality, and the smaller the QP value per frame. Therefore, the typical troubleshooting steps for encoding mosaic issues are as follows:

1. Check the QP Range of the Encoded Output

eseye or Vega H264 Analyzer and other PC tools provide single-frame image quality queries. As shown in the figure below, the QP range of this frame is 16~29. Macroblocks with QP values exceeding 40 may cause blurriness or mosaic artifacts. If QP quality values in the 40s appear, further investigation is required to determine if bitrate and encoding quality strategies need adjustment. If the image QP range is normal but mosaic artifacts still occur, the encoded input must be captured to verify whether the input itself contains mosaic patterns.



```
mb count                   :       8 160
picture size (bits)        :     534 390 [66 798]
transform coeff size (bits) :    408 519 [76.45%]
mv size (bits)             :      50 930 [9.53%]
max mb size (bits)         :         496 [3762]
max qp                     :          29 [47]
min qp                     :          16 [6840]
max mv x                   :         742 [2619]
max mv y                   :         240 [11]
min mv x                   :        -650 [3762]
min mv y                   :        -241 [608]
```

2. Check the Bitrate Parameters Rationality

The compression performance of different encoders varies significantly, and there is no standard bitrate reference table. Determining whether a bitrate is reasonable can be done by comparing against the output image quality from encoding.

```
- a) If the actual bitrate >= the application-set bitrate and blurred images
occur with excessively high QP values
      Determined that the user-specified bitrate is too low. Recommend increasing
the bitrate for testing.
- b) If the actual bitrate < the application-set bitrate, and image blurring
occurs with excessively high QP values
     Verify whether the application interface has set the variable bitrate mode
BITRATE_MODE_VBR.
- c) If the actual bitrate matches the bitrate set by the application, manual
adjustment of the image QP strategy may be required.
```

3. Adjust Image QP Strategy

Firstly, if the current configuration is using the baseline encoding, switching to the high profile may be considered. The high profile uses CABAC entropy encoding, which results in a higher overall compression ratio and better image quality at the same bitrate.

In Android 12 and later versions, MediaCodec supports image quality control, allowing the user interface to define the QP range. Setting Max QP not greater than 40 can effectively mitigate blurry and mosaic issues.

The application layer MediaCodec interface code can be configured as follows (where qp-i-min represents the minimum QP value for I-frames, and qp-p-max represents the maximum QP value for P-frames):

```
format->setInt32("video-qp-i-min", 10);
format->setInt32("video-qp-i-max", 40);
format->setInt32("video-qp-p-min", 10);
format->setInt32("video-qp-p-max", 40);
```

It should be noted that adjusting the QP range may lead to bitrate overflow. Extra caution is required to monitor the encoded output bitrate.

Android 11 and earlier versions currently do not support QP range control via the user interface. The QP range can be limited by modifying the MPP parameter configuration interface.

```
diff --git a/mpp/legacy/vpu_api_legacy.cpp b/mpp/legacy/vpu_api_legacy.cpp
index 33d08ccb..b937f0bc 100644
--- a/mpp/legacy/vpu_api_legacy.cpp
+++ b/mpp/legacy/vpu_api_legacy.cpp
@@ -173,9 +173,9 @@ static MPP_RET vpu_api_set_enc_cfg(MppCtx mpp_ctx, MppApi
*mpi, MppEncCfg enc_cf
         mpp_enc_cfg_set_s32(enc_cfg, "h264:cabac_idc", 0);
         mpp_enc_cfg_set_s32(enc_cfg, "h264:qp_init", is_fix_qp ? qp : -1);
         mpp_enc_cfg_set_s32(enc_cfg, "h264:qp_min", is_fix_qp  ? qp : 10);
-        mpp_enc_cfg_set_s32(enc_cfg, "h264:qp_max", is_fix_qp ? qp : 51);
+        mpp_enc_cfg_set_s32(enc_cfg, "h264:qp_max", is_fix_qp ? qp : 40);
         mpp_enc_cfg_set_s32(enc_cfg, "h264:qp_min_i", 10);
-        mpp_enc_cfg_set_s32(enc_cfg, "h264:qp_max_i", 51);
+        mpp_enc_cfg_set_s32(enc_cfg, "h264:qp_max_i", 40);
         mpp_enc_cfg_set_s32(enc_cfg, "h264:qp_step", 4);
         mpp_enc_cfg_set_s32(enc_cfg, "h264:qp_delta_ip", 3);
     } break;
```

2.3 MediaCodec Encoding Bitrate Configuration Overflow/Exceeding Bitrate

[Problem Description]

In Android versions >= 12, the actual encoded bitrate consistently exceeds the configured value. The typical scenario occurs when the configured bitrate is set too low, such as setting a bitrate of 1Mbps for 1080P resolution encoding, but the actual encoded bitrate exceeds 4Mbps.

[Problem Reason]

Android 12 introduces the newly enabled media format shaping feature to correct unreasonable configurations such as QP and bitrate settings.

1Mbps bitrate is deemed unreasonable for 1080P resolution by the framework. The VQApply module of the framework calculates a reference bitrate based on user-set bitrate, width, height, bpp and other parameters, then corrects and assigns this value to the encoder for use. This corrected value is typically greater than (w*h*bpp).

The logs typically include the following VQApply printout, indicating that the user-configured bitrate of 1258291 bps was corrected by the VQApply framework to 4727808 bps, which is then assigned to the encoder as the actual bitrate for use. This results in scenarios where the actual bitrate is significantly higher than expected.

```
D VQApply : minquality/target bitrate raised from 1258291 to 4727808 bps
```

[Issue Resolution]

Google provides the property value debug.stagefright.enableshaping to disable the framework's format shaping correction. If this feature is not required, the property can be set to disable it using setprop debug.stagefright.enableshaping 0.

Similarly, it can be disabled directly in the code.

Path: frameworks/av

```
diff --git a/media/libstagefright/MediaCodec.cpp
b/media/libstagefright/MediaCodec.cpp
index 382324a91c..dd1dfa062f 100644
--- a/media/libstagefright/MediaCodec.cpp
+++ b/media/libstagefright/MediaCodec.cpp
@@ -2054,7 +2054,7 @@ status_t
MediaCodec::setOnFirstTunnelFrameReadyNotification(const sp<AMessage> &
   * MediaFormat Shaping forward declarations
   * including the property name we use for control.
   */
-static int enableMediaFormatShapingDefault = 1;
+static int enableMediaFormatShapingDefault = 0;
 static const char enableMediaFormatShapingProperty[] =
"debug.stagefright.enableshaping";
 static void mapFormat(AString componentName, const sp<AMessage> &format, const
char *kind,
                       bool reverse);
```

2.4 Codec Initialization Failed Log Indicates "MPP HAL xxx init failed"

[Problem Description]

Codec initialization failure in scenarios such as video playback\recording\screen capture\photography, and the log indicates "mpp hal xxx init failed"

[Issue Analysis]

```
E HAL_JPEG_VDPU2: hal_jpegd_vdpu2_init mpp_dev_init failed. ret: -1
E mpp_hal : mpp_hal_init hal jpegd init failed ret -1
```

Codec initialization failed, and the problem log indicates "mpp hal xxx init failed". This issue is typically caused by incorrect configuration in the customer's DTS, which has not enabled the corresponding codec node(s). Adding and enabling the corresponding node(s) in the customer's DTS file will resolve the issue.

The above log hal_jpegd_vdpu2 indicates two key pieces of information:

- 1. jpegd, i.e., JPEG decoder initialization failed.
- 2. vdpu is the corresponding nodes to be enabled.

Verify if the vdpu node is correctly enabled under the MPP driver device directory.

```
ls -al /proc/mpp_service/
```

If disabled is confirmed, refer to the corresponding chip's dtsi configuration under the kernel, such as rk3588-evb.dtsi \ rk3399-android.dtsi.

The corresponding configurations added for the aforementioned issue log are as follows:

```
&mpp_srv {
    status = "okay";
};

&vdpu {
    status = "okay";
};
```

Another similar error log entry:

```
E hal_h264e_vepu541: hal_h264e_vepu541_init mpp_dev_init failed. ret: -1
E mpp_enc_hal: mpp_enc_hal_init hal hal_h264e init failed ret -1
E mpp_enc_hal: mpp_enc_hal_init could not found coding type 7
```

h264e (i.e., the 264 encoder) initialization failed. The enabled node name to be added is vepu. Thus, the corresponding configurations to be added are:

```
&mpp_srv {
  status = "okay";
};

&vepu {
  status = "okay";
};
```

2.5 MediaCodec BufferMode Decoding Efficiency Improvement

[Problem Description]

The application interface uses MediaCodec Buffer Mode without configuring a surface for decoding, which results in increased single-frame decoding time and higher CPU usage. When the issue occurs, the application scenario typically has a relatively high CPU load by itself.

[Problem Reason]

The application interface MediaCodec does not configure a Surface for decoding, meaning that the external application interface needs to retrieve the decoded output buffer for additional processing. Since the framework's decoding output is in dmaBuffer format which cannot be directly provided to the application, an additional copy operation is required in Buffer Mode. The decoded output must be copied from the framework to the application interface.

The corresponding Surface Mode refers to the scenario of configuring Surface for decoding and display, which is commonly known as the zero-copy scenario. In this mode, decoding and display do not require any copying and directly use dmaBuffer transfer, which is more efficient than Buffer Mode.

Buffer Mode decoding mode, due to an additional CPU copying process, results in increased decoding processing time per single frame and higher CPU usage.

[Solution]

Using the system RGA hardware copy instead of CPU-based copy can significantly improve decoding time and reduce CPU usage in the scenario.

For specific modification patches, please refer to the rk patch bulletin: https://redmine.rock-chips.com/issues/418670

3. Application Usage Category

3.1 Kodi\Bilibili and other applications video playback not using hardware decoder

[Problem Description]

In versions above Android 12, applications using the GitHub ijkmedia playback framework (currently known to affect Kodi and Bilibili applications) are unable to use the hardware decoders during video playback.

[Problem Reason]

The ijkmedia framework only filters codec names starting with 'OMX.' as hardware decoders when selecting a MediaCodec decoder. However, Andorid 12 uses the codec2 new framework, where the hardware decoder's codec name is c2.rk.avc.decoder, thus failing to select the platform's hardware decoder.

[Solution]

Modified to add the "OMX.c2" decoder to support adaptation to the ijkmedia framework, such as redirecting c2.rk.avc.decoder to OMX.c2.rk.avc.decoder.

Apart from applications such as Kodi\Bilibili, custom client applications using the ijkmedia media framework also fail to use hardware decoding. This can be confirmed by checking if the logs contain the "IJKMEDIA" label.

Specific modification patches can be found in the rk patch bulletin: https://redmine.rock-chips.com/issues/396998

> This issue has been identified as a problem within the ijkmedia framework. However, the formal resolution will still require modifications to the framework's codebase. Therefore, a temporary patch is provided as an interim solution without submission to the official version.

3.2 WebView Video Playback Failure or White Space at the Top

[Problem Description]

On SoC chips with AFBC decoding output functionality (rk3566\rk3568\rk3588\rk3528), client applications or web pages experience video playback failure or a white margin appears at the top when using WebView.

[Issue Cause]

Background: Under the fbc decoding format, the hardware outputs a fixed expansion of 4 upwards, resulting in a height offset of 4 lines in the decoded output.

WebView video playback ultimately uses ImageReader\NdkImageReader to acquire the video surface's buffer for rendering. In fbc mode, the video buffer has a 4-line height offset, so proper handling of the buffer's crop is required during display. Otherwise, the 4-line offset height will manifest as white borders when rendered.

[Solution]

The fbc mode is only enabled for video playback with resolutions above 1080P. Therefore, the following patch is applicable to rk356x\rk3588\rk3528 webview video playback scenarios with resolutions exceeding 1080P.

1. Regarding WebView Video Playback Failure

Error log:

```
E NdkImageReader: Crop left top corner [0, 4] not at origin
```

In the default code, the ImageReader imposes a restriction on the buffer's left-top corner, which is fixed at 0. However, the decoding output under fbc mode includes a 4-line height offset, resulting in the top corner being set to 4. Removing this restriction can resolve the playback failure issue.

Patch path: frameworks/av

```
diff --git a/media/ndk/NdkImageReader.cpp b/media/ndk/NdkImageReader.cpp
index 067c8f4ae5..6df40cc975 100644
--- a/media/ndk/NdkImageReader.cpp
+++ b/media/ndk/NdkImageReader.cpp
@@ -455,7 +455,7 @@ AImageReader::acquireImageLocked(/*out*/AImage** image,
/*out*/int* acquireFence
         Point lt = buffer->mCrop.leftTop();
         if (lt.x != 0 || lt.y != 0) {
             ALOGE("Crop left top corner [%d, %d] not at origin", lt.x, lt.y);
-            return AMEDIA_ERROR_UNKNOWN;
+            // return AMEDIA_ERROR_UNKNOWN;
         }
```

2. Regarding white borders appearing during WebView video playback

For devices already applied with the aforementioned restrictions removal patches, videos played in webview with resolutions greater than 1080P exhibit white borders at the top.

The reason for no white borders in local player playback lies in the autonomous controllability of source crop at the player end, enabling the transmission of the desired selection and cropping area to the surface.

WebView rendering obtains the surface buffer through ImageReader, and the application framework subsequently calls getHardwareBuffer to directly acquire the GraphicBuffer. As a result, the cropping processing is transferred to the WebView application code, and the framework is currently unable to resolve the white margins. If the customer cannot accept the white margins, the FBC decoding mode can be disabled in the codec component, with the result that non-FBC mode decoding may experience a slight performance degradation.

Patches for Android 11 and earlier versions (hardware/rockchip/omx_il):

```
diff --git a/osal/Rockchip_OSAL_Android.cpp b/osal/Rockchip_OSAL_Android.cpp
index 5950cd0..ac599cf 100755
--- a/osal/Rockchip_OSAL_Android.cpp
+++ b/osal/Rockchip_OSAL_Android.cpp
@@ -317,6 +317,7 @@ OMX_BOOL Rockchip_OSAL_Check_Use_FBCMode(OMX_VIDEO_CODINGTYPE
codecId, int32_t d
     OMX_U32  pValue;
     OMX_U32  width, height;

+    return OMX_FALSE;
     if (pPort->bufferProcessType != BUFFER_SHARE) {
         return OMX_FALSE;
     }
```

Patches for Android 12 and later versions (vendor/rockchip/hardware/interfaces/codec2):

```
diff --git a/component/osal/C2RKChipCapDef.cpp
b/component/osal/C2RKChipCapDef.cpp
index 78659ab..b66fb31 100644
--- a/component/osal/C2RKChipCapDef.cpp
+++ b/component/osal/C2RKChipCapDef.cpp
@@ -297,6 +297,7 @@ uint32_t C2RKChipCapDef::getGrallocVersion() {

 uint32_t C2RKChipCapDef::getFbcOutputMode(MppCodingType codecId) {
     uint32_t fbcMode = 0;
+    return 0;

     for (int i = 0; i < mChipCapInfo->fbcCapNum; i++) {
         if (mChipCapInfo->fbcCaps[i].codecId == codecId) {
```

3.3 iQIYI APP Crashes During Video Playback Stress Test

[Problem Description]

In Android 10.0 and later versions, when performing video playback stress test using the iQIYI app, memory leaks caused lowmem (low memory), eventually resulting in system anomalies and APP crashes.

[Problem Reason]

Leak type is DMA buffer memory. The leak occurs during the advertisement playback when the application uses the MediaPlayer interface with relatively late surface setup timing. The player initially decodes using buffer mode before the surface setup request arrives, which triggers a surface change event. The player needs to allocate new decoding buffers for the new surface and synchronously release the old decoding buffers.

In this scenario, the reference count of the old decoding output buffer was not reset to zero, leading to these buffers not being fully released.

[Solution]

Since the leak type is a DMA buffer, it can be determined through the system's dma_buf status information. If, during stress test, the number of DMA buffers mapped to the VPU continues to increase without being released, the issue can ultimately be confirmed.

```
cat /d/dma_buf/bufinfo
```

The specific modification patches can be found in the rk patch bulletin: https://redmine.rock-chips.com/issues/423914

3.4 The player application failed to retrieve thumbnails in real-time during playback

[Problem Description]

The player application failed to retrieve thumbnails using MediaMetadataRetriever during video playback. Log indicates:

```
MediaMetadataRetrieverJNI( 1574): getFrameAtTime: videoFrame is a NULL pointer
```

[Problem Reason]

Currently, within the player framework, to save bandwidth and ensure that thumbnail parsing in the background does not affect foreground video playback, the thumbnail functionality is disabled by default during video playback, resulting in the failure to use thumbnails during the playback process.

[Solution]

Versions above Android 12 provide an attribute switch to enable this restriction. If required by the customer application development, the following attribute can be added to the system prop file. For versions below Android 12, please submit a Redmine ticket to the corresponding engineer to update the library.

```
setprop rt_retriever_enable 1
```

3.5 RK356X Screen Recording or Video Encoding Green Screen, Log Indicates RGA Error

[Problem Description]

RK356X device screen recording or video encoding green screen, log shows RGA error, indicating unsupported buffers outside the 4GB address space.

```
E/rga_mm  (    0): RGA_MMU unsupported Memory larger than 4G!
E/rga_mm  (    0): scheduler core[4] unsupported mm_flag[0x0]!
E/rga_mm  (    0): rga_mm_map_buffer map dma_buf error!
E/rga_mm  (    0): job buffer map failed!
E/rga_mm  (    0): src channel map job buffer failed!
E/rga_mm  (    0): failed to map buffer
```

[Problem Reason]

1. The RK356X uses hardware RGA2 for image processing. The RGA2 hardware design can only handle a 32-bit address space, therefore buffers sent to RGA for processing must be allocated within 4G space.
2. The RK356X encoder can only process aligned buffers, so the Codec component relies on RGA for pre-processing before encoding. The encoding input buffer is aligned before being sent to the hardware VPU encoder.

The cause of the error is that the address space of the src or dst buffer in the RGA preprocessing of the Codec component exceeds 4G. Since Android framework's Dma buffer allocations ultimately go through Gralloc, this scenario requires Gralloc to allocate buffers within the 4G address space.

[Solution]

The dts buffer of RGA preprocessing is eventually sent into the hardware encoder. Modify to use GraphicBufferAllocator and pass the GRALLOC_USAGE_WITHIN_4G flag, which requires to allocate a 4G buffer, to Gralloc.

The src buffer for the Codec component's RGA preprocessing originates from the surface. The scenario falls under the BufferQueue producer-consumer model, where the Codec component acts as the consumer end and the Surface serves as the producer end.

The Surface buffer is controlled by the producer end but the usage set by the consumer end will ultimately be applied to the producer end's Surface buffer allocation. In the Codec component, the usage GRALLOC_USAGE_HW_VIDEO_ENCODER is used to indicate the current scenario is a hardware encoder.

Therefore, this usage is used for differentiation. When allocating the buffer, if the usage includes GRALLOC_USAGE_HW_VIDEO_ENCODER and the platform is RK356X, then control Gralloc to allocate a 4G buffer for this scenario.

The specific modification patches please refer to rk patch bulletin: https://redmine.rock-chips.com/issues/425094

3.6 Removal of Custom Audio-Video Format Support in Video Player Application

[Problem Description]

Local application players or other MediaPlayer players need to exclude support for certain audio and video formats. For example, the VP9 video format has copyright issues abroad, so support for VP9 video format must be excluded, while domestic devices are unaffected.

[Solution]

Providing configuration file to support customer to exclude custom audio and video formats. If it is required to exclude VC1\VP9 video formats and AAC\MP3 audio format support on RK3588 devices, modify the SDK with the following patch (path: device/rockchip/common):

```
diff --git a/rt_video_config.xml b/rt_video_config.xml
index 067b81a8..46ec980b 100644
--- a/rt_video_config.xml
+++ b/rt_video_config.xml
@@ -224,6 +224,7 @@
         <chip name="RK3566,RK3567,RK3568">
             <include
name="mpeg1,mpeg2,mpeg4,vp8,h264_8k_10bit_high422,hevc_8k_10bit"/>
             <include name="vp9_4k_10bit"/>
+            <forbid name="vc1,vp9"/>
         </chip>

         <chip name="RK3528">
```

```
diff --git a/rt_audio_config.xml b/rt_audio_config.xml
index 182e2a83..af645673 100644
--- a/rt_audio_config.xml
+++ b/rt_audio_config.xml
@@ -34,4 +34,10 @@
     <formats>
     </formats>
   </bitstream>
+  <forbid>
+    <formats>
+      <format>AAC</format>
+      <format>MP3</format>
+    </formats>
+  </forbid>
 </sound>
```

Manually modify and verify: Update the rt_audio_config.xml/rt_video_config.xml configuration files under the /system/etc path, then re-initiate playback to confirm effectiveness.

Other disableable audio and video formats:

```
Video optional formats: mpeg1, mpeg2, h263, mpeg4, wmv1, wmv2, wmv3, h264, vp8,
vp9, hevc, vc1, avs, avs+, avs2, flv1, av1, MVC

Audio optional formats: AAC, APE, MP3, WMALOSSLESS, WMAPRO, WMAV1, WMAV2,
ADPCM_IMA_QT, VORBIS, PCM_S16LE, PCM_S24LE, FLAC, MP1, MP2, AMR_WB, AMR_NB, G279,
OPUS, PCM_ALAW,
PCM_MULAW, ADPCM_G722, ADPCM_G726
```

3.7 Camera 4K Video Recording Exhibits Reddish During Playback

[Issue Description]

In devices supporting 4K encoding, when recording video using Camera or CameraRecorder for video capture, the recorded video playback does not match the actual image and exhibits a reddish and overly vibrant appearance.

[Issue Cause]

1. The Android native encoding framework configures a default color gamut for instances without explicitly defined colorAspect color gamut information. When the resolution is greater than or equal to 4K, the framework defaults to configuring the BT2020 color gamut.
2. Devices supporting HDR will enable HDR display when decoding and playing video sources configured with BT2020.

[Solution]

The HDR display effect produces more vivid colors, which is a positive enhancement for display. If concerned about this effect, the default configuration of the framework can be modified to set the 4K default color gamut to BT709.

Modify default color gamut configuration function setDefaultCodecColorAspectsIfNeeded in frameworks/av:

```
diff --git a/media/libstagefright/foundation/ColorUtils.cpp
b/media/libstagefright/foundation/ColorUtils.cpp
index fa722b5572..f253b68780 100644
--- a/media/libstagefright/foundation/ColorUtils.cpp
+++ b/media/libstagefright/foundation/ColorUtils.cpp
@@ -377,8 +377,8 @@ void ColorUtils::setDefaultCodecColorAspectsIfNeeded(
     // Default to BT2020, BT709 or BT601 based on size. Allow 2.35:1 aspect
ratio. Limit BT601
     // to PAL or smaller, BT2020 to 4K or larger, leaving BT709 for all
resolutions in between.
     if (width >= 3840 || height >= 3840 || width * (int64_t)height >= 3840 *
1634) {
-        primaries = ColorAspects::PrimariesBT2020;
-        coeffs = ColorAspects::MatrixBT2020;
+        primaries = ColorAspects::PrimariesBT709_5;
+        coeffs = ColorAspects::MatrixBT709_5;
     } else if ((width <= 720 && height > 480 && height <= 576)
             || (height <= 720 && width > 480 && width <= 576)) {
         primaries = ColorAspects::PrimariesBT601_6_625;
```

3.8 Video Switching Displays Black Frame with the Player

[Problem Description]

By default, when switching videos, the last frame of the previous video is maintained until the decoded output image of the next video displayed. This mode is called still-frame mode. Correspondingly, if the last frame of the previous video should not be maintained when playing a new video, a black frame is injected into the surface at the end of playback. This mode is called black-frame mode.

[Solution]

In the player using Android MediaPlayer interface, setting the following property can enable black frame mode. If the application development requires black frame mode, these properties should be added to the system prop file.

```
adb shell setprop media.rockit.video.black_frame 1
```

3.9 Intermittent Artifact Distortion or Stuttering in Screen Mirroring Display

[Problem Description]

When in-vehicle or other P2P screen-mirroring devices mirror to RK devices, screen artifacts or stuttering may occur intermittently. Logs indicate the decoder is proactively dropping frames due to internal error frames.

```
C2RKMpiDec: skip error frame with pts 0
```

[Problem Analysis]

The reason is network packet loss. The current decoder adopts a dropping strategy for corrupted frames. Network packet loss causes the entire corresponding GOP sequence of that frame to be marked as error. Therefore, excessive packet loss may manifest as stuttering. Excessive packet loss leading to the POC reference sequence becoming disordered and unrecoverable will manifest as color artifacts.

Refer to the following steps for troubleshooting:

1. Check the packet loss condition of the hardware Wi-Fi P2P connection and ensure the stability of the hardware connection.
2. If the encoding end is autonomously controllable, modifications such as adjusting to use TCP transmission or reducing the GOP interval can be attempted based on specific circumstances.
3. When dealing with real-time streaming transmission scenarios, packet loss and frame dropping in the bitstream caused by network instability can be addressed by the decoder internally supporting the disabling of POC (Picture Order Count) continuity checks. Leveraging the decoder's internal error correction functionality, minor packet loss and frame drops can be recovered without affecting overall playback smoothness. Refer to the following user configurations.

MediaCodec enables configuration extension parameters (this configuration is currently supported only on Android versions >= 13):

```
Configuration Method:
mediaFormat.setInteger("vendor.disable-dpb-check.value", 1);

Confirmation Effective Log:
c2_info("disable poc discontinuous check");
```

Note:

1. Hardware error correction is not limitless. Overly severe frame dropping can still cause screen artifacts.
2. POC continuity detection is a standard process for bitstream decoding and is therefore supported solely as a user configuration feature.

3.10 Player seek operation causes progress bar jump back

[Problem Analysis]

The player's seek operation is non-precise seek, and the media source's seek relies on IDR frames. Therefore, when specifying a time point for a seek operation, the player will jump to the position of the first IDR frame near the specified time and start playing from there. This results in discrepancies between the progress bar and the dragged time point, manifesting as backward jumps in the progress bar.

Some applications implement precise seek by first locating the position of the I-frame immediately before the target timestamp, then decoding sequentially until the target timestamp frame is obtained before playback starts. This approach can cause prolonged decoding delays and unresponsive seek operations when IDR intervals are large or the chip's decoding capability is limited. Therefore, the SDK does not support precise seek functionality.

3.11 TikTok\Youku Video Browsing Encounter Android Reboot Issues

[Issue Description]

TikTok\Youku APK occasionally triggers an Android system reboot while scrolling through videos, and logs indicate surfaceFlinger crash anomalies.

```
E mali_config_interface: Unsupported dataspace standard (xxxx)
D skia: Could not create EGL image, err = (0x3003)
```

[Problem Analysis]

When using GPU rendering, and the parsed color gamut values of the played media are of a type unsupported by the GPU, the color gamut values are passed to Mali. The Mali interface returns unsupported, which causes surfaceFlinger to crash.

Unsupported types such as:

```
HAL_DATASPACE_STANDARD_BT601_625_UNADJUSTED = 196608
HAL_DATASPACE_STANDARD_BT601_525_UNADJUSTED = 327680
HAL_DATASPACE_STANDARD_BT2020_CONSTANT_LUMINANCE = 458752
HAL_DATASPACE_STANDARD_BT470M = 524288
HAL_DATASPACE_STANDARD_DCI_P3 = 655360
```

[Solution]

Perform compatibility processing in the decoder HAL and Mali library to convert the transmitted color gamut values to those supported by the GPU.

The specific modification patches please refer to rk patch bulletin: https://redmine.rock-chips.com/issues/533602