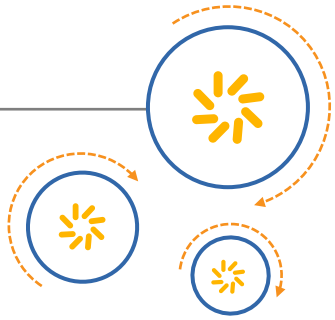
Qualcomm Technologies, Inc.



Android Video

Debug guide

80-NU330-1 C

May 12, 2017

Qualcomm Technologies, Inc.
5775 Morehouse Drive
San Diego, CA 92121
U.S.A.

Revision history

Revision	Date	Description
A	December 2014	Initial release
B	March 2015	Reference to *Multimedia Driver Development and Bringup Guide – Video* moved from a separate chapter into the Introduction; added section on Video Core Firmware Debugging; four chapters related to Wi-Fi Display moved to *Wi-Fi Display Debug Guide;* added two documents to References
C	May 2017	▪ Title updated to replace the words “Multimedia” with “Android” ▪ Numerous changes were made to this document; it should be read in its entirety.

Contents

Tables

1 Introduction

1.1 Purpose

This document provides a variety of solutions that customers can use to enable debug information and debug video use cases; the document is based on SalesForce solution 00028029.

This document is applicable to MSM8992, MSM8994, MSM8936, MSM8939, MSM8x74, APQ8084, MSM8x26, MSM8x09, and MSM8916 chipsets.

To debug Wi-Fi display issues, see *Wi-Fi Display Debug Guide* (80-NP566-1).

Ensure you have followed the instructions in the *Multimedia Driver Development and Bringup Guide - Video* (80-NU323-5) before using any of these debugging guidelines.

NOTE: Refer to the appropriate chipset multimedia video overview document listed in Appendix B to verify whether the device supports specific features such as content protection playback, broadcast, Wi-Fi Display, etc.

1.2 Conventions

Function declarations, function names, type declarations, attributes, and code samples appear in a different font, for example, `#include`.

Code variables appear in angle brackets, for example, `<number>`.

Commands to be entered appear in a different font, for example, **`copy a:*.* b:`**.

Button and key names appear in bold font, for example, click **Save** or press **Enter**.

Shading indicates content that has been added or changed in this revision of the document.

1.3 Technical assistance

For assistance or clarification on information in this document, submit a case to Qualcomm Technologies, Inc. (QTI) at https://createpoint.qti.qualcomm.com/.

If you do not have access to the CDMATech Support website, register for access or send email to support.cdmatech@qti.qualcomm.com.

2 Playback/Recording – Performance

2.1 Debug a performance issue

1. Check the number of frame drops while playing a video

   ```
   adb shell setprop persist.debug.sf.stats 1
   ```

   Sample log

   ```
   I/NuPlayerDriver(396): NuPlayer
   I/NuPlayerDriver(396): mime(video/avc)
   I/NuPlayerDriver(396): decoder(OMX.qcom.video.decoder.avc)
   I/NuPlayerDriver(396): resolution(1920 x 1080)
   I/NuPlayerDriver(396): numFramesTotal(1160), numFramesDropped(1),
   percentageDropped(0%)
   ```

2. Modify the Venus clock and verify the use case

   a. Set Venus Clock to TURBO

      ```
      adb shell setprop vidc.debug.turbo 1
      ```

   b. Disable DCVS for decoder

      ```
      adb shell "echo 0 > /d/msm_vidc/dcvs_dec_mode"
      ```

   c. Disable DCVS for encoder

      ```
      adb shell "echo 0 > /d/msm_vidc/dcvs_enc_mode"
      ```

3. Enable all CPUs.

   ```
   adb shell stop mpdecision
   adb shell stop thermald
   adb shell "echo 1 > /sys/devices/system/cpu/cpu1/online"
   adb shell "echo 1 > /sys/devices/system/cpu/cpu2/online"
   adb shell "echo 1 > /sys/devices/system/cpu/cpu3/online"
   ```

   If chipset is Quadcore, do same for cpu4, cpu5 , cpu6 and cpu7

4. Change the CPU to Performance mode.

   ```
   adb shell "echo performance >
   /sys/devices/system/cpu/cpu0/cpufreq/scaling_governor"
   adb shell "echo performance >
   /sys/devices/system/cpu/cpu1/cpufreq/scaling_governor"
   adb shell "echo performance >
   /sys/devices/system/cpu/cpu2/cpufreq/scaling_governor"
   adb shell "echo performance >
   /sys/devices/system/cpu/cpu3/cpufreq/scaling_governor"
   ```

If the chipset is Quadcore, perform the same for cpu4, cpu5 , cpu6 and cpu7

5. Buffer transactions between OMX and firmware (ETB/EBD/FTB/FBD) and verify the buffer location.

   `adb shell cat /d/msm_vidc/core0/inst*/info will provide instance details.`

   - Buffer transaction count example

     ```
     ETB Count: 17
     EBD Count: 17
     FTB Count: 17
     FBD Count: 17
     ```

     As per buffer transaction logs, firmware returns all filled buffers (17 FBD) but waits for more at input (ETB 17, EBD 17 = 0) and output (FTB 17, FTB 17).

6. Systrace – Capture video-enabled system trace logs in sys-trace recorder and check for bottlenecks. See http://developer.android.com/tools/help/systrace.html for more details.

2.2 Verify clocks affecting video performance

The video clocks are through debugfs. All clock values are derived from the chipset .dtsi files.

- vcodec

  ```
  /d/clk/venus0_vcodec0_clk
  Eg.  root@sdm660:/d/clk/gcc_venus0_vcodec0_clk # cat rate
  cat rate
  404000000
  ```

NOTE: The example values are from SDM660 chipset.

2.3 Framework limitations for high frame rate video playback

See *Application Note: Achieving Smooth Playback for High Frame Rate Video Scenarios on Android Framework* (80-NL385-1) for more details.

2.4 Performance changes due to slices per frame

Multislicing is set, by default at the framework for ExtendedACodec. For high-resolution clips, each encoded frame may have a large (and unnecessary) number of slices, which causes problems during playback. As the video cores decoding pipeline is re-created for every slice, a higher number of slices causes severe performance issues. QTI recommends fewer than 10 slices per frame.

NOTE: Default multislicing is disabled in ExtendedACodec.cpp

2.5 Modify DDR bus frequency for video encode/decode

The DDR bus frequencies are populated in the chipset specific dtsi file. For SDM660, kernel/msm-4.4/arch/arm/boot/dts/qcom/sdm660-vidc.dtsi file has below entries for video decoder

```
qcom,profile-dec {
                    qcom,codec-mask = <0xffffffff>;
                    qcom,load-busfreq-tbl =
                         <979200 2365000>,  /* UHD30D     */
                         <864000 1978000>,  /* 720p240D   */
                         <489600 1133000>,  /* 1080p60D   */
                         <432000 994000>,   /* 720p120D   */
                         <244800 580000>,   /* 1080p30D   */
                         <216000 501000>,   /* 720p60E    */
                         <108000 255000>,   /* 720p30D    */
                         <0 0>;
          };
```

The first entry indicates the load in macroblocks and the second number indicates the bus frequency corresponding to it. The second argument can be modified based on the use case you are running.

2.6 Capture required debug information

1. Capture kernel and user space logs with the following debug masks:
   a. For hardware codecs

```
adb root;adb remount;
adb shell setprop vidc.debug.level 7
adb shell setprop vpp.debug.level 7 (if vpp enabled)
adb shell "echo 0x103f > /d/msm_vidc/debug_level"
adb shell "echo 0x3F > /d/msm_vidc/fw_level"
adb shell "echo 0 > /proc/sys/kernel/kptr_restrict"
```

   b. For QTI software codecs

```
adb root;adb remount;
adb shell setprop omx_swvdec.log.level 7
```

2. Buffer transactions between OMX to firmware (ETB/EBD/FTB/FBD), and verify the buffer location.

   `adb shell cat /d/msm_vidc/core0/inst*/info will provide instance details.`

   Buffer transaction count example

   ```
   ETB Count: 17
   EBD Count: 17
   FTB Count: 17
   FBD Count: 17
   ```

   As per buffer transaction logs; firmware returns all filled buffers (17 FBD), however waits for more at input (ETB 17, EBD 17 = 0) and at output (FTB 17, FTB 17).

3. Capture media player statistics

   ```
   adb shell setprop persist.debug.sf.stats 1
   ```

4. Systrace – Capture video-enabled system trace logs in systrace recorder and include all events. See http://developer.android.com/tools/help/systrace.html for more details.

   - Alternate command to capture systrace

     ```
     systrace.py gfx input view webview wm am audio video camera hal res
     dalvik sched freq idle disk load sync workq -b 50480 -t 10 -o
     mynewtrace.html
     ```

   The buffer and duration are modified to capture the issue. Ensure that the device is rooted and remounted.

5. Capture user space logs and kernel logs with the following commands:

   ```
   “adb logcat –v threadtime” for collecting userspace(logcat) logs
   “adb logcat –b kernel” for collecting kernel(dmesg) logs
   ```

3 Recording – Quality/Rate control

3.1 Rate control specifications for video core

There are two basic bitrate control strategies used in video, Variable Bitrate Rate (VBR) and Constant Bitrate (CBR).

1. VBR minimizes the frame-by-frame video quality fluctuation and is used in Storage mode.
2. CBR aims to reduce the bitrate fluctuations and is used in real-time communication with channel bandwidth limitation.
3. To maintain bitrate, frames are dropped; this process is Variable Frame Rate (VFR). VBR and CBR are combined with VFR and CFR (Constant Frame Rate, no frame drop). However, bitrate convergence is not guaranteed on CFR modes.

Typical usages are:

1. CBR – Video telephony, Wi-Fi display, adaptive live streaming
2. VBR – Camcorder, Wi-Fi display

The specifications for 30 fps are:

1. CBR_VFR – Bitrate within ±5% of target; converges in 500 ms
2. VBR_VFR – Bitrate within ±10% of target; converges in 10 sec
3. VBR_CFR – Best effort bitrate convergence, no bitrate convergence guaranteed

The frame size is:

1. VBR_CFR – No max limit; depends upon the video input
2. CBR_VFR
    a. I frame is 3 to 6x
    b. P frame is 1 to 4x

3.2 Debug a rate control issue

1. Follow the recommendations in Section 3.1.
2. Capture input/output OMX buffers to analyze bitstream, GOP structure, macroblock QP, and so on, to analyze the rate control content.
   a. Output buffer log

      “adb shell setprop vidc.dec.log.out 1” for hardware codecs
      or
      “adb shell setprop omx_swvdec.dump.op 1” for QTI software codecs

   b. Input buffer log

      “adb shell setprop vidc.dec.log.in 1” for hardware codecs
      Or
      “adb shell setprop omx_swvdec.dump.ip 1” for QTI software codecs

3. Capture OMX IL encoder logs and ensure that the input buffer timestamps are included.

   “adb shell setprop vidc.debug.level 7” for hardware codecs or QTI software codecs

4. Capture kernel logs (video driver and firmware logs).

```
adb shell “echo 0x103F > /d/msm_vidc/debug_level”
adb shell "echo 0x3F > /d/msm_vidc/fw_level"
```

The quality issues are narrowed down to OMX/video core issues or to network problems by analyzing encoder output bitstream.

Capturing input YUV buffers helps QTI to reproduce an issue internally, determine the problem, and provide recommendation. The input buffer timestamp is an input variable for rate control, hence it is important to capture it on OMX IL logs.

3.3 Enable/Disable Perceptual Quantization (PQ)

Perceptual Quantization (PQ) is a new feature in encoder to improve the quality for homogenous region in a frame. Use the following commands to enable/disable PQ:

To enable PQ

```
adb shell setprop vidc.enc.disable.pq 0
```

To disable PQ

```
adb shell setprop vidc.enc.disable.pq 1
```

NOTE: Not all chipsets support PQ. Verify the corresponding chipset related Video Overview Document to confirm if PQ is supported on the chipset or not.

4 Playback/Recording – Video OMX/Driver

4.1 Debug the OMX/driver

The steps to debug issues are as follows.

1. YUV corruption
   a. Disable Universal Bandwidth Compression (UBWC) color format. The UBWC color format is QTI proprietary color format which the other multimedia blocks (like MDSS, GPU) can understand. The UBWC YUV cannot be viewed in standard YUV viewers. To convert the UBWC to standard color format, support from the QTI CE engineers is required.

      ```
      adb shell setprop debug.gralloc.gfx_ubwc_disable 1
      ```

   b. Capture input/output buffer logs to confirm input accuracy and output corruption.
      - Input buffer log

        `"adb shell setprop vidc.dec.log.in 1"` for hardware codecs
        or
        `"adb shell setprop omx_swvdec.dump.ip 1"` for QTI software codecs
      - Output buffer log:

        `"adb shell setprop vidc.dec.log.out 1"` for hardware codecs
        or
        `"adb shell setprop omx_swvdec.dump.op 1"` for QTI software codecs

      - Input buffer log location – /data/misc/media

   c. Pass input/output through video analyzer to verify the bitstream state.
   d. Check the input/output YUV.
   e. Open the YUV file in YUV file viewer and ensure YUV file accuracy (by analyzing visually).
   f. Check for corruption due to video core concealment.
2. Concealment
   - Change the default concealment color if it is not the required one (black is by default), in hardware/qcom/media/mm-video-v4l2/vidc/vdec/src/omx_vdec_v4l2.cpp.
   - adb shell setprop persist.vidc.dec.conceal_color 32784

3. Video does not play.
   a. Confirm bitstream codec specifications.
   b. Check multimedia video chipset overview for chipset compliance.
4. Hang – The device hangs when the core expects input or output buffer.

The following steps help to debug video buffers.

1. Capture buffer transactions between OMX to firmware (ETB/EBD/FTB/FBD), and locate the buffers

   ```
   adb shell cat /d/msm_vidc/core0/inst*/info will provide instance
   details.
   Buffer transaction count eg:
   ETB Count: 17
   EBD Count: 17
   FTB Count: 17
   FBD Count: 17
   ```

   The buffer transaction log lists that firmware returns all filled buffers (17 FBD) but waits for more at input (ETB 17, EBD 17 = 0) and output (FTB 17, FTB 17).

2. Check kernel logs for video msm_vidc errors.

   Example SYS_ERROR kernel logs:

   ```
   <6>[  667.161547] msm_vidc: 4: Opening video instance: efdcd000, 1
   <6>[  667.167882] pil_venus fdce0000.qcom,venus: venus: loading from
   0x07500000 to 0x07a00000
   <6>[  667.283917] pil_venus fdce0000.qcom,venus: venus: Brought out of
   reset
   <7>[  668.021406] msm_vidc: 1: Failed to get reqbufs, -16
   <3>[  677.091935] wlan: [4586:E :SME] sme_QosPmcDeviceStateUpdateInd:
   7354: nothing to process in PMC state 6
   <3>[  677.091944] wlan: [4586:E :TL ]  ----> CRegion 0, hRSSI -49, Alpha
   5
   <3>[  677.092067] wlan: [4600:E :HDD] hdd_conf_arp_offload:
   1003:  fenable = 0
   <3>[  677.092070]
   <3>[  679.277333] wlan: [4586:E :HDD]
   hdd_tx_rx_pkt_cnt_stat_timer_handler: Disable split scan
   <3>[  692.322009] scm_call failed with error code -4
   <7>[  692.322073] msm_vidc: 1: HFI_EVENT_SYS_ERROR: 1, 0x0
   <7>[  692.322091] msm_vidc: 2: SYS_ERROR received for core f5696000
   ```

3. Capture OMX logs with the ADB command `setprop vidc.debug.level` for the hardware codecs and, `setprop omx_swvdec.log.level` for the QTI software codecs

4. Check for poll timeout.

```
Line 170: 01-02 02:15:56.399   297  5820 D OMX-VENC-720p: Poll timedout,
pipeline stalled due to client/firmware ETB: 1873, EBD: 1873, FTB: 1874,
FBD: 1874
```

5. Locate the video buffers.

4.2 Minimal debug information to debug an OMX/driver issue

1. Capture OMX logs.

   "setprop vidc.debug.level 7" for the hardware codecs
   "setprop omx_swvdec.log.level 7" for the QTI software codecs

2. Capture V4L2 logs.

```
echo 100 > /sys/module/videobuf2_core/parameters/debug
```

3. Capture driver logs.

```
adb shell "echo 0x103f > /d/msm_vidc/debug_level"
adb shell "echo 0 > /proc/sys/kernel/kptr_restrict"
```

4. Capture firmware logs.

```
adb shell "echo 0x3F > /d/msm_vidc/fw_level"
adb shell "echo 0 > /proc/sys/kernel/kptr_restrict"
```

5. Capture input/output buffer logs.

```
adb root
adb remount
adb shell chmod 777 /data/misc/media
```

-Encoder

```
adb shell setprop vidc.enc.log.in 1
adb shell setprop vidc.enc.log.out 1
```

-Decoder – Hardware codecs

```
adb shell setprop vidc.dec.log.in 1
adb shell setprop vidc.dec.log.out 1
```

-Decoder – QTI software codecs

```
adb shell setprop omx_swvdec.dump.ip 1
adb shell setprop omx_swvdec.dump.op 1
bitstreams/yuv will be stored at /data/misc/media/
```

6. Capture the number of ETB/EBD and FTB/FBD.

```
adb shell cat /d/msm_vidc/core0/inst*/info
```

4.3 OMX_QCOM Smooth Streaming mode

The OMX_QCOM Smooth Streaming mode is a mechanism that QTI has implemented on the OMX IL decoder to avoid reallocation of buffers in a port reconfiguration event. Hence, it reduces the configuration time for switching between resolutions during the same decoding session. It is useful in streaming scenarios where the same bitstream contains different video frame resolutions.

1. During decoder initialization, the OMX client sets Smooth Streaming mode in the OMX IL decoder.

```
OMX->setParameter(…,
(OMX_INDEXTYPE)OMX_QcomIndexParamEnableSmoothStreaming, ... )
```

2. The OMX client configures the maximum supported resolution for the session.

   If OMX_QCOM Smooth Streaming mode is set, the OMX decoder replies with the worst case buffer requirements.

1. While decoding, if there is a resolution change on the bitstream, OMX IL checks if the current buffer allocation is sufficient according to the bitstream header.

   When the buffer size is insufficient, the next event is sent to `OMX client omx->event (OMX_CORE_OUTPUT_PORT_INDEX, OMX_IndexParamPortDefinition, OMX_COMPONENT_GENERATE_PORT_RECONFIG)`; hence, the buffers are deallocated and reallocated.

2. If the buffer number and size are enough to hold new resolution, the next event is sent to `omx->event (OMX_CORE_OUTPUT_PORT_INDEX, OMX_IndexConfigCommonOutputCrop, OMX_COMPONENT_GENERATE_PORT_RECONFIG);`

3. The OMX client locates the event and index case after step 5 is met.

```
OMX_EventPortSettingsChanged:

} else if (data1 == OMX_CORE_OUTPUT_PORT_INDEX &&
 (data2 == OMX_IndexConfigCommonOutputCrop
```

   a. If an event is detected, the OMX client queries the new resolution from OMX_getConfig.

```
OMX->getConfig(..., OMX_IndexConfigCommonOutputCrop, &rect,
sizeof(rect));
```

b. As no deallocation and reallocation are needed, the OMX client correctly configures the renderer.

```
anw->perform(anw, NATIVE_WINDOW_UPDATE_BUFFERS_GEOMETRY,width,
height, colorFormat);
```

4.4 decode_order for QTI video decoder on Android

The decode_order is set in video telephony and WFD scenarios to reduce the initial frame delay/latency.

In OMXCodec.cpp, the function OMXCodec::configureCodec():

```
Inside if (!strncasecmp(mMIME, "video/", 6)) { ..... }

QOMX_VIDEO_DECODER_PICTURE_ORDER picture_order;
picture_order.nPortIndex = 1;
picture_order.eOutputPictureOrder = QOMX_VIDEO_DECODE_ORDER;
ALOGV("\nSet picture order\n");
if(mOMX->SetParameter(mNode,

(OMX_INDEXTYPE)OMX_QcomIndexParamVideoDecoderPictureOrder,
(OMX_PTR)&picture_order, sizeof(picture_order)) != OMX_ErrorNone)
{
ALOGV("\n ERROR: Setting picture order!");
return -1;
}
```

4.5 Full-seg demux buffer overflow

In the full-seg live stream, depending on the customers AV sync policy and the initial delta between PCR and video PTS, there could be a buffering period of ~0.5 sec between demux and video output. If the OMX input buffer number is not increased, then the demux overflows in video and a few video frames are dropped.

- For Demux buffer overflow
  - Call DMX_GET_EVENT IOCTL command results in the DMX_EVENT_BUFFER_OVERFLOW event.
  - Log output is `dmxdev: buffer` overflow

To avoid demux buffer overflow, OMX input buffer number is increased as follows.

```
---
OMX_GetParameter(dec_handle, (OMX_INDEXTYPE)OMX_IndexParamPortDefinition,
&portFmt);
portFmt.nBufferCountActual = XX; //XX is input buffer number, Maximum is 32
OMX_SetParameter(dec_handle, (OMX_INDEXTYPE)OMX_IndexParamPortDefinition,
&portFmt);

---
```

5 Playback/Recording – Stability

5.1 Required debug information

NOTE: The responses to the following queries must be shared with QTI to reproduce and debug stability issues.

1. Clearly state reproduction rate and steps.
   - Is there any dependency on OEM/third parties that prevents reproduction at QTI? The information shared with QTI helps reproduction on the MTP.
2. Capture RAM dumps. The RAM dump procedure varies with OEMs.
3. Collect Venus firmware RAM dumps, as follows:

   ```
   chmod 664 /dev/ramdump_venus
   echo 1 > /sys/module/subsystem_restart/parameters/enable_ramdumps
   ./system/bin/subsystem_ramdump
   ```

   This reproduces the crash issue.

   The crash can also be triggered with `echo 3 > /sys/kernel/debug/msm_vidc/core0/trigger_ssr`.

   RAM dump is stored in internal memory at /data/ramdump or in the SD card at /sdcard/ramdump.
4. Collect driver logs.

   ```
   adb shell
   cd /d/msm_vidc
   su
   echo 0x1015 > debug_level
   ```

5. Collect firmware logs.

   ```
   echo 0x3F > fw_level
   ```

5.2 Reasons for Venus response stops

Venus stops responding for the following reasons:

- Buffer starvation

  Venus hangs while waiting for input and/or output buffers. Buffer transaction logs can confirm. `adb shell cat /d/msm_vidc/core0/inst*/info` provides instance details.

- Buffer transaction count example

```
ETB Count: 17
EBD Count: 17
FTB Count: 17
FBD Count: 17
```

  As per buffer transaction logs, firmware returned all filled buffers (17 FBD) but waits for more at input (ETB 17, EBD 17 = 0) and at output (FTB 17, FTB 17).

  Hence, the Venus core crashes. The firmware logs and RAM dumps help debug the issue.

```
adb shell
cd /d/msm_vidc
su
echo 0x1015 > debug_level
```

- Firmware logs

```
echo 0x3F > fw_level
```

6 Playback/Recording – Video framework

6.1 Debug a video framework playback issue

1. Confirm with QTI if the Android software stack supported feature can be enabled.
2. Verify whether QTI parser is used. QTI parser supports most container format, and QTI has tested only with QTI parser.

```
adb shell setprop mm.enable.qcom_parser 37491
adb shell getprop mm.enable.qcom_parser
```

3. Disable audio from the scenario; it helps narrow down the issue.

"adb shell setprop persist.debug.sf.noaudio 1″ for playback use case
"adb shell setprop persist.debug.sf.noaudio 3″ for recording use case

6.2 Debug a video framework recording issue

1. Confirm that the feature supports the Android software stack.
2. Disable audio from the scenario and verify if the issue repeats.

```
adb shell setprop persist.debug.sf.noaudio 3
```

6.3 Minimal debug information to debug a framework issue

Enable the use case-related logs on an Android framework, e.g., for a scenario where data flow passes through ACodec, enable #define LOG_NDEBUG 0 in /frameworks/av/media/libstagefright/ACodec.cpp.

```
01-02 00:01:09.349   213   213 I ACodec: [OMX.qcom.video.decoder.avc] AVC
profile = 100 (High), level = 40
01-02 00:01:09.349   213   213 I ExtendedACodec: Thumbnail mode enabled.
01-02 00:01:09.349   213   213 E OMX-VDEC-1080P: Set Resolution failed
```

6.4 Increase camera source buffers

The camera source buffers are introduced to debug performance issues during recording. The ETB/EBD transaction logs are captured and checked if bottle neck is on encoder input.

File: CameraSource.cpp

```
status_t CameraSource::start(MetaData *meta)
```

Increase the buffer count for mNumInputBuffers

6.5 Disable audio during video recording

The audio recording is disabled to narrow down the root cause while debugging recording video issues.

```
adb shell setprop persist.debug.sf.noaudio 3
```

6.6 Support for MPEG-4 data partition on Venus-based chipsets

Venus core does not support MPEG-4 clips with data partition, but Google software decoders support these clips.

QTI has modified AOSP to parse the VOL header passed from QC parser to ACodec through kKeyRawCodecSpecificData for the data partition bit and fall back to the software codec (if found).

6.7 QC-mm parser usage

QTI recommends QC parser for container formats that are not supported in the AOSP parser.

Refer to corresponding chipset Video Overview documents for supported file format matrix information.

6.8 Enable QC parser

The procedure for enabling the QC core supported parser is as follows.

```
-temporal-
adb root
adb shell setprop mm.enable.qcom_parser 37491
adb shell getprop mm.enable.qcom_parser

-permanent-
change mm.enable.qcom_parser in devices system.prop
```

6.9 Frame-by-Frame and Arbitrary modes

- Frame-by-Frame – OMX_QCOM_FramePacking_OnlyOneCompleteFrame; single frames are read by the parser in single chunks by looking at the container metadata. The parser then pushes one single full frame to the decoder. This helps save power by avoiding bitstream parsing when looking for start codes. The QTI hardware decoder works on a per-frame basis.
- Arbitrary – OMX_QCOM_FramePacking_Arbitrary; in streaming scenarios a chunk of stream is read and pushed to the decoder. The chunk can have many frames, single or partial frame. The QTI hardware decoder works on a per-frame basis, so the OMX IL decoder needs to assemble frame-by-frame by parsing the bitstream looking for start codes. After a frame is assembled, it pushes down to the video hardware.

6.10 Codecs/containers supported on Frame-by-Frame and Arbitrary modes

The OMX IL decoder component is set in Frame-by-Frame mode, OMX_QCOM_ FramePacking_OnlyOneCompleteFrame. Frame-by-Frame mode is preferable to avoid parsing the input bitstream. Arbitrary mode-only, (OMX_QCOM_FramePacking_Arbitrary, is used for specific container and codec formats.

The component is set to Arbitrary mode for the following video codecs.

Container	Video format
AVI\DIVX\XVID	MPEG-4\Divx (4\5\6)
ASF\WMV	VC1-AP

The remaining codecs are in Frame-by-Frame mode.

6.11 QTI AVEnhancements

The QTI video team moved the Android Open Source Project (AOSP) changes and customizations to a separate proprietary project (avenhancements) for better maintainability and ease of AOSP upgrades.

The AOSP code contains hooks and minor structural modifications. Customization is performed in extended classes. Because the customization code resides in a separate project, and the changes are not required to be carried over for every release.

The Avenhancements project is released as a binary format (libavenhamcements.so).

Enable logs in avenhancement project

```
adb shell setprop persist.debug.av.logs.level 3
```

Enable Stats from avenhancement project

```
adb shell setprop persist.debug.sf.extendedstats 1
```

Features are compiled in only if ENABLE_AV_ENHANCEMENTS is defined. Otherwise implementations are bypassed.

See *Migration Plan for AOSP Customization* (80-NV396-68) for more details on avenhancement implementation design.

6.12 Disable audio during video playback

```
adb shell setprop persist.debug.sf.noaudio 1
```

6.13 Dynamic resolution

See *Seamless Resolution on Android Decoder* (80-NK913-1) for more details.

6.14 Disable/Enable DivX

- Due to licensing constraints, Divx is disabled by default.
- If an OEM wants to enable Divx, see KBA-170130215910 at https://createpoint.qti.qualcomm.com/

6.15 Change a codec profile and enable/disable B frames during encoding on Android

B frames are enabled during encoding when the codec profile is higher than:

- OMX_VIDEO_MPEG4ProfileSimple for MPEG-4 encoding
- OMX_VIDEO_AVCProfileBaseline for AVC encoding

ACodec sets B frames by calling OMX_SetParameter() to the OMX QTI IL component with respecting index OMX_IndexParamVideoMpeg4 or OMX_IndexParamvideoAvc with nBFrames listing the number of B frames within the GOP structure.

The profiles for QTI releases are set by setprop.

```
adb shell setprop encoder.video.profile [simple][asp][main][high]
by default baseline and simple profiles are default, therefore B frames are
disable.
```

7 Playback/Recording – Firmware

7.1 Debug a firmware issue

1. Confirm that firmware is correctly integrated into the build environment.

   QCT releases unsigned elf Venus binary (venus.mbn). OEMs should sign, split, and ship under /etc/firmware/. Firmware will be authenticated before loading on secure devices.

   a. Sign venus.mbn. See *Application Note: Enable Secure Boot on APQ8084, MSM8974, MSM8x26, MSM8x10, and MSM8x12 Chipsets* (80-NA157-20).

   b. Split

      i Input mbn equal to elf. b00 to b04 is generated.

      ii Usage – android\vendor\qcom\proprietary\common\scripts\pil-splitter.py <elf> <prefix>

   c. Load all venus.b00, venus.b01, venus.b02, venus.b03, venus.b04, venus.mbn, and venus.mdt files to the device at /etc/firmware.

2. Confirm that video firmware is successfully loaded when run.

```
<6>[   95.389889] subsys-pil-tz 1de0000.qcom,venus: venus: loading from
0x8f100000 to 0x8f600000
<6>[   95.459730] subsys-pil-tz 1de0000.qcom,venus: venus: Brought out
of reset
```

3. Capture kernel logs with the following log mask if the device crashes

   Usage: `adb shell “echo 0x103F > /d/msm_vidc/debug_level”, adb shell "echo 0x3F > /d/msm_vidc/fw_level"`

7.1.1 Minimum information required to debug a firmware issue

- Firmware logs
  - Usage – adb shell "echo 0x3F > /d/msm_vidc/fw_level"
- RAM dumps (if required)

  QPST automatically captures the dumps. vmlinux with symbols is shared along with RAM dumps. The OEM needs to provide the Venus loading address found in the kernel log.

```
<6>[ 95.389889] subsys-pil-tz 1de0000.qcom,venus: venus: loading from
0x8f100000 to 0x8f600000
```

7.2 Calculate Venus encoding/decoding processing frame time duration for each frame

1. Enable firmware profile logs 0x5c (fw_level) and capture the kernel logs.

```
adb shell
cd /d/msm_vidc
su
echo 0x1000 > debug_level
echo 0x5c > fw_level
```

2. The firmware logs print for each frame VPP start to VPP end (values in hex).

   (End time - start time)/Venus processor speed

3. Kernel logs

```
Line 5443: <7>[  726.526355] [3: kworker/u:3:  164] msm_vidc: 4096: FW-
SAYS: FW H264e: Session e4425380 Frame 5 VPP start f21f4407 VPP end
f13e91a3.
```

   - Number of cycles – 4062135303(hex f21f4407) - 4047409571(f13e91a3) = 14725732
   - Number of ms – 14725732/456000(core frequency*1000) = 32 ms
     - 32 ms is the frame processing time for the frame.
     - 456000 – Venus running frequency

7.3 Venus power collapse overview

Venus power collapse is an additional power gain feature that further reduces power consumption apart from interframe clock gating. Power collapse is initiated if the Venus core is idle for more than the intended time (say 10 sec).

- Typical use case
  a. Play the local clip from the gallery.
  b. Perform a pause operation.
  c. Perform a resume operation.

  When the delay between Steps b and c increases, the Venus core is virtually switched off (as there is no pending task) and the following events take place to power collapse the Venus core.
- Typical power collapse sequence
  a. Receive HFI_MSG_SYS_IDLE from the video core.
  b. Disable VCODEC_CLK and queue delayed work to venus_hfi_pm_work with msm_ vidc_pwr_collapse_delay (typical value in QTI releases is 10 sec).

c. Assume there is no further transaction within the next msm_vidc_pwr_collapse_delay seconds (like PAUSE operation), then venus_hfi_pm_work is triggered and sends the HFI_CMD_SYS_PC_PREP packet to the Venus core.

d. The driver waits for HFI_MSG_SYS_PC_PREP_DONE from the core.

e. Upon receiving HFI_MSG_SYS_PC_PREP_DONE, call TZBSP_VIDEO_STATE_ SUSPEND, detach the IOMMU group, disable the regulator, and unvote the bus for DDR memory. This successfully completes the Venus power collapse.

f. Upon Resume, scale the DDR bus, enable the regulator, attach the IOMMU group, enable VCODEC_CLK, call TZBSP_VIDEO_STATE_RESUME, and reprogram VBIF/QoS registers.

g. If a transaction (ftb/etb/command) needs to be sent to the video core within msm_vidc_pwr_collapse_delay (between steps b and c), cancel the work to venus_hfi_pm_work and enable VCODEC_CLK and continue processing.

Power collapse delay is a configurable parameter as shown here:

```
kernel/drivers/media/platform/msm/vidc/msm_v4l2_vidc.c
uint32_t msm_vidc_pwr_collapse_delay = 10000;
```

- Logs

The following messages confirm Venus is in Power Collapse mode.

```
                Line 156435: <7>[  379.625249] msm_vidc: 8: Received
HFI_MSG_SYS_IDLE --> Step a
                Line 156499: <7>[  389.648021] msm_vidc: 8: Received
HFI_MSG_SYS_PC_PREP_DONE --> Step e
                Line 156513: <7>[  389.649871] msm_vidc: 4: entering
power collapse --> Step f
```

- Intended clocks
  - Check clock value constantly; usage – adb shell "cat /system/debug/clk/venus0_vcodec0_clk/measure"
  - Venus clock resets to 0 even before Venus power collapse since clocks are disabled as part of interframe clock gating.
- Power gain contributors – Clocks, regulator, IOMMU, TZ, bus unvoting

7.4 PIL messages while loading firmware images

- Invalid firmware metadata
  - Definition – The blob containing the metadata (the <image>.mdt file) that failed authentication
  - Impact – PIL will fail to load the image and the subsystem will not be booted
  - Sample kernel log message

```
<3>[ 89.122212] pil_venus fdce0000.qcom,venus: venus: Invalid
firmware metadata
```

- Check – Ensure that the .mdt file is valid, follow up with the TZ team and with image owners. This is usually a TZ authentication issue.

- Brought out of reset
  - Definition – Reset is a hardware term that implies a processor is not running (not executing instructions). Out of reset therefore means that the processor has begun execution.
  - Impact – None; expected message
  - Sample kernel log message

```
<6>[ 807.528312] pil_venus fdce0000.qcom,venus: venus: Brought out of
reset
```

  - Check – None; expected message

- Failed to locate blob <string>
  - Definition – One of the blobs or parts (the files named <image>.mdt, .b01, etc.) of the ELF image was not located on the file system, or the request to load the blob failed.
  - Impact – PIL fails to load the image and the subsystem is not booted.
  - Sample kernel log message

```
[39.966721] pil_venus fdce0000.qcom,venus: venus: Failed to locate
blob venus.b02 or blob is too big
```

  - Check – Ensure that the blob file is present on the file system and that the symlinks are set up correctly in the /etc/firmware folder.

- Failed to allocate relocatable region
  - Definition – Ion API to allocate physical memory for a relocatable segment failed.
  - Impact – PIL will fail to load the image and the subsystem will not be booted.
  - Sample Kernel log message

```
[53.274120] pil_venus fdce0000.qcom,venus: venus: Failed to allocate
relocatable region
```

  - Check – The Linux memory and debug team is responsible for Ion. Ensure that the Ion entries for heaps to be used by PIL in the device tree or platform data are correct.

7.5 Load Venus firmware

Firmware is stored on the device in flash memory at /etc/firmware/venus*. The firmware consists of various split files, i.e., venus.b00, venus.b01, venus.b02, venus.b03, venus.b04, venus.mdt.

- Firmware download

  The following instructions are performed in the subsystem_get() API (part of PIL).

  a. Copy the firmware binaries from the flash memory location (/etc/firmware/venus*) onto DDR memory to /system/etc/firmware/.
  b. Authenticate the firmware (in case of secure PIL).
  c. Bring Venus out of reset.

The PIL location is shared with other images. If a Venus PIL request comes before WCNSS/PR PIL, then the PIL driver will load Venus firmware at the base PIL address location. Otherwise, Venus firmware will be loaded at the highest available location.

The PIL memory map is usually on mentioned chipset release notes, e.g., 80-NM886-1_B_M8916AAAAANLYD1005.pdf.

The Venus firmware is relocatable from the Android KitKat release (relocatable firmware - Memory is allocated when request for firmware download is made). The Venus firmware will be loaded in a relocatable region of approximately 29 MB. The relocatable region is shared by WCNSS, ADSP, and Venus. The size of Venus is approximately 5 MB. PIL will allocate the memory based on task entry, e.g., if ADSP comes first, then it will allocate memory for ADSP followed by Venus or WCNSS.

7.6 Video core firmware debugging

The Venus core firmware provides the following debug utilities:

- Venus Subsystem Failure Reason (SFR) messages
- Firmware debug log in the kernel log
- Venus firmware memory dump

7.6.1 Analyze SFR messages

Use Table 7-1 and Table 7-2 to determine the failure reason that prompted the error message.

Table 7-1 Venus core firmware SFR messages

Error types	Error messages	Error conditions
Stray IRQ	default_ISR %x,%x,%x (ISR information)	Spurious interrupt
AHBVBIF error	Err_Fatal - VenusCoreCtrl_AhbVbifErrorIRQ	External memory access issue
Stack overflow	Exception: Stack overflow around sp = 0x%x (stack pointer)	Software stack overflow
Software fatal error	Err_Fatal - %s:%d: (file name & line number)	Assertion
Exceptions	Exception: TID = Unknown IP = 0x%x FA = 0x%x cause = 0x%x	See cause codes in Table 7-2

Table 7-2 SFR message cause codes

Cause code	Cause	Instruction pointer	Fault address
0x04	Undefined instruction	Error PC	Fault address
0x0C	Instruction abort	Error PC	Error PC
0x10	Data abort	Error PC	Fault address
0x14	FIQ (CPU WD bite)	0x0	0x0
0xFFFFFFFF	unknown	0x0	0x0

For example, if you force a Venus memory SSR dump via the ARM9 CPU watchdog timeout with the following command:

```
echo 3 > /sys/kernel/debug/msm_vidc/core0/trigger_ssr
```

Venus generates an SFR message similar to the following:

```
Crashinfo: SFR Message from FW : Exception: TID = Unknown IP = 0x0 FA = 0x0
cause = 0x14
```

The 0x14 Cause Code in the message refers to the ARM9 CPU watchdog timeout, which correctly identifies the action initiated.

7.6.2 Enable the Venus firmware debug log in kernel log

The firmware debug log helps you analyze a crash from use cases and the last captured Venus video core firmware process.

Use `debugfs` to enable to firmware debug log in the kernel log, as follows:

```
adb root
adb wait-for-devices
adb shell “mount -t debugfs none /d”
adb shell “echo 0x1000 > /d/msm_vidc/debug_level”
adb shell “echo 0x3F > /d/msm_vidc/fw_level”
```

The following list defines the debug log levels:

```
#define  HFI_DEBUG_MSG_LOW      0x00000001 /**< Low level messages. */
#define  HFI_DEBUG_MSG_MEDIUM   0x00000002 /**< Medium level messages. */
#define  HFI_DEBUG_MSG_HIGH     0x00000004 /**< High level messages. */
#define  HFI_DEBUG_MSG_ERROR    0x00000008 /**< Error messages. */
#define  HFI_DEBUG_MSG_FATAL    0x00000010 /**< Fatal messages. */
#define  HFI_DEBUG_MSG_PERF     0x00000020 /**< messages containing
performance data */
```

7.6.3 Access the Venus firmware memory dump

The firmware triggers a memory dump during any of the following:

- Venus watchdog
- Assertion
- Exception
- Stray interruption

1. Access the memory dump via Subsystem Restart (SSR). For example:
   - To trigger a fatal system error:
     ```
     # echo 1 > /d/msm_vidc/core0/trigger_ssr
     ```
   - To trigger a watchdog bite:
     ```
     $echo 3 > /d/msm_vidc/core0/trigger_ssr
     ```
2. Find the ramdump at: `/data/ramdump`.
3. Forward the ramdump to QTI for analysis. QTI can recover the call stack and registers and can use proprietary ELF information for a deeper analysis.

8 CTS/GTS/Compliance

8.1 Debug GTS/CTS/Compliance tests issue

1. Perform faulty test individually and repeatedly as occasionally the problem is with test suite automation.
2. Note the test suite version.
3. Inform the step 1 and step 2 results to the QTI CE engineer.

8.2 Required debug information to debug GTS/CTS/Compliance tests

1. Capture OMX logs.

   `setprop vidc.debug.level 7` for hardware codecs
   or
   `setprop omx_swvdec.log.level 7` for QTI software codecs
   `OMX input/output buffer logs`

2. Capture output buffer logs.

   `adb shell setprop vidc.dec.log.out 1` for hardware codecs
   or
   `adb shell setprop omx_swvdec.dump.op 1` for QTI software codecs

3. Capture input buffer logs.

   `adb shell setprop vidc.dec.log.in 1` for hardware codecs
   or
   `adb shell setprop omx_swvdec.dump.ip 1` for QTI software codecs

8.3 CTS/XTS/GTS known failures

See *Application Note: Known CTS Issues* (80-NR633-1) for more details.

9 Streaming

9.1 Streaming protocols supported in Android

The following network protocols are supported for audio and video playback as per the Android website at http://developer.android.com/guide/appendix/media-formats.html.

- RTSP (RTP, SDP)
- HTTP/HTTPS progressive streaming
- HTTP/HTTPS live streaming – MPEG-2 TS media files only Protocol ver 3 (Android 4.0 and above), Protocol ver 2 (Android 3.x) not supported before Android 3.0

NOTE: HTTPS is not supported before Android 3.1. See *Presentation: Qualcomm MPEG-Dash Solution Overview* (80-NM976-1) for more information on the MPEG-DASH.

9.2 Debug guide for RTSP streaming issues on Android

9.2.1 Debug a streaming issue

Capture the bitstream at the OMX IL level and feed offline to reproduce and narrow down issues to lower layers.

1. Capture input bitstream by setting *adb shell setprop vidc.dec.log.in*. This generates dump at /data/misc/media/.
2. Feed input bitstream to the OMX test app.

```
adb shell mm-vdec-omx-test <path-to-input-bitstream> follow/select steps
provided in command window           eg: #mm-vdec-omx-test 1.264 1 2 0
1 0 0 0 --> arbitrary mode for H264 bitstream and no YUV captured
#mm-vdec-omx-test 1.264 1 2 2 1 0 0 0 --> arbitrary mode for H264
bitstream and to capture YUV
```

   a. Capture a PCAP log and confirm that it is not a network-related issue.

   b. Open PCAP using Wireshark.

   c. Navigate Telephony > RTP > Show All Streams.

   d. Observe the “Lost packets” Column for RTPType-96 and RTPType-97. If it is more %, then it could be a network issue.

9.3 Minimum debug information required to debug a streaming issue

- PCAP logs
  - Usage – `adb shell tcpdump -i any -p -s 0 -w /data/op_pcap.pcap`
- Elementary *bitstream* – Setting the vidc.dec.log.in to 1 enables bitstream dump; dump is available at /data/misc/media
  - Usage – `adb shell setprop vidc.dec.log.in 1`
- OMX logs – Setting the log masks below enables detailed OMX Video component logs
  - Usage – `adb shell setprop vidc.debug.level 7`
- Kernel logs – Setting the log masks below captures detailed kernel logs
  - Usage – `adb shell "echo 4127 > /d/msm_vidc/debug_level"`
- RTSP stack logs – Add the following lines in the given source files to enable debug messages:
  From: //#define LOG_NDEBUG 0
  To: #define LOG_NDEBUG 0
  a. NuPlayer.cpp
  b. NuPlayerRenderer.cpp
  c. MyHandler.h
  d. ACodec.cpp
  e. RTSPSource.cpp
  f. ARTPConnection.cpp

10 Content protection playback

10.1 Debug a content protection issue

- Corruption – HLOS is not able to save OMX input-output logs as per CP. The encrypted content is captured and decrypted with an offline tool. Check for corruption after it passes the bitstream through OMX IL.
- Crash – RAM dumps indicate that the address firmware is loaded so QTI can dump firmware memory and analyze offline. Look for "scm", "msm_vidc" errors and failures on kernel logs.

  scm and msm_vidc errors examples are:

```
<6>[  667.161547] msm_vidc: 4: Opening video instance: efdcd000, 1
<6>[  667.167882] pil_venus fdce0000.qcom,venus: venus: loading from
0x07500000 to 0x07a00000
<6>[  667.283917] pil_venus fdce0000.qcom,venus: venus: Brought out of
reset <7>[  668.021406] msm_vidc: 1: Failed to get reqbufs, -16
<3>[  677.091935] wlan: [4586:E :SME] sme_QosPmcDeviceStateUpdateInd:
7354: nothing to process in PMC state 6
<3>[  677.091944] wlan: [4586:E :TL ]  ----> CRegion 0, hRSSI -49, Alpha
5 <3>[  677.092067] wlan: [4600:E :HDD] hdd_conf_arp_offload: 1003:
fenable = 0  <3>[  677.092070]
<3>[  679.277333] wlan: [4586:E :HDD]
hdd_tx_rx_pkt_cnt_stat_timer_handler: Disable split scan
<3>[  692.322009] scm_call failed with error code -4
<7>[  692.322073] msm_vidc: 1: HFI_EVENT_SYS_ERROR: 1, 0x0
<7>[  692.322091] msm_vidc: 2: SYS_ERROR received for core f5696000
```

10.2 Minimal debug information required to debug a content protection issue

1. Capture OMX IL logs.

```
setprop vidc.debug.level 7
```

2. Capture driver logs.

```
adb shell
cd /d/msm_vidc
su
echo 0x1015 > debug_level
```

3. Capture firmware logs.

```
cho 0x3F > fw_level
```

4. Capture TZ logs.

```
adb shell cat /sys/kernel/debug/tzdbg/log
```

5. Capture RAM dumps (crash).

   When a device crashes, RAM dumps are automatically generated else a crash is induced.

   A CPU RAM crash is triggered through `adb shell "echo c > /proc/sysrq-trigger"`. RAM dumps are stored in the OEM-specific location.

10.3 Enable Widevine Level 3 secure playback on the MSM8916 chipset

The MSM8916 chipset supports Widevine Level 1 secure playback by default. To enable Widevine level 3, follow these steps to enable Widevine Level 3 Secure Playback before building the source code.

1. Comment all the lines in vendor\widevine\proprietary\drmwvmplugin\oemcryptolevel.mk except "LOCAL_OEMCRYPTO_LEVEL := 3".
2. Use "LINUX\android\vendor\qcom\proprietary\prebuilt_<xxx>\target\product\<model>\obj\STATIC_LIBRARIES\liboemcrypto_intermediates\liboemcrypto.a" to generate libdrmwvmplugin.so, libwvm.so, and libdrmdecrypt.so.
3. Ensure that libWVStreamControlAPI_L3.so and libwvdrm_L3.so to /system/vendor/lib are pushed. Also, remove libWVStreamControlAPI_L1.so and libwvdrm_L1.so if they existed in /system/vendor/lib.

11 Broadcast

11.1 ISDB-T (Full/One-seg) TS clips types

ISDB-T (Full/One-seg) TS clips that conform to the following conditions are played back with MTP/CDP.

- Clip should not be scrambled
- TS packet size should be 188 bytes (not 192 bytes with timestamp)
- Clip should have valid PAT, PMT table
- Clip should have a valid Video I-frame entry
- An H.264 codec clip should have a valid SPS/PPS entry
- Seek will be supported only if valid key frame presents in segment

11.2 ISDB-T TCR hardware counter frequency for MSM8974/MSM8994

TSIF Clock Reference (TCR) is used to take a 4 byte timestamp for each Broadcasting TS packet to prepare network streaming. For MSM8974, tsif_ref_clk (TCR hardware counter-frequency) is configured as follows.

- In "kernel/arch/arm/mach-msm/clock-8974.c"

```
---
static struct clk_freq_tbl ftbl_gcc_tsif_ref_clk[] = {
            F(105000,    cxo,   2,   1,  91),
            F_END
};
---
```

It results in tsif_ref_clk = 19.2 MHz/2x(1/91) = 105.4945 kHz.

For MSM8994, tsif_ref_clk is configured as follows.

- In file "kernel/drivers/clk/qcom/clock-gcc-8994.c"

```
---
static struct clk_freq_tbl ftbl_tsif_ref_clk_src[] = {
              F(    105500,         gcc_xo,    1,    1,   182),
              F_END
};
---
```

It results in Tsif_ref_clk = 19.2 MHz/1x(1/182) = 105.4945 kHz.

11.3 ISDB-T support

The following chips have hardware capability for ISDB-T:

- MSM8996
- MSM8994
- MSM8992
- MSM8974
- MSM8x62
- APQ8084
- APQ8064

For software implementation, see *Application Note: OpenMAX IL Video Decoder for ISDB-T Full-Seg on MSM8974 and APQ8064* (80-NE866-1) for information about ISDB-T feature implementation responsibilities between QTI/OEM (or third party).

- Frontend (tuner/demo) implementation is the OEM's responsibility.
- QTI provides the lower driver layer for descramble and demux functionalities.
- All above layers for ISDB-T stack implementation must be done by the OEM or their third-party partners.

The ISDB-T driver's availability for each chip must be checked case-by-case.

A OMX decoder

A.1 Run an OMX decoder test app on Android releases

The OMX decoder test app can be executed with the command line options on Android devices. It takes video bitstream as input and writes the data in the file.

Test app users can use adb shell to check if "mm-vdec-omx-test" is under /system/bin or not. If it does not exist, the user can run the following command to copy it to /system/bin.

```
adb push mm-vdec-omx-test /system/bin
```

mm-vdec-omx-test can be obtained from android\out\target\product\[platform]\system\bin after successful compilation of the build.

Currently, the OMX decoder test app accepts only video bitstream as input which should be extracted before from the file container.

```
/////////////// Sample for H.264 bitstream in Arbitrary
Mode/////////////////
adb shell
# mm-vdec-omx-test /data/input.264
mm-vdec-omx-test /data/input.264
Command line argument is available
To use it: ./mm-vdec-omx-test <clip location> <codec_type>
           <input_type: 1. per AU(.dat), 2. arbitrary, 3.per NAL/frame>
           <output_type> <test_case> <size_nal if H264>
****************************************
 ENTER THE TEST CASE YOU WOULD LIKE TO EXECUTE
 ****************************************
 1--> H264
 2--> MP4
 3--> H263
 4--> VC1
 5--> DivX
 6--> MPEG2
 7--> VP8
 8--> HEVC
 9--> HYBRID
 10-> MVC
1
1
```

```
 ******************************************
 ENTER THE TEST CASE YOU WOULD LIKE TO EXECUTE
 ******************************************
 1--> PER ACCESS UNIT CLIP (.dat). Clip only available for H264 and Mpeg4
 2--> ARBITRARY BYTES (need .264/.264c/.m4v/.263/.rcv/.vc1/.m2v)
 3--> NAL LENGTH SIZE CLIP (.264c)
 4--> START CODE BASED CLIP (.264/.h264)
2
2
 ******************************************
 Output buffer option:
 ******************************************
 0 --> No display and no YUV log
 1 --> Diplay YUV
 2 --> Take YUV log
 3 --> Display YUV and take YUV log
2
2
 ******************************************
 ENTER THE TEST CASE YOU WOULD LIKE TO EXECUTE
 ******************************************
 1 --> Play the clip till the end
 2 --> Run compliance test. Do NOT expect any display for most option.
       Please only see "TEST SUCCESSFULL" to indicate test pass
 3 --> Thumbnail decode mode
1
1
 ******************************************
 ENTER THE COLOR FORMAT
 0 --> Semiplanar
 1 --> Tile Mode
 ******************************************
0
0
 ******************************************
 Output picture order option:
 ******************************************
 0 --> Display order
 1 --> Decode order
0
0
 ******************************************
 Number of frames to decode:
 0 ---> decode all frames:
 ******************************************
0
0
Input values: inputfilename[/data/input.264]
```

```
 Inside OMX_GetComponentsOfRole

**************************************
**********...TEST SUCCESSFULL...*********
**************************************
```

YUV dump is generated at the same place where the test is executed. In case the test is not successful, the OEM needs to provide logcat and kernel logs with the following log mask and also share the input bitstream used

```
1. adb root;adb remount;
2. adb shell setprop vidc.debug.level 7
3. adb shell "echo 4127 > /d/msm_vidc/debug_level"
4. adb shell "echo 0x3F > /d/msm_vidc/fw_level"
```

Alternatively, OEMs can refer to Section 4.2 for more details on debugging.

A.2 Android OMX encoder config test app

The command to run the config test app from adb shell is:

```
mm-venc-omx-test ENCODE config.cfg 1
```

The following is a sample config file:

```
################################################
############ COMMON STATIC CONFIG ############
################################################
FrameWidth = 176                ## Frame width
FrameHeight = 144               ## Frame height
OutputFrameWidth = 176          ## DVS Frame width
OutputFrameHeight = 144         ## DVS Frame height
DVSXOffset = 0                  ## DVS Frame width
DVSYOffset = 0                  ## DVS Frame height
Codec = H264                    ## MP4 | H263 | H264 | VP8
Profile = H264_HIGH             ## MPEG4_SP| MPEG4_ASP | H263_BASELINE |
H264_BASELINE | H264_MAIN | H264_HIGH | VP8_MAIN
Level = DEFAULT                 ## DEFAULT | VP8_VERSION_0 | VP8_VERSION_1
FPS = 30                        ## Frame rate
RC = RC_VBR_CFR                 ##
RC_OFF  |  RC_VBR_VFR  |  RC_VBR_CFR  |  RC_CBR_VFR
InBufferCount = 9               ## Number of input buffers
OutBufferCount = 5              ## Number of output buffers
Rotation = 0                    ## Rotation in degrees 0 | 90 | 180 | 270
Bitrate = 128000                ## Bits per second
ResyncMarkerType = NONE         ## NONE | BITS | GOB | MB
```

```
ResyncMarkerSpacing = 0          ## Resync marker spacing (valid only if
resyncmarkertype != none)
IntraRefreshMBCount = 10         ## Intra Refresh MB count ( valid if less
than total number of MB's in a frame)
InFile =  /sdcard/Encoder/qcif.yuv  ## Input yuv file
OutFile = /sdcard/Encoder/qcif.264  ## Output bitstream file (leave blank
for no output)
NumFrames = 100                  ## Number of frames to encode
IntraPeriod = 50                 ## The iframe interval in units of frames
MinQp = 2                        ## The minimum qp
MaxQp = 51                       ## The maximum qp
ProfileMode = 0                  ## 0 file mode | 1 profile mode
DynamicFile = /sdcard/Encoder/dynamiccfg.scn ## list of dynamic
configurations to be exercised
Extradata = 0                    ## 0 disable| 1 enable
IDRPeriod = 0                    ## 0 => 1 IDR/session, 1 => every I frame
= IDR frame, 2 => every 2nd I frame is an IDR frame....
################################################
############  MPEG4 STATIC CONFIG ############
################################################
HECInterval = 0                  ## Header extension coding interval (0 for
disable)
TimeIncRes = 30                  ## MPEG4 time increment resolution
EnableShortHeader = 0            ## MPEG4 SVH enable

################################################
############  H.263 STATIC CONFIG ############
################################################

################################################
############  H.264 STATIC CONFIG ############
################################################

CABAC = 1 ##  0 CAVLC | 1 CABAC (valid for MAIN/HIGH Profile)
Deblock = 1 ## 0 DISABLE | 1 ENABLE_ALL | 2 DISABLE_SLICE_BOUNDARY

################################################
########### COMMON DYNAMIC CONFIG ############
################################################
IFrameRequestPeriod = 5          ## Dynamically request an iframe every N
frames
UpdatedFPS = 15                  ## The new frame rate for change quality
UpdatedBitrate = 128000          ## The new bitrate for change quality
UpdatedNumFrames = 90            ## The new number of frames to encode
UpdatedIntraPeriod = 25          ## The new intra period

################################################
##########END OF CONFIG.CFG####################
```

```
##############################################
Changing last parameter (from 1 to higher number) will change the
iterations of the testcase.
mm-venc-omx-test ENCODE config.cfg 100
The above command will run the same test case 100 times back to back.
(generates 100 separate output files)

As mentioned in config.cfg there is a parameter for "DynamicFile". This is
the location of another config file containing the dynamic setting intended
to test on encoder.
Dynamic configuration command syntax:
frame_num bitrate bps
frame_num ivoprefresh
E.g.
40 bitrate 128000
20 ivoprefresh
80 ivoprefresh
140 bitrate 1400000
```

The test app is only for basic verification; it is not intended for production or end-to-end use cases. Not all features or combinations are supported

B References

B.1 Related documents

Title	Number
Qualcomm Technologies, Inc.	
Application Note: Achieving Smooth Playback for High Frame Rate Video Scenarios on Android Framework	80-NL385-1
Wi-Fi Display Setup Guide	80-NP152-1
Application Note: Known CTS Issues	80-NR633-1
Seamless Resolution on Android Decoder	80-NK913-1
Application Note: Wi-Fi Display Capability Description	80-NP350-1
Application Note: Enable Secure Boot on APQ8084, MSM8974, MSM8x26, MSM8x10, and MSM8x12 Chipsets	80-NA157-20
Application Note: OpenMAX IL Video Decoder for ISDB-T Full-Seg on MSM8974 and APQ8064	80-NE886-1
OPENMAX Integration Layer Video Encoder For Linux Android	80-N1933-3
OPENMAX Integration Layer Video Decoder For Linux Android	80-VT322-2
Content Protection For Media Playback	80-NA157-206
MSM8992.LA Linux Video Overview	80-NN899-2
MSM8x36 Linux Android Video Overview	80-NM846-1
Presentation: MSM8x09 Linux Android Video Overview	80-NR964-18
Presentation: MSM8974 Linux Android Video Overview	80-NA157-11
Presentation: APQ8084 Linux Android Video Overview	80-NH717-7
Presentation: MSM8x26 Video Overview	80-ND928-75
Presentation: MSM8x12/Msm8x10 Linux Android Video Overview	80-NC839-9
Presentation: MSM8916 Linux Android Video Overview	80-NL239-14
Presentation: MSM8994.LA Linux Video Overview	80-NM328-20
Presentation: Qualcomm MPEG-Dash Solution Overview	80-NM976-1
Wi-Fi Display Debug Guide	80-NP566-1
Multimedia Driver Development and Bringup Guide - Video	80-NU323-5
Resources	
Compatibility Test Suite (CTS) User Manual	http://static.googleusercontent.com/media/source.android.com/en//compatibility/android-cts-manual.pdf

B.2 Acronyms and terms

Acronym or term	Definition
CBR	Constant bitrate
DUT	Device under test
HDCP	High-bandwidth digital content protection
ISDB-T	Integrated Services Digital Broadcasting-Terrestrial
OCMEM	On-chip memory
OMX	OpenMAX
SSR	Subsystem restart
TZ	TrustZone
VBR	Variable bitrate rate