Nexus
Transcode

Broadcom
Corporate Headquarters: San Jose, CA

Web: www.broadcom.com

Revision History

Revision	Date	Change Description
0.1	11/8/11	Initial draft
0.5	12/21/11	Updated
1.0	3/7/12	Add fast channel change for video encoder; Update TS user data, AFD/Bar data and speech codec support;
1.1	4/19/13	updated multiple places

Table of Contents

Introduction

This document describes techniques needed to implement a ViCE2-based transcode system using Nexus. It is assumed that the reader has general knowledge of nexus, and run and studied the nexus/examples/encoder/* and rockford/unittests/nexus/encoder/* applications, and therefore can build and run a simple nexus transcode application.

The intended audience is the nexus user. Nexus has the following transcode interfaces:

Capability	Explanation
NEXUS_VideoEncoderHandle	Handle to a video encoder (ViCE2 HW)
NEXUS_AudioMuxOutputHandle	Handle to an audio mux output (Audio DSP)
NEXUS_AudioEncoderHandle	Handle to an audio encoder (Audio DSP). It must connect to audio mux output interface to output encoded audio buffer.
NEXUS_DisplayHandle	Handle to an encode display which can have a simple timing generator (STG HW). The interface between encode display and ViCE2 HW is called “virtual displayable point”
NEXUS_StreamMuxHandle	Handle to a stream mux which can mux audio and video encoders output along with user provided system data to an MPEG2 TS stream (with XPT HW assistance).
NEXUS_FileMuxHandle	Handle to a file mux which can mux audio and video encoders output to a container file, like MP4 or ASF file
NEXUS_SyncChannelHandle	Handle to an AV sync module that can synchronize audio and video decoders up to the displayable point.
NEXUS_StcChannelHandle	Handle to a transport STC channel that tracks a timebase and can be configured in MOD300 or binary mode.

In order to explain how certain nexus interfaces work, there may be some reference to lower-level Magnum API’s or actual hardware blocks. The following is a mapping:

Nexus Interface	Magnum Module	Hardware Block
VideoEncoder	VCE (Video Coding Engine)	ViCE2

Audio	APE (Audio Process Engine)	RAAGA
Display	VDC (Video Display Controller)	STG/BVN
StreamMux	Muxlib/XPT	XPT
FileMux	Muxlib	File I/O
StcChannel	XPT	XPT

This document is also related to the following transcoder source topics:

Topic	Details
NEXUS_VideoDecoder	Decode MPEG2/H.264/MPEG4-Part2/VC1/MVC/SVC/VP8/Sorenson/AVS video using video decoder HW. See Nexus_Usage.pdf for documentation.
NEXUS_AudioDecoder	Decode various audio formats using audio DSP. See Nexus_Usage.pdf for documentation.
NEXUS_HdmiInput	Some SoC has HDMI input with both uncompressed video and audio source for transcoder. See Nexus_Usage.pdf for documentation.
NEXUS_ImageInput	USB Camera YUV input may be fed to transcoder via the NEXUS image input interface. See Nexus_Usage.pdf

Overview

The ViCE2 based transcoder system is partitioned into source decoder/display side and encoder/mux side with the so-called **Virtual Displayable Point** (**VDP**) to mimic the regular decoder box's display output in terms of AV frame synchronization.

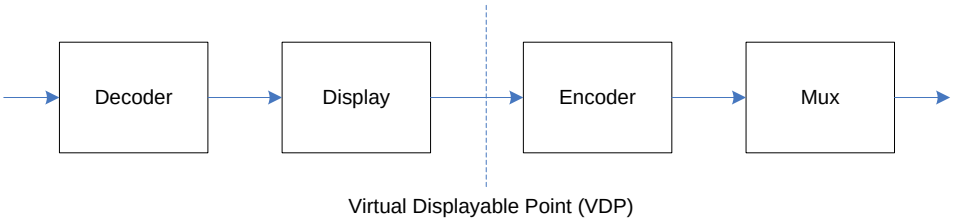

Figure 1. Transcoder system partition

This allows the transcoder system to reuse the most of the regular decoder system. This partition also simplifies the encoder design since the source discontinuity won't propagate to the encoder but is filtered by the decoder side TSM scheme as is in the regular decoder box. The encoder can assume its immediate input audio and video frames are synchronized and continuous as if are shown on a real display device.

Features

NEXUS StreamMux module currently supports MPEG2 TS output, FileMux module currently supports MP4 and ASF container output. The audio/video codecs and formats supported for each MUX container output may be different from chip to chip. Please talk to the relevant support team to get the complete feature list.

Note: specific 3rd party codecs or container support requires proper license in place to get the source code distribution.

TS StreamMux

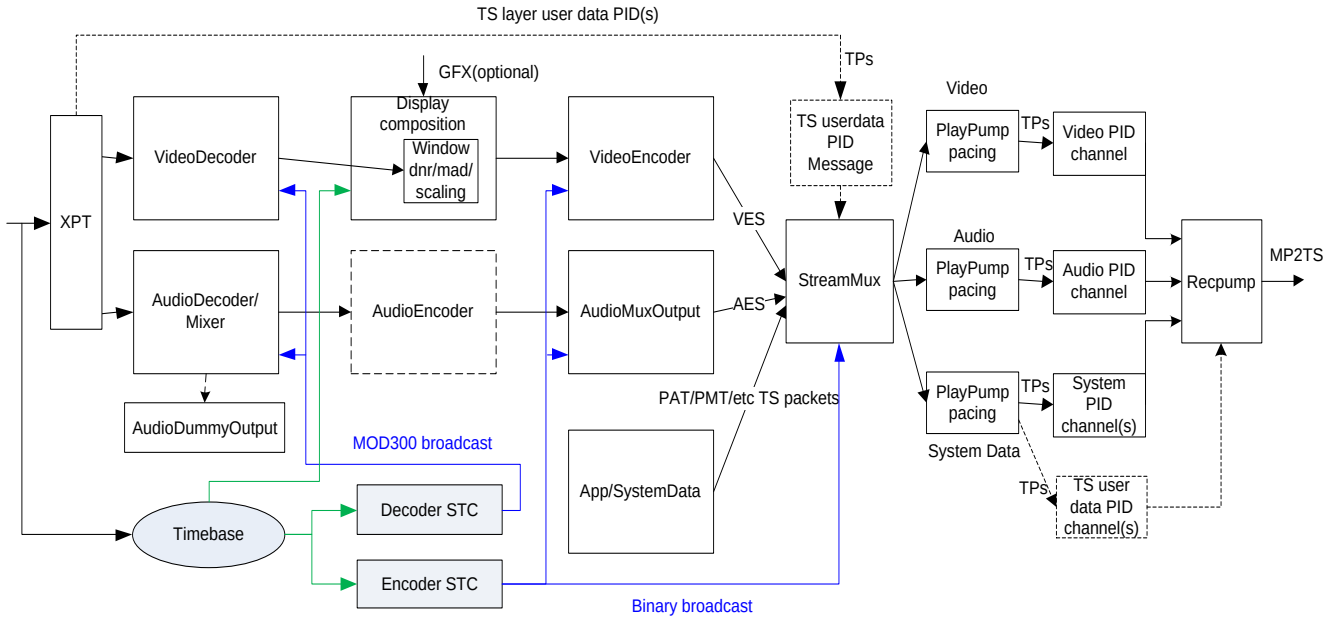

Figure 2. TS StreamMux system usage diagram

This diagram shows both video and audio transcoder. The output goes through TS stream MUX via playpump with PES pacing and TS packetization then multiplex'd by Recpump as MPEG2 TS stream. The input of audio and video decoders could be live tuner or XPT playback etc same as the regular decoder box input. The audio encoder could be removed and audio decoder would be in passthrough mode for the transcoder.

See nexus/examples/encoder/transcode_playback_to_ts.c, transcode_qam_to_ts.c and rockford/unittests/nexus/encoder/transcode_ts.c.

TS StreamMux with HDMI Input

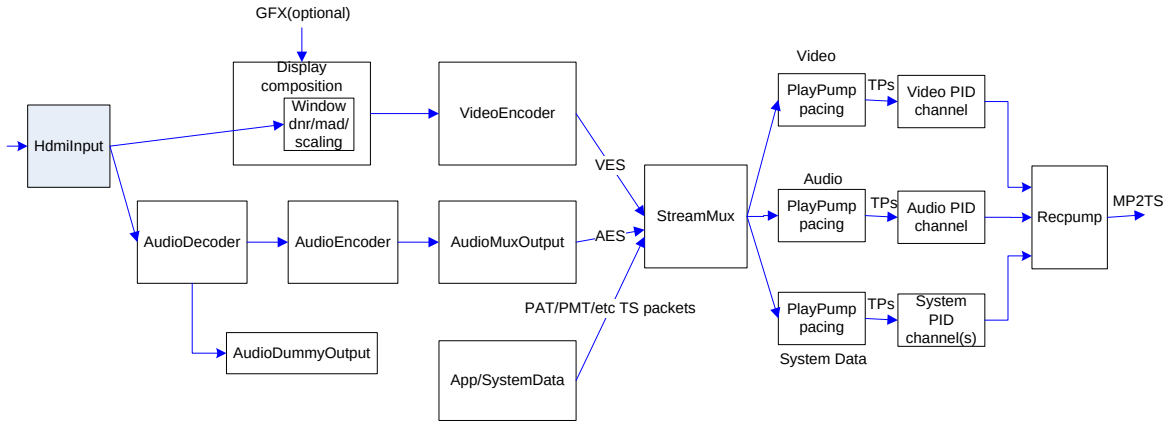

Figure 3. HDMI input Transcode

This diagram shows HDMI input being transcoded into TS output. The HDMI input video is uncompressed and directly fed to video window; while the audio could be compressed or uncompressed, and fed to the compressed or PCM audio decoder.

See nexus/examples/encoder/transcode_hdmi_to_ts.c and rockford/unittests/nexus/encoder/transcode_ts.c.

File Mux

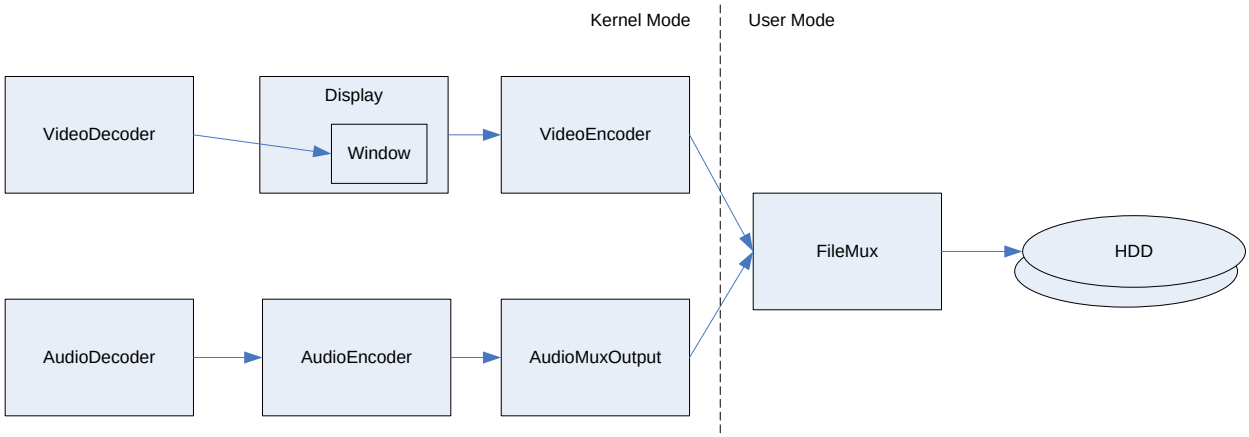

Figure 4. File Mux

This diagram shows the transcoder output is sent to a file MUX module to record as a media container file.

Note when NEXUS is built for kernel mode (export NEXUS_MODE=proxy), the NEXUS FileMux module is running in the user mode since it operates on the file I/O.

Note the ASF mux support is discontinued in the latest refsw releases.

See rockford/unittests/nexus/encoder/transcode_mp4.c.

ES output

Applications can directly get the elementary streams (ES) out of VideoEncoder and AudioMuxOutput without going through the stream mux or file mux module. One usage may be for video conferencing that application may packetize the H.264 video ES and G.7xx audio ES outputs into RTP stream to send to remote conferencing party.

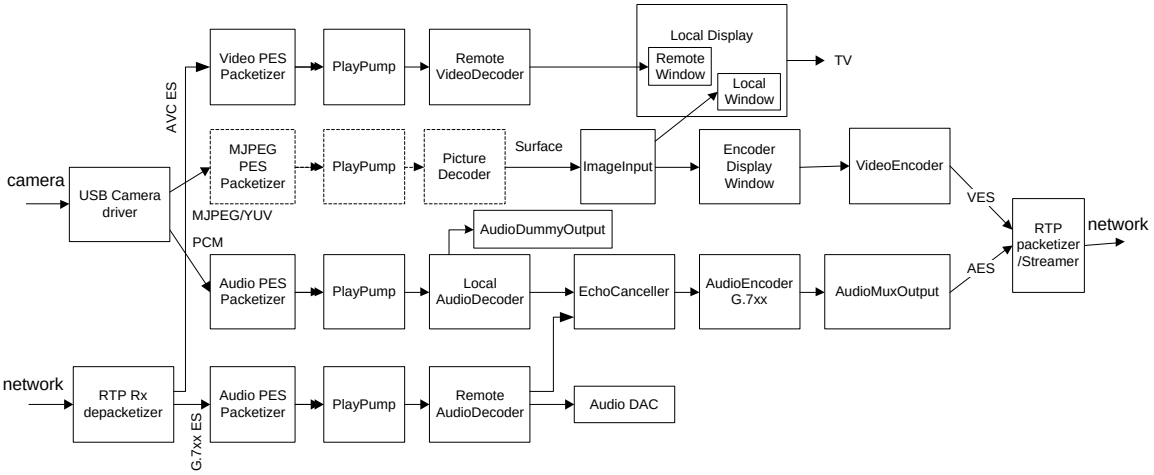

Figure 5. Video Conference Usage

Another application usage of ES output of encoder is to construct application specific container output, e.g., fragmented MP4 output or customer proprietary ASF output stream etc.

See nexus/examples/encoder/encode_video.c and encode_audio_es.c for how to get encoders output buffers.

App Notes

System

STC Channel

Each **Real-Time** mode transcoder's audio/video encoders and mux should share the same binary mode broadcast STC channel for AV sync and MUX pacing. The audio/video decoders also share a separate MOD300 mode STC channel for lipsync of the Virtual Displayable Point. To achieve end-to-end transcoder synchronization without slip, the decoders mod300 STC channel and encoders binary STC channel should share the same timebase.

However, in **Non-Real-Time** mode (file-to-file transcode), transcoder's audio and video decoders should use different STC channels and the two STC channels must be paired with some tolerant window to allow As Fast As Possible (AFAP) transcode and still keeps AV sync at the virtual displayable point. Note the NRT mode audio and video STC channels are separately triggered to increment by audio decoder and video display with frame-interval instead of 27MHz clock. The reason NRT mode audio and video decoders have different STC channels is the audio and video frame intervals are typically different.

AV Sync

The transcoder's AV sync is partitioned into decoder side and encoder side two parts.

The decoder side is responsible to keep AV sync of the Virtual Displayable Point and utilizes regular decoder AV sync algorithm found in the regular decoder boxes. In this regard, the NEXUS_SYNC_CHANNEL is reused to maintain the lip sync of virtual displayable point at the audio and video encoders input.

The encoder side just needs to balance the **A2P delay** or arrival to presentation delay (from virtual displayable point at input of encoders, to the downstream decoder's display) of the audio and video encoder buffer model. The encoder buffer model derives constant A2P delay given a set of encoding configuration to assure no overflow and underflow at the encoder buffer and downstream HRD decoder buffer.

Note given the XPT input audio/video frames carry the original PTS (coded or interpolated by decoder TSM), and the original PTS is carried forward to encoder output frames, it's possible to compute the transcoder AV sync error on-the-fly by correlate the audio and video transcoder output descriptors' original PTS and PTS:

AV sync error = (aPts – aOpts) - (vPts – vOpts);

If >0, video leads; Else, video lags; Where, aPts and vPts are encoder generated PTS per frame; aOpts and vOpts are original PTS per frame.

See the nexus/examples/encoder/transcode_ ts.c for the transcoder AV sync setup and measurement.

NRT Mode

Non-Real-Time (NRT) mode is a file-to-file transcode operation where each component executes as fast as possible (AFAP) and not tied to a fixed rate. It allows faster than real time operation – can go a lot faster if each component's throughput capacity is under-utilized in real-time mode, for example, SD decode -> SD transcode. Obviously, NRT mode could be slower than real time operation as well if the instructed transcode configuration exceeds the HW throughput limitation or allocated system bandwidth.

Under NRT mode, the key issues are:

1. How to control the transcoder data flow not to overrun to corrupt data and avoid under-run to utilize the HW throughput capability efficiently for the AFAP operation.

2. How to synchronize the audio and video transcoder pipelines so that one is not much ahead of each other that results in significant buffering for AV sync, and long transition of AV sync loss when source file hits discontinuity.

Broadcom HW/FW/SW architecture ensures the flow control of the NRT mode transcoder for the first item. Application just needs to instruct the relevant end-to-end NEXUS modules to operate in NRT mode via API.

To be noted, to properly operate in NRT mode, all the relevant NEXUS modules need to be set into NRT mode end-to-end, including decoders, display, encoder and MUX modules. Also the source playback module should give NULL STC channel as no trick mode needed at playback for NRT mode. The source file playback is inherently able to stall the file input when downstream decoders stall to avoid overflowing the decoder buffer. The recpump after the stream mux should enable band hold feature to be able to stall upstream MUX module to prevent record buffer being overflown. So the NRTmode file-to-file transcoder is under end-to-end flow control.

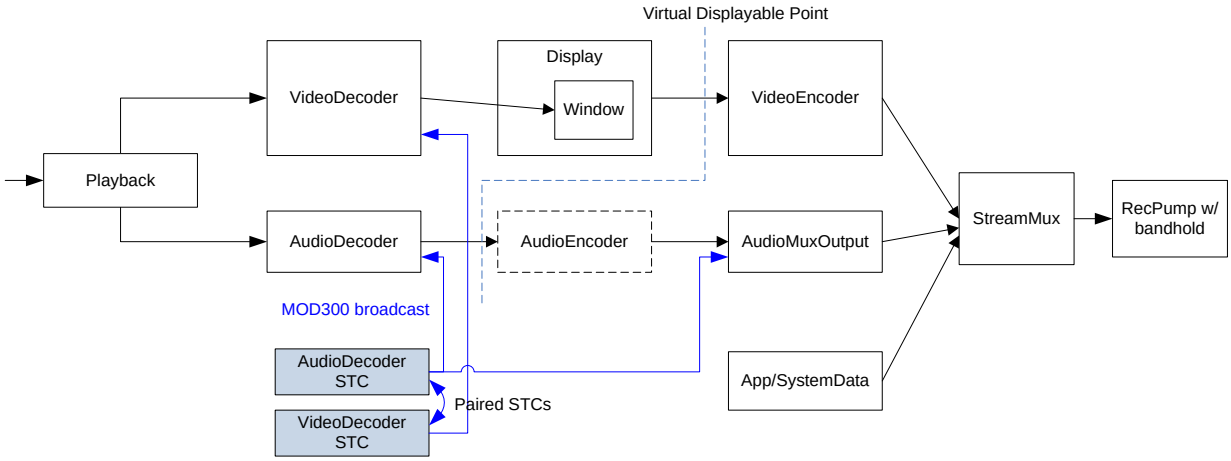

Figure 6. NRT mode STC usage

To properly synchronize the NRT mode audio and video decoders at the virtual displayable point, two **separate** STC channels need to be assigned to the audio and video decoders, and the two STC channels need to be **paired** with certain AV window via the NEXUS_SYNC_CHANNEL API to prevent audio and video races ahead of each other by too much (constrained by AV_WINDOW). The NRT mode audio and video decoder STCs are controlled by audio and video timestamp management (TSM) agents to advance frame by frame to be as fast as possible.

To be noted is the AudioMuxOutput should use the same STC channel as the audio decoder in NRT mode, which is different from the RT mode transcoder.

See examples in /nexus/examples/encoder/transcode_playback_to_ts_afap.c and /rockford/unittests/nexus/encoder/transcode_ts.c.

System Data

PCR packets are generated and inserted by the system data channel of the TS StreamMux internally complying with strict timing. User just needs to specify the NEXUS_StreamMuxPcrPid information in the NEXUS_StreamMuxStartSettings.

Application may add extra system data to the TS StreamMux via NEXUS_StreamMux_AddSystemDataBuffer API, for example to insert PAT/PMT packets.

Note for MPEG2 TS StreamMux, the non-PCR system data has to be complete TS packets. Application is responsible for the compliance of the TS packets. For example, the CRC at the end of PAT/PMT TS packets need to be correct, the continuity counter in the TS packet headers needs to be incremented properly for each system data PID etc.

See /nexus/examples/encoder/transcode_playback_to_ts.c and /rockford/unittests/nexus/encoder/transcode.c.

StreamMux Shutdown

Note to support clean shutdown of the stream mux, there is a "**finish**" API for StreamMux module. App can hook up a callback to wait for the clean finish of the stream mux before stopping it.

Also note the actual TS stream mux module owns and opens the audio and video **PID channels** when started, so the stream mux stop function would close the audio and video PID channels. That implies the downstream recpump or record must remove the audio and video PID channels from stream mux output before application stops the stream mux during shutdown.

See /nexus/examples/encoder/transcode_playback_to_ts.c etc.

FileMux Shutdown

Note to support clean shutdown of the file mux (especially MP4 file mux), there is a "**finish**" API for FileMux module. App should hook up a callback to wait for the clean finish of the file mux before stopping it. This is especially important for MP4 file mux since it needs to finalize the MP4 file (writing the "moov" box) to make the container valid and playable.

Note MP4 file format has option to enable progressive download support, which requires the moov box (file offsets etc metadata) in front of mdat box (actual ES data). However, the moov box cannot be generated until the whole MP4 mdat is available, i.e., at the end of transcode. that means the progressive downlable MP4 file mux finish would take long time to generate moov and copy mdat into output file especially for long file.

One way to speed up MP4 file mux finish, is to disable progressive download feature (MP4 finishes instantly since mdat box can be written to output file while transcoding operates without waiting to stop time and moov box can be written at end of file. The other way to speed up MP4 finish time is to increase relocationBuffer size in NEXUS_FileMuxCreateSettings.mp4 structure.

See /rockford/unittests/nexus/encoder/transcode_mp4.c.

Video

A video transcoder path consists of a video decoder, a video encoder display, a video window that connects with the video decoder input, and a video encoder. The video encoder display can optionally take in a graphics input in addition to the video input similar like regular display.

Static Settings

Video encoder static settings are in NEXUS_VideoEncoderStartSettings:

```
typedef struct NEXUS_VideoEncoderStartSettings
{
    NEXUS_DisplayHandle input;
    bool interlaced;
    bool nonRealTime;
    bool lowDelayPipeline;
    bool encodeUserData;
    NEXUS_VideoCodec codec;
    NEXUS_VideoCodecProfile profile;
    NEXUS_VideoCodecLevel level;
    NEXUS_VideoEncoderBounds bounds; /* Encoder Bounds - If the
bounds are known and specified, encoder latency can be improved.
These bounds are for a single encode session.  E.g. if the
output frame rate is known to to be fixed at 30fps, then the
output frame rate min/max can be set to 30. */
    unsigned rateBufferDelay;       /* in ms.  Higher values
indicate better quality but more latency.  0 indicates use
encoder's defaults */
    NEXUS_StcChannelHandle stcChannel;
} NEXUS_VideoEncoderStartSettings;
```

Dynamic Settings

These are the encoder settings changeable on-the-fly in NEXUS_VideoEncoderSetSettings after encoder is started:

```
typedef struct NEXUS_VideoEncoderSettings
{
    unsigned bitrateMax; /* units of bits per second */
    unsigned bitrateTarget; /* default 0: CBR and target=max;
non-zero: VBR target bitrate */
    bool variableFrameRate;/* default false since not every
decoder can handle variable framerate stream */
    bool enableFieldPairing;/* to enable PicAFF with repeat
cadence detection, to improve bit efficiency */

    NEXUS_VideoFrameRate    frameRate;
    NEXUS_VideoEncoderStreamStructure streamStructure; /* GOP
structure */
    unsigned encoderDelay; /* encoder delay, should be within
NEXUS_VideoEncoderDelayRange.min ..
NEXUS_VideoEncoderDelayRange.max range */
} NEXUS_VideoEncoderSettings;
```

Note that the transcoder does support dynamic resolution, but the encode resolution is set as a part of display format setting of the encoder display instead of video encoder setting.

Correct Encode Table

Correct Encode Table is concerned about display aspect of an encoder/transcoder system. Given input source format and encode format, it addresses the behaviors of Display Manager, display and video encode modules to help the encoded stream to achieve correct display on the target decoder system.

The general recommendation is to set the encoder display rate to 60/59.94 Hz for ATSC-based output formats and 50 Hz for PAL-based output formats.

If one requires frame rate pass-through transcoding, for example, input is 24/23.97p movie and encode frame rate is 24/23.97p as well, the encoder display rate can be set as 24/23.97p. This may help the NRT mode transcoder to run faster compared to always setting display rate of 60 Hz, for example.

Low Delay

Video conferencing requires low latency end-to-end from the camera input to the remote decoder display. It requires careful system level design to achieve. This section only touches video encoder configurations available for low delay usage.

The Broadcom video encoder supports the following fine control of low delay configuration.

```
To enable low delay config for video encoder, for example:

/* avoid B-frame in GOP structure */
videoEncoderConfig.streamStructure.framesB = 0;

/* disable Inverse Telecine Field Pairing to reduce delay */
pVideoEncoderConfig->enableFieldPairing = false;

/* 0 to means default 750ms rate buffer delay; change it to lower encode
delay in ms at cost of quality */
pVideoEncoderStartConfig->rateBufferDelay = 50; /* try 50 ms for example */

/* higher input minimum framerate for lower delay!
 * Note: lower minimum framerate means longer encode delay */
pVideoEncoderStartConfig->bounds.inputFrameRate.min =
NEXUS_VideoFrameRate_e59_94;

/* higher minimum output frame rate for lower delay! */
pVideoEncoderStartConfig->bounds.outputFrameRate.min =
NEXUS_VideoFrameRate_e29_97;
pVideoEncoderStartConfig->bounds.outputFrameRate.max =
NEXUS_VideoFrameRate_e30;

/* lower max resolution for lower encode delay */
pVideoEncoderStartConfig->bounds.inputDimension.max.width = 720;
pVideoEncoderStartConfig->bounds.inputDimension.max.height = 480;

/* enable video encoder pipeline low delay (MB row pacing);
   Note, lowDelayPipeline cannot be set true if
NEXUS_VideoEncoderOpenSettings.type != NEXUS_VideoEncoderType_eSingle; */
pVideoEncoderStartConfig->lowDelayPipeline = true;

To reduce delay for video encoder display:

/* full-size video may avoid 1-frame BVN capture delay */
NEXUS_VideoWindowSettings.forceCapture = false;
```

See rockford/unittests/nexus/encoder/transcode_ts.c for example of how to configure video encoder delay parameters.

Fast Channel Change

To support fast channel change for transcoder output live streaming, video encoder offers **eImmediate** stop mode that would flush/drop the internal buffered frames, instead of eNormal mode that has to wait for the encoder internal buffer being completely encoded:

```
/**

Summary:

Options for video encoder stop

**/

typedef enum NEXUS_VideoEncoderStopMode

{

    NEXUS_VideoEncoderStopMode_eImmediate,

    NEXUS_VideoEncoderStopMode_eAbort,

    …

} NEXUS_VideoEncoderStopMode;

typedef struct NEXUS_VideoEncoderStopSettings

{

    NEXUS_VideoEncoderStopMode mode;

} NEXUS_VideoEncoderStopSettings;
```

Video encoder also offers **adaptive low delay mode** for NEXUS_VideoEncoderStartSettings:

```
    bool adaptiveLowDelayMode;    /* If set  encoder will drop all incoming
frames until the first decoded frame is seen. The first frame will be encoded
with a low delay.  Then delay will automatically ramp to the value specified
in NEXUS_VideoEncoderSettings.encoderDelay, in the following frames. */
```

The adaptive low delay mode start would help downstream live decoder to show the video faster than normal mode. Once encoder delay is ramped to the specified delay (typically higher than low delay), the encoding quality can be retained as normal.

Camera Input

Camera input for video conferencing typically is via USB interface. The uncompressed YUV video or compressed MJPEG stream decoded via picture decoder becomes YUV video. Both need to go through the image input module to display then video encoder.

Other than the encoder low delay configurations, there are optimization items in the camera driver to reduce latency from camera input to NEXUS image input. The display path may also disable forced capture to reduce latency.

Audio

Audio Dummy Output

Note that in **real-time** transcode, audio decoder needs to be driven with an actual output or dummy output as shown above besides the transcoder path. 7425 supports 4 **audio dummy outputs** that can drive both compressed and decompressed audio decoder. Attaching dummy output here can save the actual audio output, like DAC or I2S etc for the local display usage instead of being tied to the transcoder.

In **non-real-time** transcode usage, there is no need to attach the actual audio output or dummy output to the audio decoder.

See /rockford/unittests/nexus/encoder/transcode_ts.c.

Audio DSP Mixer

To avoid source discontinuity propagates to the audio mux output that might confuse the stream mux operation and result in dropped audio, it's recommended to connect audio DSP mixer with the audio transcoder path's decoder output and before audio encoder (transcode mode) or audio mux output (passthrough mode).

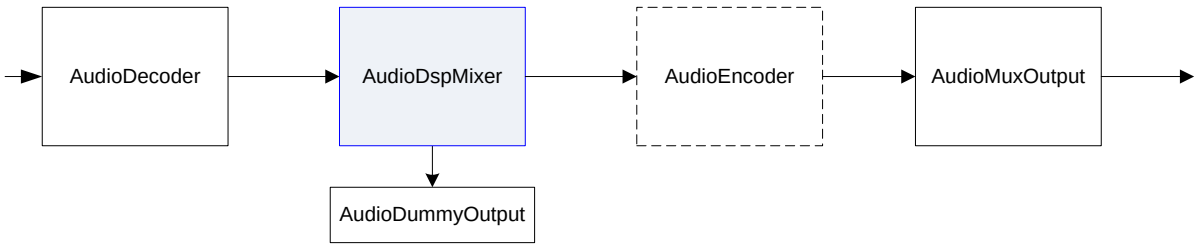

Figure 7. Audio DSP mixer usage

The audio DSP mixer output will be always continuous for the audio transcoder path even the source audio stream is discontinued or wrapped around. However, note this is DSP mixer that consumes audio DSP task resource, and may have audio memory footprint scaled to support multiple audio transcodes. TODO: optimize audio FW memory footprint.

See /rockford/unittests/nexus/encoder/transcode_ts.c.

Multiple Audios

We support a feature that a source stream has single video channel and multiple audio PIDs (up to 6x PIDs) to be compressed into a different stream with video being transcoded (in terms codec/bitrate/resolution etc), while keeping the multiple audios passed through or one of those audios being transcoded with the rest passed through. All the audio PIDs and video channel are based on the same timebase PCR. This works in both Real-Time mode and Non-Real-Time mode.

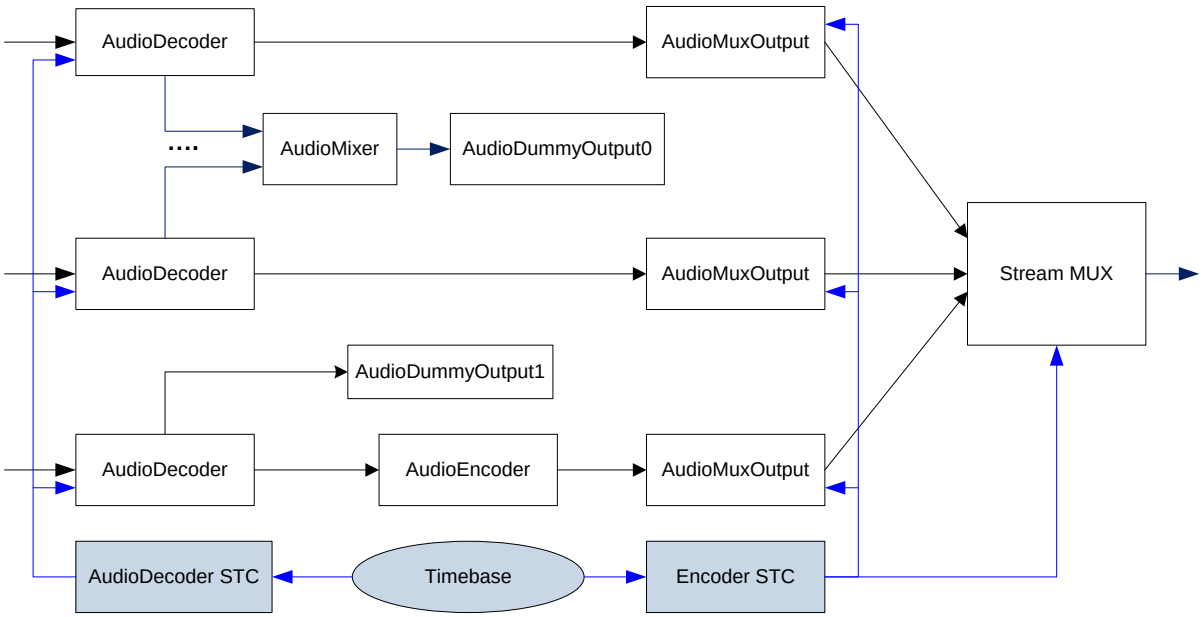

Figure 8. Multiple audio channels usage

The non-DSP audio mixer allows multiple audio decoders outputs (compressed or not) to be connected with a single dummy output. This is useful to have single mixer/dummy output to drive multiple audio pass-through decodes to save the audio usage of actual outputs or dummy outputs to drive multiple audios.

Just to note that the pass-through audio decoder cannot share the same mixer with the decompression audio decoder.

Also note that the AudioMuxOutput uses the encoder STC in RT mode but audio decoder STC in NRT mode. So above diagram shows the RT mode multiple audios transcode usage. In NRT mode, the same AudioDecoder STC should route to the multiple AudioMuxOutputs as well.

See /nexus/examples/encoder/transcode_playback_to_ts_6xaudio.c.

Audio Speech Codecs

Conferencing audio typically uses G.711 or G.729 etc low latency speech codecs. The USB camera input probably comes with uncompressed PCM audio.

Since G.711 codec is simple enough to perform in host SW, and custom speech echo cancellation algorithm could be implemented in application as well to avoid HW latency.

More complex G.729 etc codec is preferred to be done in audio DSP to offload CPU cycles. In that case, the echo cancellation algorithm is preferred to be performed in audio DSP as well instead of in host application to avoid unnecessary latency between host and DSP data flow.

See /rockford/unittests/nexus/audio/echo_canceller.c.

User Data

Closed Caption and ES layer User Data

The user data transcoding is currently supporting the following Closed Caption pass-through:

- CEA-608-E
- CEA-708-D
- A/53 D
- SCTE 21 2001
- SCTE 20 2001

The NEXUS API to turn on the Closed Caption user data transcoding is a Boolean in the NEXUS_VideoEncoderStartSettings structure:

```
bool encodeUserData;
```

See /rockford/unittests/nexus/encoder/transcode_ts.c

The AFD and bar user data in the source stream will be decoded to assist video processing pipeline to automatically crop the bar area so that AFD/Bar won't need to be encoded in h.264 output. When bar areas are cropped, the video aspect ratio would be automatically adjusted and coded properly for h.264 output.

See /rockford/unittests/nexus/encoder/transcode_ts.c as example.

Transport Layer User Data

The supported TS layer VBI user data specs are:

- ANSI/SCTE 127 2007
- ETSI EN 300 472 v1.3.1 (2003-05)
- ETSI EN 301 775 v1.1.1 (2000-11)

Subtitle, Teletext and AMOL etc VBI user data carried in MPEG2-TS layer can be passed through the TS stream mux via following API:

```
typedef struct NEXUS_StreamMuxUserDataPid {

    NEXUS_MessageHandle message; /* The userdata is expected to arrive as PES
packets encapsulated in TS packets. Application is responsible for calling
NEXUS_Message_Start prior to calling NEXUS_StreamMux_Start and call
NEXUS_Message_Stop after mux session was completed */

} NEXUS_StreamMuxUserDataPid;

typedef struct NEXUS_StreamMuxStartSettings

{

    ……

    NEXUS_StreamMuxUserDataPid userdata[NEXUS_MAX_MUX_PIDS];

}
```

Note, for TS layer VBI user data pass-through, in addition to the user data bitstreams, application probably also needs to pass through the PSI data in the PMT table that describes the VBI user data services.

See /rockford/unittests/nexus/encoder/transcode_ts.c as example.

Debug Tools

In addition to the regular BDBG debug modules message log, there are extra debug log tools in the relevant transcoder modules.

Video Encoder

Runtime API to get video encoder status is NEXUS_VideoEncoder_GetStatus, which reports the currently encoded pictures count, received pictures count and error flags etc.

User could also dump the video encoder output buffers to files if compiling VCE PI with the following flags in user mode:

export BVCE_DUMP_OUTPUT_CDB=y

export BVCE_DUMP_OUTPUT_ITB=y

The encoder FW log can be dumped to file with the following compile flag set:

export BVCE_DUMP_ARC_DEBUG_LOG=y

Audio Transcoder

The audio transcoder can be debugged with the same method like the regular decoder box.

The latest nexus audio module also added SW debug support of DSP log into file:

```
#include “nexus_audio_dsp.h”

typedef struct NEXUS_AudioModuleSettings

{

   /* …… */

   NEXUS_AudioDspDebugSettings dspDebugSettings;   /* DSP Debug settings */

} NEXUS_AudioModuleSettings;
```

To log the DSP debug into file, runtime environments on box are:

```
# export audio_debug_file=<file name>

# export audio_uart_file=<file name>

# export audio_core_file=<file name>
```

Mux

To log the mux’s input: audio and video encoders per-frame-based descriptors (including PTS/DTS/ESCR/frame_size/flags etc), the compile time flags are:

export BMUXLIB_INPUT_DUMP_DESC=y

export BMUXLIB_INPUT_DUMP_FRAME_DESC=y

The video and audio encoders' frame descriptors would be logged into separate BMUXLIB_INPUT_*.csv files for each input ES streams.

To log the TS mux's output transport descriptors that drive the XPT playback pacing of TS mux data, the compile time flag is:

export BMUXLIB_DUMP_TRANSPORT_DESC=y

The transport pacing descriptors would be logged into separate BMUXLIB_TRANSPORT_DESC_xx.csv files for each TS mux playpump channels.

This works with NEXUS user mode build.

Display

The video encoder display can be debugged via similar method like regular display module, e.g., to detect and log BVN errors in the transcoder display path from console, set following compile flag:

export BVDC_SUPPORT_BVN_DEBUG=y

To log the display RULs to a binary file for offline debug of display errors, set following compile flag for user mode NEXUS build:

export BRDC_USE_CAPTURE_BUFFER=y

then at runtime box console,

export rul_captures=<filename>

when you suspect display is not delivering correct pictures information to video encoder, you can turn on debug message for BVDC_DISP_STG module at runtime.

Limitations

Video Throughput

7425/7435 video encoder HW core supports up to 1080p30 or 1080i60 throughput in single transcoder platform usage, even though the BVN transcoder path supports up to 1080p60 throughput, in which case, the video encoder drops every second frame to encode 1080p30. For dual transcoder platform usage, the combined throughput of the dual transcoder channels is also restricted by this maximum throughput. That said, the dual transcoder platform supports

up to dual 720p30 transcoders or 1440x1080i60 + 480p30 dual transcoders. However, 7435 has two HW cores, so it supports up to quad 720p30 transcodes.

Since B-frame requires two reference pictures which consume a lot of bandwidth, 7425 only supports B-frame for up to half HD (720p30) encode.

7445 video encoder HW core throughput is doubled, so true B-frame support for HD (1080p30 and 1080i) will be available.

7445 also has two video encoder cores, each with doubled throughput, so 7445 supports quad 1080p30 transcodes.

Video Decoder Resource

Note that a video transcoder expects to set aside a video decoder and corresponding resource from the system. So for 7425 based system that supports 3 independent HD video decoders, the dual transcoders platform would only leave one video decoder for the local displays (i.e. no PIP decode).

So 7425 dual transcodes cannot co-exist with main+PIP local display system.

7445 has more orthorgonal resource available for system, so it's more flexible. Please see the relevant SoC system usage document for more details.

Number of Audio Channels

The audio DSP system does not have hard limitation of number of audio channels as far as the DSP MHz cap supports. However, typically number of audio transcode channels matches with video path. In other words, single transcoder platform supports up to one audio decoder and one audio encoder. Dual transcoders platform supports up to two audio decoders and two audio encoders.

If the rest of system is idle, 7425 supports up to 6x audio transcoder passthrough or 5x audio transcoder passthrough channels + one audio transcode channel simultaneously when those are based on the same PCR.

Please see the relevant SoC system usage document for more details.