Nexus

Usage Guide

Broadcom
Corporate Headquarters: San Jose, CA

Web: www.broadcom.com

Revision History

Revision	Date	Change Description
STB_Nexus-SWUM200-R	5/23/2008	Initial version
STB_Nexus-SWUM201-R	9/18/2008	Various updates
STB_Nexus-SWUM202-R	5/20/2009	All new version
STB_Nexus-SWUM203-R	10/19/09	Various updates
STB_Nexus-SWUM203-R	03/24/04	Various updates
2.0	10/26/2011	Document conversion, updated PM section
2.1	1/25/12	Document NEXUS_HAS/NUM macros
2.2	6/14/12	Add AudioDecoderPrimer
2.3	2/26/14	Rewrite VBI and Userdata
2.4	2/3/15	Remove reference to old message filter code
2.5	6/24/15	Update Common Macros section
2.6	9/14/15	Update Channel Change section
2.7	10/12/15	App note for handling 32 bit timestamp overflow on recordings
2.8	8/23/16	Add Dynamic Range and Mastering
2.9	8/24/16	Add Low Latency Decode
2.10	9/22/16	Update Linux Kernel Mode Without Proxy

Table of Contents

Introduction

Overview

Nexus is a high-level, modular Application Programming Interface (API) for Broadcom cable, satellite and Internet set-top boxes. This document describes how to use Nexus in your application. The emphasis is on explaining Interface interconnections and how an entire system is constructed. For more detailed, function-by-function API documentation, please refer to the Interface header file comments.

This document does not provide information on the Nexus architecture or describe how to write Nexus Modules. See Related Documents below.

Related Documents

- nexus/docs/Nexus_Architecture.pdf
- nexus/docs/Nexus_Development.pdf
- nexus/docs/Nexus_Graphics.pdf
- nexus/docs/Nexus_Memory.pdf

Terminology and Acronyms

Term/Acronym	Description
ASF	Advanced Systems Format
ASTM	Adaptive System Time Management
AVC	Advanced Video Codec (also referred to as H.264)
AVD	Advanced Video Decoder hardware block, used by VideoDecoder and StillDecoder
AVI	Audio Video Interleave file format
BBS	Broadband Studio, a Windows-based tool for Broadcom broadband silicon
BVN	Broadcom Video Network hardware block, used by Display
CDB	Compressed Data Buffer (buffer used by RAVE for stream data)
CGMS	Copy Generation Management System
DAC	Digital-to-Analog Convertor
DBG	Debug Interface, a Magnum base module
DSP	Digital Signal Processor
DVR	Digital Video Record (also referred to as PVR)
EOTF	Electro-Optical Transfer Function
FPGA	Field Programmable Gate Array
GFD	Graphics Feeder
HDR	High Dynamic Range
Interface	A Nexus Interface
ITB	Index Table Buffer (buffer used by RAVE to index the CDB)
KNI	Kernel Interface, a Magnum basemodule
MKV	Matroska file format
Module	A Nexus Module
OS	Operating System
OSD	On-screen Display
PCR	Program Clock Reference (part of MPEG-2 Systems)
PES	Packetized Elementary Stream (part of MPEG-2 Systems)
PID	Packet ID
PIG	Picture-In-Graphics
PIP	Picture-In-Picture
Platform	A Nexus Platform
PM	Power Management
PPS	Picture Parameter Set (part of AVC syntax)
PSI	Program Specific Information (part of MPEG-2 Systems)

PTS	Presentation Time Stamp (part of MPEG-2 Systems)
PVR	Personal Video Record (also referred to as DVR)
RAP	Random Access Point
RAVE	Record Audio Video Engine (part of XPT hardware block)
TSM	Time Stamp Management
SID	Still Image Decoder hardware block (Nexus interface is PictureDecoder)
SDR	Standard Dynamic Range
SPS	Sequence Parameter Set (part of AVC syntax)
STC	System Time Clock
TS	MPEG-2 Transport Stream
VBI	Vertical Blanking Interval
VDC	Video Display Control (Magnum porting interface for control of video display), controls the BVN
VPS	Video Program System
XPT	Transport hardware block, also Magnum software module for XPT hardware
XVD	Extended Video Decoder, Magnum software module for AVD
WSS	Wide Screen Signaling

Getting Started

Example Applications

The Nexus software bundle comes with a set of example applications which can quickly show how to use Nexus. You can build and run them on reference board, assuming all the Interfaces are supported. Example applications are located in the nexus/examples directory.

If you are running Nexus on your own board (not a reference board), if a required Module does not exist you may get a link error on some examples. If a feature is not supported, you may get a runtime error.

The example applications are intentionally brief. They contain little or no error checking in order to keep the code uncluttered. They contain no runtime options. You may need to modify the code and change audio/video PID's or codec types in order to run them on your system.

Building and Running Example Applications

First, set up a Linux® build server[1] with the appropriate toolchain and Linux kernel source for your set-top platform. Refer to the installation guide that comes with your software release bundle. Nexus has been built on Linux systems as old as Red Hat Linux v7.3 (with GNU Make 3.79.1 (06/23/2000) and Perl 5.6.1 (04/08/2001)). All newer Linux systems should be supported as well.

Next, define the NEXUS_PLATFORM and BCHP_VER environment variables. The NEXUS_PLATFORM variable specifies the Broadcom reference board that the software release is intended for. It corresponds to the nexus/platforms/$(NEXUS_PLATFORM) directory. The BCHP_VER variable specifies the revision of the System-on-a-Chip (SoC) on the reference board.

For instance:

```
export NEXUS_PLATFORM=97445
export BCHP_VER=D0
```

Now you are ready to build:

```
cd nexus/examples
make
```

[1]Nexus build is not supported on UNIX or Solaris. If you provide your own toolchain and configured Linux kernel source, then it may be possible to build.

The resulting Nexus shared library, driver, and example application binaries will be copied into obj.$(NEXUS_PLATFORM)/nexus/bin[2]. Copy those binaries and the accompanying scripts to your set-top platform. Run the example applications using the “nexus” script as a prefix. This script will install the driver, create device nodes, set your shared library path, and then invoke your application.

Run as follows:

```
nexus graphics

nexus decode
```

Creating Your Own Application

To write your own application, pick an example application as a template.

Study nexus/examples/Makefile for directions on how to use the nexus/platforms/$(NEXUS_PLATFORM)/build/platform_app.inc Makefile include.

Creating Your Own Platform

When you first receive Nexus, you should run one of the supplied Platforms on a Broadcom reference board. You should be able to run some of the example applications without making modifications.

You may not want to create your own Platform for your application. Writing a Platform requires greater knowledge of Nexus internal structures than an application author typically needs to know, Platform development is therefore documented in the *Nexus_Development.pdf*, not this document.

Using KNI, DBG, and OSlib in Applications

If you look at applications in the nexus/examples directory, you’ll see that many of them use Magnum’s KNI and DBG base modules. This is allowed because these base modules are reentrant and do not have chip dependencies.

Nexus Base is not directly callable by an application. It is designed only to be called by Nexus modules.

[2] Referred to elsewhere as obj.97xxx/nexus/bin. You can override this location using NEXUS_BIN_DIR

You can find a general purpose higher-level OS abstraction in nexus/libs/os. This provides many similar features as Nexus Base, but is callable by applications. This is part of Broadcom’s AppLib architecture.

Module and Interface Overview

The Nexus directory tree is organized by module. Each module has one or more interfaces for the features it supports.

Lists of Modules and Interfaces by Functional Group

Functional Group	Module	Interfaces	Overview
Demux	Transport	InputBand	External transport inputs from demods or streamers
		ParserBand	Route an InputBand to multiple PidChannels
		PidChannel	Route one PID from a ParserBand to a consumer (decoder, message filtering, record, remux)
		Message	Transport message filtering from a PidChannel
		Remux	Remultiplex output
Video	Display	Display	Compositing and display of video and graphics
		VideoWindow	Position of windows on a Display
		ComponentOutput	Video output from Display
		CompositeOutput	Video output from Display
		SvideoOutput	Video output from Display
		ScartOutput	Set SCART output control pins
	VideoDecoder	VideoDecoder	Decode compressed digital video for input to VideoWindow
		VideoDecoder Primer	Primer for fast channel change
		StillDecoder	Decode MPEG/AVC/VC1 stills to a graphics surface
	Video Input	VideoImageInput	Route graphics to a VideoWindow (uses video feeder, not graphics feeder)
Audio	Audio	AudioDecoder	Audio input from compressed digital audio
		AudioDecoder Primer	Primer for fast audio track switching
		AudioPlayback	Audio input from PCM audio
		AudioMixer	Mix multiple audio inputs for routing into multiple audio outputs
		AudioDac	Audio output
		SpdifOutput	Audio output
Lipsync	Transport	StcChannel	Synchronization connection between audio and video decoders

		Timebase	Clock recovery, synchronization
	SyncChannel	SyncChannel	High-precision lipsync (extends StcChannel with audio/video fine tuning)
	Astm	Astm	Adaptive System Time Management
Tune and Demod	Frontend	Frontend	Digital demodulator
		Tuner	RF tuners
		Amplifier	Low Noise Amplifier (LNA)
Graphics	Surface	Surface	Block of memory with graphics properties.
	Graphics2D	Graphics2D	Fill and blit using graphics compositor (M2MC)
	GraphicsV3D	Graphicsv3d	Internal API for 3D graphics. See OpenGL ES-2.0 library for public API.
	Picture Decoder	Picture Decoder	Decode JPEGs, PNGs and other image formats to a graphics surface
IO Control	Pwm	Pwm	Pulse-width modulation interface
	Gpio	Gpio	General purpose IO pin control
	I2c	I2c	I2C bus
	Uart	Uart	Universal asynchronous receiver/transmitter
	Dma	Dma	Host-controlled memory-to-memory DMA
	Spi	Spi	Serial Peripheral Interface bus
	IrBlaster	IrBlaster	Infrared Blaster output
	IrInput	IrInput	Infrared Remote input
	Keypad	Keypad	Front panel keypad input
	Led	Led	Front panel LED control
	UhfInput	UhfInput	UHF Remote input
DVR	Transport	Playpump	Low-level playback interface used by Playback
		Recpump	Low-level record interface used by Record
	Playback	Playback	High-level playback including trick modes and file I/O
	Record	Record	High-level record including indexing
	File	File	File I/O used by Playback and Record
HDMI	HdmiInput	HdmiInput	HDMI receiver. Has video and audio input connectors.
	HdmiOutput	HdmiOutput	HDMI transmitter. Has video and audio output connectors.

Finding Interface Header Files

The API documentation for Nexus interfaces is contained in inline comments in the header files. Use a code browser that has a cross-reference feature in order to quickly traverse the API.

All Interface header files are found in the following directories:

```
nexus/modules/MODULENAME/include/*.h
```

Be aware that the mapping of Interface to Module is not fixed. Modules may be refactored, causing Interfaces to be moved into other modules. This will not change its functionality or break your application, but it does require you to be aware of changes in the Nexus directory tree when browsing the code.

You can also request nexus to assemble all public API header files into a single location by doing "cd nexus/examples; make nexus_headers; ls ../obj.97xxx/nexus/bin/include".

Decode Example

Overview

A simple way to start learning Nexus is to study an example application. This section explains the nexus/examples/decode.c example that performs a live video and audio decode. Nexus applications are built by the following steps:

1. Initialize the Platform
2. Open Interfaces for the features you need
3. Connect those Interfaces into a filter graph that models the data flow in your system
4. Set settings per Interface
5. Call Start functions to begin processing data

A diagram of the decode.c example application is shown here:

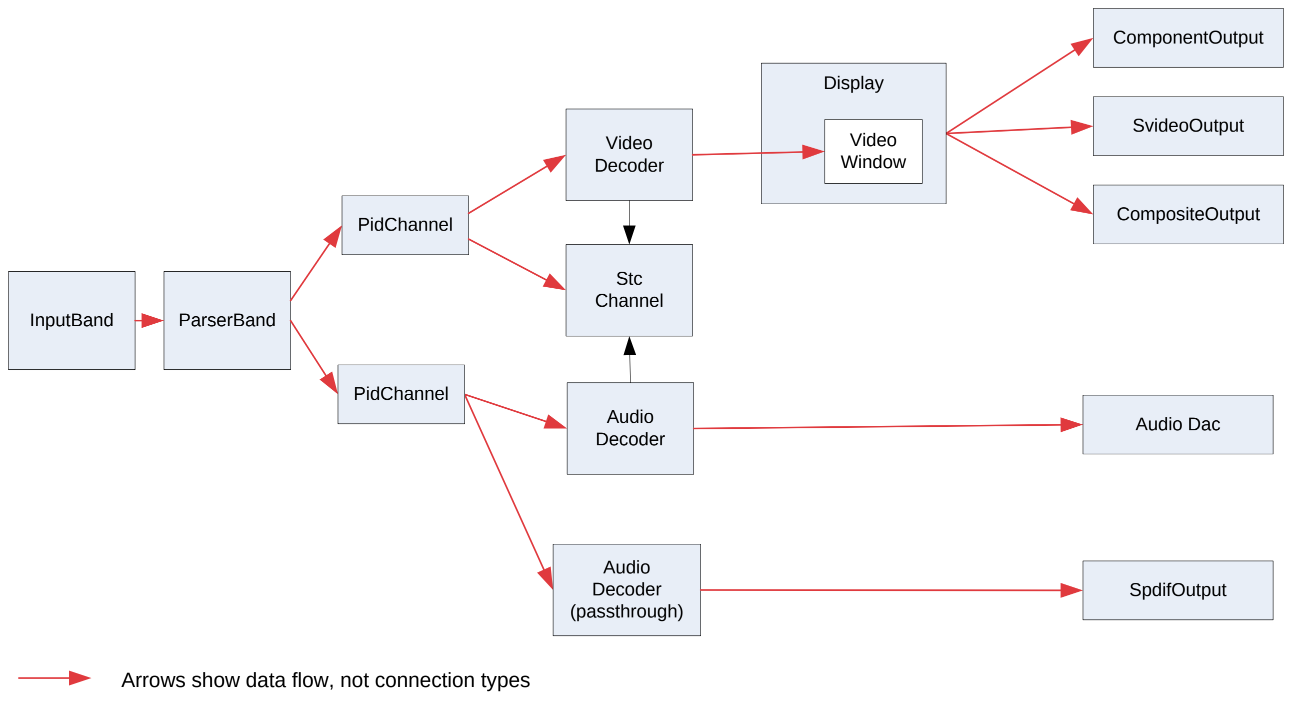


Figure 1: decode.c Example Application

Making Connections

There are different ways of connecting interfaces depending on their characteristics[3]. The connections in decode.c are made as shown in the following table:

Interface	Description
InputBand	decode.c using an external transport input from a streamer. The selection of the input band depends on the FPGA configuration on our reference board, which is platform-specific. Therefore, the application calls NEXUS_Platform_GetStreamerInputBand to obtain the NEXUS_InputBand.
ParserBand	The ParserBand is connected to an InputBand using NEXUS_ParserBand_SetSettings.
PidChannel	Video, audio and PCR PID channels are opened on a ParserBand using NEXUS_PidChannel_Open.
StcChannel	An StcChannel is connected to a PCR PidChannel using NEXUS_StcChannel_SetSettings.
VideoDecoder	VideoDecoder is given a video PidChannel and the shared StcChannel in NEXUS_VideoDecoder_Start.
AudioDecoder	AudioDecoder is given an audio PidChannel and the shared StcChannel in NEXUS_AudioDecoder_Start.
Display	The Display is an aggregator. A set of inputs and outputs is connected to it.
VideoWindow	VideoWindow is opened on a Display using NEXUS_VideoWindow_Open.
ComponentOutput, SvideoOutput, CompositeOutput	The configuration of video outputs is platform-specific; therefore they are opened by the Platform and obtained by the application using NEXUS_Platform_GetConfiguration. Video outputs are connected to the Display using NEXUS_Display_AddOutput.
AudioDac	The configuration of audio outputs is platform-specific; therefore they are opened by the Platform and obtained by the application using NEXUS_Platform_GetConfiguration. The AudioDac output is connected to the AudioDecoder using NEXUS_AudioOutput_AddInput.
SpdifOutput	The configuration of audio outputs is platform-specific; therefore they are opened by the Platform and obtained by the application using NEXUS_Platform_GetConfiguration. The SpdifOutput output is connected to the pass through AudioDecoder using NEXUS_AudioOutput_AddInput.

[3]Refer to the *Nexus_Development.pdf* for a discussion of Interface patterns and connection patterns. Nexus does not have a single connection pattern, but we did attempt to minimize the number of ways that interfaces connect.

Starting

Some interfaces have Start functions. Once Start is called, Nexus begins internal processing of data.

The following interfaces have Start functions:

- VideoDecoder - `NEXUS_VideoDecoder_Start`
- AudioDecoder - `NEXUS_AudioDecoder_Start`
- Playpump - `NEXUS_Playpump_Start`
- Playback - `NEXUS_Playback_Start`

Other interfaces have enable/disable booleans. In most cases, these interfaces default to being enabled:

- ParserBand
- PidChannel
- VideoWindow (visible boolean)

Other interfaces are turned on just by virtue of being connected. Audio and video outputs work like this.

General API Usage

Nexus Interfaces follow consistent patterns that make Nexus easy to learn and use. This section describes those basic patterns.

Starting the Application by Calling NEXUS_Platform_Init

Platform code is the place for customer-specific integration. Every Nexus bundle includes Nexus Platforms that support Broadcom reference platforms. These Broadcom-supplied Platforms have a few common features that make cross-platform applications possible.

`NEXUS_Platform_Init()` contains the code to initialize all the modules in the system. It connects the modules as needed and performs any board initialization. `NEXUS_PlatformSettings` contains settings that can customize this process:

```
NEXUS_PlatformSettings platformSettings
NEXUS_Platform_GetDefaultSettings(&platformSettings);
NEXUS_Platform_Init(&platformSettings);
```

`NEXUS_PlatformSettings` has options for `NEXUS_Platform_Init()` to automatically open and configure certain board-specific Interfaces. This includes setting the DAC assignments for various video outputs, detecting tuner daughter cards, and other conveniences. You can call `NEXUS_Platform_GetConfiguration` to retrieve these handles. Note that you cannot reopen an Interface that has already been opened. You must get the handle via the `NEXUS_Platform_GetConfiguration` command.

```
NEXUS_PlatformConfiguration config;
NEXUS_Platform_GetConfiguration(&config);
```

GetDefaultSettings, Open and Close

Interfaces that manage hardware resources have `Open` and `Close` functions. The `Open` function usually takes an index that enumerates the hardware resource as a Settings structure for initial configuration.

In order to make the Settings structure resilient to the future addition of new structure members, every Settings structure that is used in an `Open` call is required to have a `GetDefaultSettings` call.

If an Interface is able to be opened without any required user-supplied Settings, the user may pass NULL for the Settings structure. This is equivalent to calling `GetDefaultSettings` and leaving the structure unmodified.

```
NEXUS_SvideoInputSettings settings;
NEXUS_SvideoInput_GetDefaultSettings(&settings);
svideo = NEXUS_SvideoInput_Open(0, &settings);

...do work...

NEXUS_SvideoInput_Close(svideo);
```

Create and Destroy

For Interfaces that do not enumerate finite hardware resources, we use `Create` and `Destroy`. No index is needed.

```
playback = NEXUS_Playback_Create(&settings);

...do work...

NEXUS_Playback_Destroy(playback);
```

GetSettings and SetSettings

After opening or creating the Interface, you can configure it using various Settings structures. For simple interfaces, there is one main `GetSettings` and `SetSettings` pair.

`GetSettings` is guaranteed to return the same contents as the last successful `SetSettings` call. Therefore, applications can make incremental changes to structures.

```
/* change position */
NEXUS_VideoWindow_GetSettings(window, &settings);
window.position.x = 100;
window.position.y = 50;
NEXUS_VideoWindow_SetSettings(window, &settings);

/* change size */
NEXUS_VideoWindow_GetSettings(window, &settings);
window.position.width = 300;
window.position.height = 300;
NEXUS_VideoWindow_SetSettings(window, &settings);
```

GetBuffer/ReadComplete

Data stream capture is performed using a consistent method. A `GetBuffer` function will return the pointer and size for the next available data. `GetBuffer` is nondestructive. It can be called any number of times and the buffer pointer will not advance. The size may increase if more data becomes available.

The buffer pointer returned is to cached memory and has already been properly flushed. The application can access the read the buffer directly. No copy is required.

After the user processes the data returned by `GetBuffer`, it calls `ReadComplete` to inform Nexus of how much data has been consumed. The buffer point will advance by that amount.

Here is an example using the Message interface:

```
data_ready() {
    BKNI_SetEvent(event);
}

NEXUS_Message_Start(message);
while (!done) {
    NEXUS_Message_GetBuffer(message, &buffer, &size);
    if (!size) {
         BKNI_WaitForEvent(event, 1000);
         continue;
    }

     /* process data in buffer */
     size_used = process(buffer, size);
     NEXUS_Message_ReadComplete(message, size_used);
}
```

All `GetBuffer`/`ReadComplete` interfaces will have a data ready `NEXUS_CallbackDesc`. If the callback function is set, the user will get a callback when data becomes available. After receiving a callback, no more callbacks will be fired until `GetBuffer` is called.

The user must call `GetBuffer` after every `ReadComplete`. The user cannot call `ReadComplete` more than once after a `GetBuffer` call. Nexus may have an internal state that requires this. Most `GetBuffer`/`ReadComplete` interfaces will enforce this requirement.

The name “ReadComplete” was chosen with reference to the user. The user will read from the buffer and do work. When they are done reading from the buffer, they call `ReadComplete`.

GetConnector

Nexus has four abstract connection Interfaces: `NEXUS_VideoInput`, `NEXUS_VideoOutput`, `NEXUS_AudioInput`, and `NEXUS_AudioOutput`. These connectors allow a wide range of different interfaces to interconnect without introducing direct dependencies in their public APIs. Internally, direct connections are made as needed.

Abstract connectors are obtained using a `GetConnector` API. For interfaces that provide both audio and video connections (for example, RFM, HDMI), a `GetAudioConnector` and `GetVideoConnector` function are used. For interfaces that provide both input and output connectors (for example, SCART), functions like `GetVideoOutputConnector` are used to differentiate inputs from inputs.

The `GetConnector` function returns that same connector handle regardless of how many times it is called. There is no `Open`/`Close` for the connector.

Not all connections are valid. For instance, you cannot connect a ComponentInput directly to a VideoWindow: it must be routed through an AnalogVideoDecoder first. Nexus enforces that valid connections are made. Valid connections are also documented in this document as well as in the API documentation (see Related Documents).

Some connections can be made from one producer to multiple consumers. If so, this will be documented in the API documentation and Nexus will allow it (for example, dual VEC output of one video decoder). If not supported, Nexus will return an error on a second connection.

Shutdown

When an abstract connector is created in the source, it creates software state. This state is typically destroyed when the source is destroyed. However, when a connection is made using that connector, there is software state created in the destination. The destination software state is only freed on “Shutdown”. It is typically not freed when disconnected.

Shutdown can be performed manually or automatically[4].

Manual shutdown is done with one of four functions: NEXUS_VideoInput_Shutdown, NEXUS_VideoOutput_Shutdown, NEXUS_AudioInput_Shutdown, and NEXUS_AudioOutput_Shutdown. When the function is called, the connection is broken and the destination state is destroyed. It is as-if the connection never happened. Calling shutdown on a disconnected interface has no effect.

Here is an example of a manual shutdown of `NEXUS_VideoInput`:

[4] Automatic shutdown was added in July 2011.

```
vdecode = NEXUS_VideoDecoder_Open(0, &settings);
window = NEXUS_VideoWindow_Open(display, 0);
videoInput = NEXUS_VideoDecoder_GetConnector(vdecode);
NEXUS_VideoWindow_AddInput(window, videoInput);

# perform decode

NEXUS_VideoWindow_RemoveInput(window, videoInput);
NEXUS_VideoInput_Shutdown(videoInput);
NEXUS_VideoWindow_Close(window);
NEXUS_VideoDecoder_Close(vdecode);
```

The `Shutdown` function is not required if you are reconnecting to another consumer. It is only required before closing the producer. The `Shutdown` is required because the consumer may cache connection data about a producer. This allows the application to reconnect without having to reapply data.

Automatic shutdown is done when the source is closed. The thunk layer creates a shutdown call based on the /* attr{shutdown=xxx} */ attribute. The automatic shutdown function must be manually coded in the platform layer.

Here is an example of an automatic shutdown of `NEXUS_VideoInput`:

```
vdecode = NEXUS_VideoDecoder_Open(0, &settings);
window = NEXUS_VideoWindow_Open(display, 0);
videoInput = NEXUS_VideoDecoder_GetConnector(vdecode);
NEXUS_VideoWindow_AddInput(window, videoInput);

# perform decode

NEXUS_VideoWindow_Close(window);
NEXUS_VideoDecoder_Close(vdecode);
```

After the source (for example, VideoDecoder) is closed, the NEXUS_VideoInput is no longer valid. It has already been shutdown, either manually or automatically. Calling shutdown on its VideoInput will result in an assertion because you are accessing freed memory.

Callbacks

Nexus has a simple, consistent callback mechanism for all Interfaces. A callback descriptor (`NEXUS_CallbackDesc`) bundles three elements together:

- User-supplied callback function pointer
- User-supplied `void *` context pointer
- User-supplied `int` context parameter

Here is an example of the `sourceChanged` callback that is called whenever the video source size or other meta-data changes:

```
NEXUS_VideoDecoder_GetSettings(vdecode, &settings);
settings.sourceChanged.callback = my_source_changed_callback;
settings.sourceChanged.context = (void *)vdecode;
settings.sourceChanged.param = 0;
NEXUS_VideoDecoder_SetSettings(vdecode, &settings);

void my_source_changed_callback(void *context, int unused_param)
{
    NEXUS_VideoDecoder vdecode = (NEXUS_VideoDecoder)context;
    NEXUS_VideoDecoderStatus status;

    NEXUS_VideoDecoder_GetStatus(vdecode, &status);
    do_work(status.source.width, status.source.height);
}
```

When your application receives a callback, it can call back into Nexus immediately with no danger of re-entrance or deadlock. The application should still attempt to be as fast as possible in the callback because other callbacks may be queued up waiting for your callback function to return.

The Nexus callback does not pass any internal data in the callback. The only data provided are the `void *` context pointer and the `int` parameter passed in by the user. You can use that data to reference your own data structures in order to retrieve Nexus handles. We do not provide internal data for the following reasons:

- Synchronization of “current state” data would require additional buffering and the information will go out of date. If your application needs a state change notification, requesting a state change callback then querying the current state in the callback will always result in up-to-date information.
- The Nexus callback mechanism minimizes contexts switches and uses prioritized callback queuing. If the callback was generated in response to an interrupt, Nexus will queue up the callback from inside the interrupt handler based on module priority. Nexus will callback your application from a thread. In that callback, you can do work immediately. This means that there will only be one context switch before you are able to do work, which is optimal.

- Nexus is designed to operate across execution contexts, including spanning Linux kernel/user mode and working between processes. A simple callback mechanism makes that possible.

Refer to the *Nexus_Architecture.pdf* for more details.

Common Macros

NEXUS_HAS_<MODULE> macros

When a module's .inc file is included into platform_modules.inc, it will be built into nexus. The plaform's main Makefile will also include nexus/build/nexus.inc, which will process each module's defines. nexus.inc will also automatically #define a NEXUS_HAS_<MODULE> macro and add it to the NEXUS_CFLAGS given to the app.

A typical question that is frequently asked is, "Where is NEXUS_HAS_DISPLAY defined?" The answer is: it is dynamically defined by nexus.inc; therefore your search cannot find it. Also, you cannot remove it directly. You can only remove it **indirectly** by removing display.inc from your platform_modules.inc.

The main use of NEXUS_HAS_<MODULE> macros is to know whether an API is callable. For instance, if NEXUS_HAS_HDMI_OUTPUT is not defined, then you will get a linking error if you call NEXUS_HdmiOutput_GetDefaultSettings. Instead, your code should be written like this:

```
#if NEXUS_HAS_HDMI_OUTPUT
#include "nexus_hdmi_output.h"
NEXUS_HdmiOutputSettings settings;
NEXUS_HdmiOutput_GetDefaultSettings(&settings);
Handle = NEXUS_HdmiOutput_Open(0, &settings);
#else
/* do not make any HDMI output include or call */
#endif
```

Because nexus module's are always singular, the NEXUS_HAS macros are always singular. You would use #if NEXUS_HAS_VIDEO_DECODER and never #if NEXUS_HAS_VIDEO_DECODERS.

Also, no one should explicitly define NEXUS_HAS_<anything> for something else in nexus. For instance, if your module has a certain feature, to avoid confusion you should not explicitly #define NEXUS_HAS_VIDEO_DECODER_FEATURE_X.

NEXUS_MAX_<INTERFACE> macros

NEXUS_MAX_<INTERFACE> is used for fixed-sized arrays in the API or implementation. The value is constant for all chips and platforms; therefore it could be much larger than what is actually usable on your chip or platform.

Use run-time per-module capabilities functions, like NEXUS_GetVideoDecoderCapabilties or NEXUS_GetDisplayCapabilities or NEXUS_GetTransportCapabilities to learn which resources are actually available on your chip or platform. The actual usable resources may be sparsely populated. For instances, NEXUS_MAX_DISPLAYS could be 8, the actual usable number could be 3 and they may be indices 0, 1 and 5.

The semantics of NEXUS_MAX_<INTERFACE> are more like TOTAL. If NEXUS_MAX is 7, that means resources 0 through 6 may be valid. 7 will not be valid.

NEXUS_NUM_<INTERFACE> macros

NEXUS_NUM_<INTERFACE> have been deprecated. They should no longer be used and may be removed in the future. They were unable to express the run-time variations we have on newer silicon.

Instead, use NEXUS_HAS_<MODULE> to know if an interface exists, use NEXUS_MAX_<INTERFACE> for Nexus API's, and use run-time capabilities functions for actual resources.

For backward compatibility, NEXUS_NUM_<INTERFACE> macros will be retained for a while, but may be different from the previous values as they are converted to generic values. The conversion will follow these guidelines:

- If NEXUS_NUM_<INTERFACE> was 0, it will remain 0. This means a "#if NEXUS_NUM_<INTERFACE>" test should remain valid.
- NEXUS_NUM_<INTERFACE> will be greater than or equal to the previous chip-specific value.

Transport

The Transport interfaces route data from a variety of inputs to a variety of outputs. Inputs include:

- `NEXUS_InputBand`
- `NEXUS_PlaypumpHandle`
- `NEXUS_PlaybackHandle`

Outputs include:

- `NEXUS_VideoDecoderHandle` and `NEXUS_AudioDecoderHandle`[5]
- `NEXUS_RecpumpHandle` and `NEXUS_RecordHandle`
- `NEXUS_MessageHandle`
- `NEXUS_RemuxHandle`

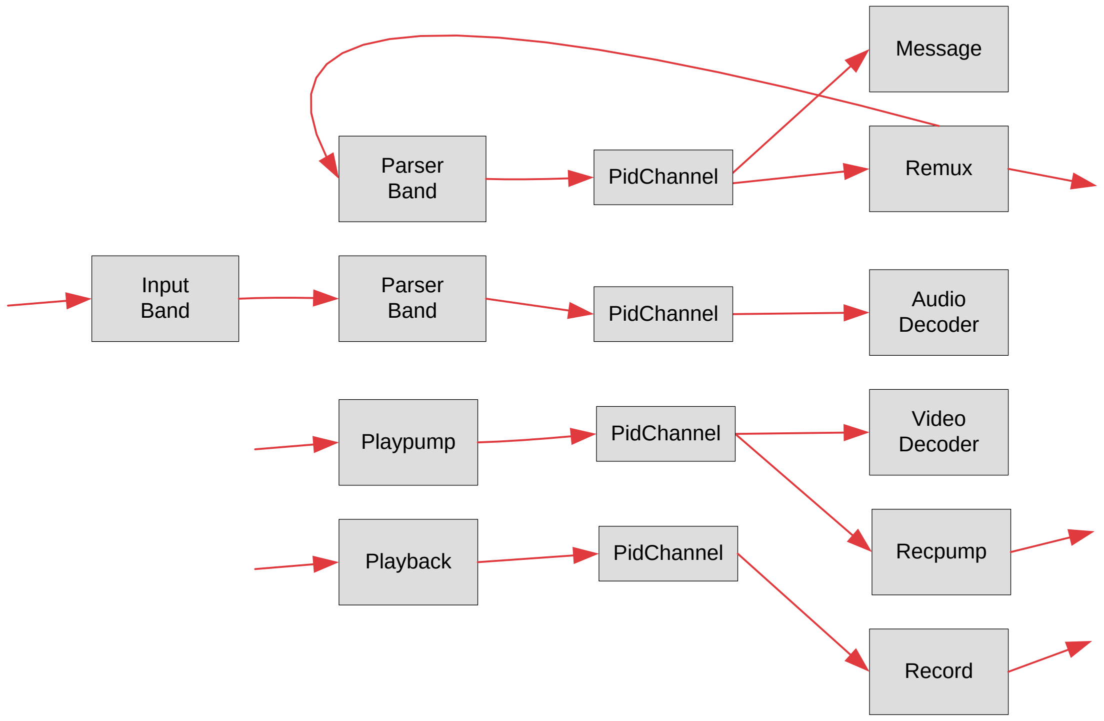


Figure 2: Transport Interfaces

[5]If a feature requires a Magnum ISR function to feed data or state directly into another Magnum ISR function, then this must be implemented inside Nexus to avoid the context switch. This is already handled for cases like picture ready, user data, VBI, and other internal data flows. If another case is needed, we will implement the appropriate Nexus code.

External devices are mapped to input bands by the design of the board or (on many Broadcom reference boards) through a configurable FPGA. Any FPGA control is performed in the Platform layer, not a Nexus module.

PidChannel

A PID channel routes one PID from a parser band to a transport consumer.

A `NEXUS_PidChannelHandle` can be opened in three ways:

- `NEXUS_PidChannel_Open` — used for live streams
- `NEXUS_Playpump_OpenPidChannel` — used for low-level DVR
- `NEXUS_Playback_OpenPidChannel` — used for high-level DVR

The reason for the playpump/playback-based `OpenPidChannel` functions is that the API needs more information per-PID in order to do DVR. You cannot open PIDs for playback using `NEXUS_PidChannel_Open`.

Some platforms (for example, the BCM740X) do not allow duplicate PID channels for the same PID and parser band. Nexus manages this using two techniques:

- Reference-counted PID channel handles. If you open the same PID/parser band, you will get the same handle back. You must close both for the hardware PID channel to be freed.
- Filter groups for multiple message filters on the same PID channel.

Timebase and StcChannel

The NEXUS Timebase interface manages a clock. This clock may be locked to a PCR or to XXXXXX clock recovery. `NEXUS_Timebase` corresponds to a DPCR hardware block. The timebase can be programmed from a variety of sources including a PCR PID, system clock, or various analog inputs. The timebase is used by a `NEXUS_StcChannel` for lipsync as well as various audio and video output interfaces.

See

Lipsync for more information about StcChannel.

Message Filtering

A PID channel can be routed to the Nexus Message interface for PSI section filtering as well as PES and TS capture. The Message interface uses the `GetBuffer`/`ReadComplete` paradigm. See

General API Usage for an overview of this paradigm.

Each message filter has a dedicated buffer. A single pid channel can be routed to multiple message buffers. Only messages that match the filter will be added to the NEXUS_MessageHandle's buffer.

PES and TS capture are special cases. The majority of the comments in this section (anything that mentions "message") only apply to PSI section filtering. Also, if you are doing TS capture, consider using the Recpump message instead.

The hardware will only write whole messages into the message buffer. If there is not enough memory to write the next message (usually because the application does not evacuate the buffer in time), an overflow event will occur. As more space becomes available, new messages will be added to the buffer.

The application can filter for particular messages by setting a `NEXUS_MessageFilter`, which has bit-wise inclusion and exclusion logic. All messages which match the filter will be added to that filter's buffer.

The `NEXUS_MessageSettings.maxContiguousMessageSize` feature allows applications to assume that all PSI messages will be contiguous in memory. This makes for simpler application code. Nexus achieves this by using a small secondary buffer to concatenate a message that spans a buffer wrap. `maxContiguousMessageSize` feature only applies to PSI data; PES packets of any size may be discontinuous on wraparound.

Most PSI parsing applications require the dynamic use of many message filters. To avoid deadlock, memory fragmentation and performance problems, consider the following suggestions:

- Open your maximum number of message filters at system initialization and do not close them until system shutdown. This avoids fragmentation and deadlock.
  - If you must open and close at runtime, be sure that all message filtering is stopped before closing and consider doing your open buffer management using `NEXUS_MessageStartSettings.buffer` to avoid fragmentation.
  - Do not acquire a mutex inside the message data ready callback which is also acquired by your application before it calls NEXUS_Message_Close. If you do, your application could create deadlock.
- Call `NEXUS_Message_GetBuffer`/`ReadComplete` as many times as needed directly inside the DataReady callback to completely process all available data. Your application may do some PSI processing in the callback if it is fast. You are also allowed to call `NEXUS_Message_Stop` from inside a callback. You cannot call `NEXUS_Message_Close` from inside a callback.
- Avoid copying message data unless required. Consider initially processing data in place and only copying when you know data must be saved.

RAVE

Transport RAVE configuration is handled internally in Nexus. When you connect a `NEXUS_PidChannel` to a VideoDecoder, AudioDecoder, Recpump, or Record, a RAVE context is allocated and configured.

If you are doing BBS debug, you may be interested in how RAVE contexts are being allocated. This can be learned by enabling `msg_modules=nexus_rave` and looking for the RAVE index messages on the console.

A typical RAVE allocation scheme is as follows:

1. Main CDB for VideoDecoder
2. Software RAVE manipulation of ITB for VideoDecoder (used for DivX®, VC1 simple/main)
3. CDB for StillDecoder
4. AudioDecoder (decode channel)
5. AudioDecoder (pass-through channel)
6. Record 0 (including all video, audio, subtitles, etc. for a program)
7. Record 1 (including all video, audio, subtitles, etc. for a program)
8. Record 2 (including all video, audio, subtitles, etc. for a program)
9. Record 3 (including all video, audio, subtitles, etc. for a program)

Video

Connections

NEXUS_VideoInput connects video sources up to the VideoWindow. A VideoWindow is opened from its Display. A Display can have one or two VideoWindows. NEXUS_VideoOutput is used to connect various video outputs to the Display.

PIP System Using Two VideoWindows on One Display shows a PIP system using two VideoWindows on one Display.

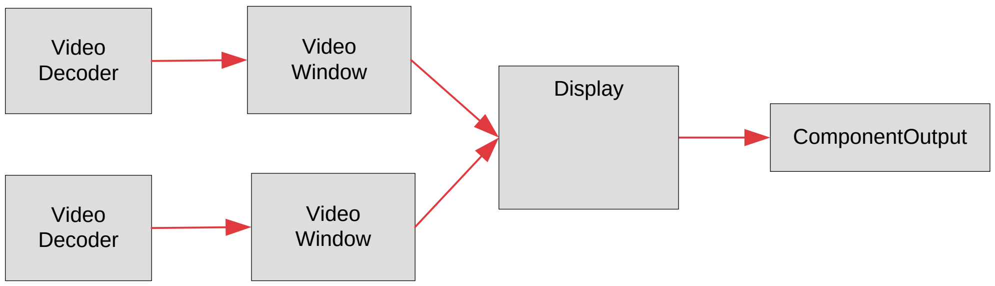


Figure 4: PIP System Using Two VideoWindows on One Display

HD/SD Simultaneous Output from a Single Decoder shows HD/SD simultaneous output from a single decoder.

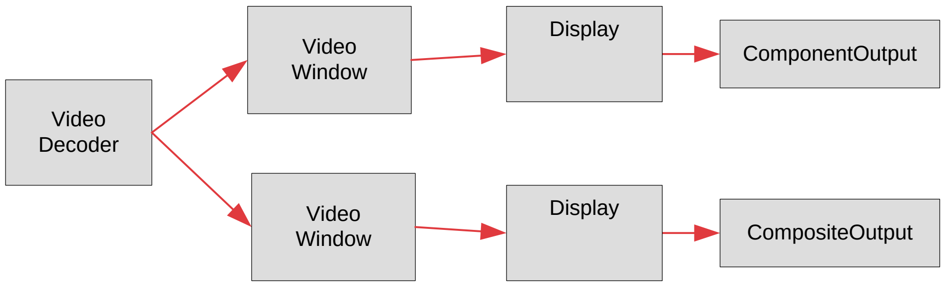


Figure 5: HD/SD Simultaneous Output from a Single Decoder

VideoDecoder

VideoDecoder take compressed digital data from a transport PidChannel, decodes the stream, then routes the uncompressed pictures to a VideoWindow.

Transport's RAVE hardware is handled internally to Nexus. There is no Nexus public API for RAVE configuration.

The VideoDecoder has a trick mode interface that allows the Playback module to perform trick modes like pause, slow motion, fast forward, and rewind.

Still Decoder

The StillDecoder interface decode still pictures using the same AVD hardware used for live decode. The AVD firmware will decode the still with remaining bandwidth without disturbing the main decode. Because of this, the bitrate of the live decode will affect the performance of the still decoder. It may take between 1 and 6 vsyncs (i.e., 15 to 100 milliseconds) to decode one still.

A “still picture” is an I-frame of any codec that the VideoDecoder interface and underlying AVD hardware supports. This includes MPEG, AVC, and VC1. This does not include JPEG and PNG. There is a separate Picture Decoder that handles these formats.

Because the still is fed to the StillDecoder through the media framework and transport, the still can be embedded in any stream format that Nexus supports, including TS, PES, ASF, AVI, or just ES.

The decoder requires that the still picture be terminated with a SeqEnd code (for MPEG) or an EOS (for AVC) in the stream. The application must append this if the picture does not already have it.

The compressed still is fed into the VideoDecoder using the Playpump or Playback interface, then is routed to the decoder using a PidChannel. The decoded still is placed into memory and made available to the application as a NEXUS_Surface in YCrCb 4:2:0 format. This can be converted to an RGB pixel format using the NEXUS_Graphics2D interface. This is shown in The Still Image Decoder.

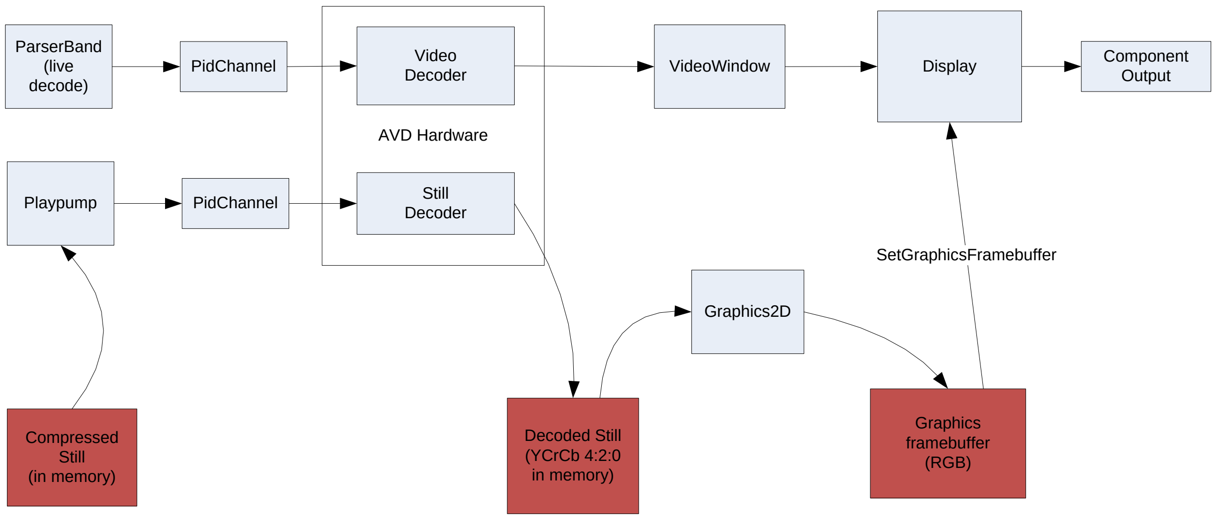


Figure 6: The Still Image Decoder

Refer to nexus/examples/video/still_decoder.c for a sample application. Also see BSEAV/app/thumbnail for a higher level application that extracts and decodes stills from a stream.

Picture Decoder

The Picture Decoder interface uses the Still Image Decoder (SID) to decode JPEG, PNG, and other compressed image formats.

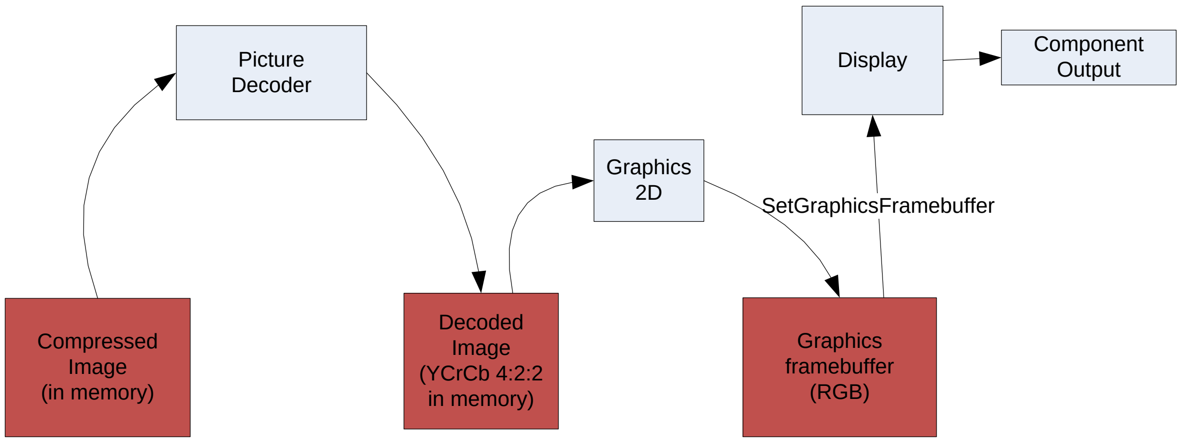

Picture Decoder
Display
Component Output
Graphics 2D
SetGraphicsFramebuffer
Compressed Image (in memory)
Decoded Image (YCrCb 4:2:2 in memory)
Graphics framebuffer (RGB)

Figure 7: Using the Still Image Decoder

The PictureDecoder interface and its SID hardware should be distinguished from the StillDecoder interface and its AVD hardware. See the previous section for the StillDecoder.

Because still images decoded by the PictureDecoder are not fed through the media framework or transport, the image must be in its native file format. The SID will read the JPEG or PNG header and use that information to decode the image.

Refer to nexus/examples/graphics/picture_decoder.c for a sample application.

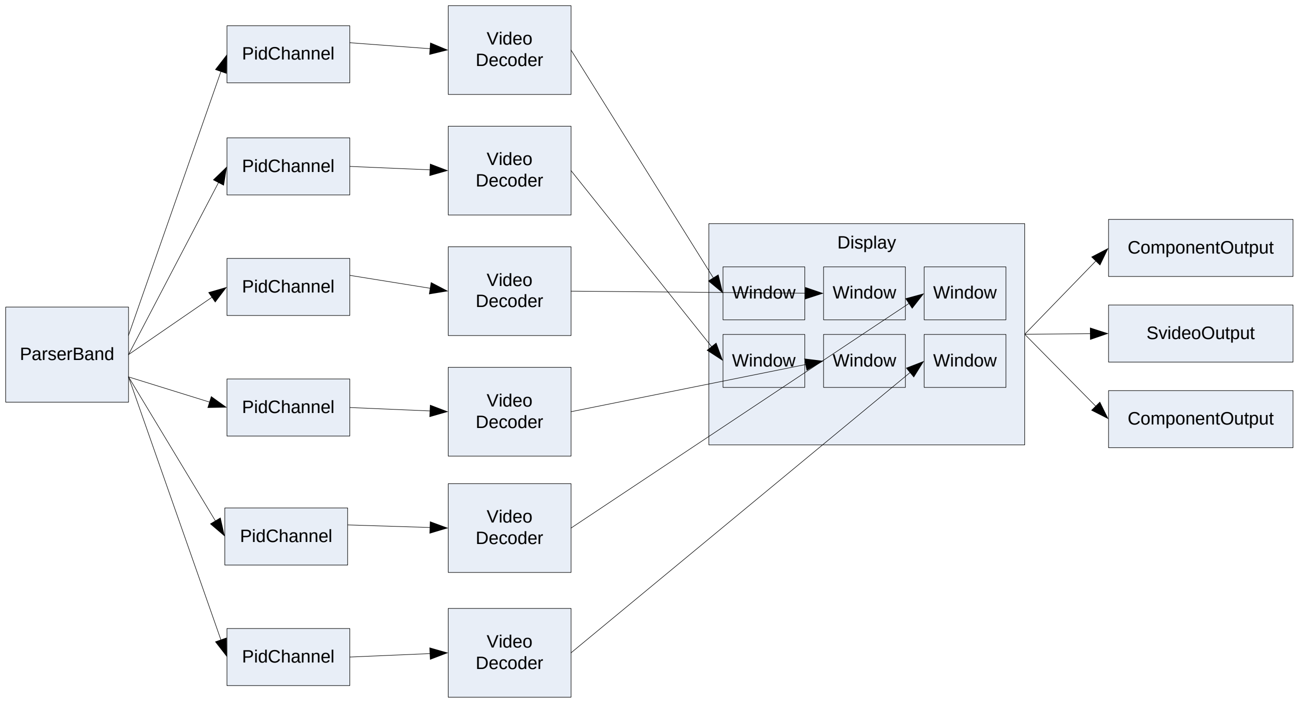
Mosaic Mode

In mosaic mode, a single AVD hardware core uses time-slicing to decode multiple lower-resolution streams at the same time. Each stream is fed into AVD using a separate PID channel and RAVE context. After decoding the streams, the VideoDecoder SW feeds a linked list of mosaic pictures to the Display SW for output to the display. The Display module programs the MPEG feeder (MFD) to scan out each picture into a single VideoWindow.

Nexus represents each mosaic decoder with a NEXUS_VideoDecoderHandle. It must be opened with NEXUS_VideoDecoder_OpenMosaic and the possible features are reduced. Nexus represents each mosaic window with a NEXUS_VideoWindowHandle. It must be opened with

NEXUS_VideoWindow_OpenMosaic, has some special functions and its features are reduced as well.

A typical mosaic filter graph looks like this:



After setting up the filter graph, the basic control of mosaic mode decoders and windows is very similar to non-mosaic decoders and windows. Functions like NEXUS_VideoDecoder_Start, NEXUS_VideoDecoder_GetStatus, NEXUS_VideoWindow_AddInput, and NEXUS_VideoWindow_SetSettings work in the same way.

See nexus/examples/video/mosaic_decode.c for a live decode mosaic example. See nexus/examples/dvr/mosaic_playback.c for a playback mosaic example.

One important difference between live and playback mosaic mode is in TSM (time stamp management) control. In playback mode, each mosaic VideoDecoder gets its own StcChannel, but each channel must point to the same underlying HW STC (NEXUS_StcSettings.stcIndex) and the same DPCR (NEXUS_Timebase). For live, mosaics with the same DPCR must also share a NEXUS_StcChannel. These differences are illustrated in the example applications. When programming mosaic mode, first use vsync mode (NEXUS_VideoDecoderStartSettings.stcChannel = NULL) for basic functionality, then add TSM.

Audio Connections

NEXUS_AudioInput connects audio sources up to an AudioMixer. Audio output uses the Mixer Interface to control the routing of data from various producers to various outputs. Connections are made using NEXUS_AudioInput and NEXUS_AudioOutput. A typical configuration is shown in Audio Output Mixer Interface.

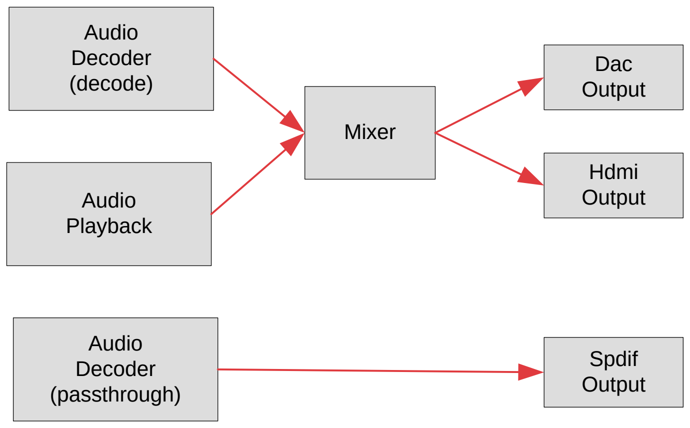


Figure 8: Audio Output Mixer Interface

This filter graph can be constructed as follows:

```
# mix decoded audio and PCM
NEXUS_AudioMixer_AddInput(mixer,
    NEXUS_AudioDecoder_GetConnector(audioDecoder));
NEXUS_AudioMixer_AddInput(mixer,
    NEXUS_AudioPlayback_GetConnector(pcmplayback));
NEXUS_AudioOutput_AddInput(
    NEXUS_AudioDac_GetConnector(audioDac),
    NEXUS_AudioMixer_GetConnector(mixer));

# connect compressed passthrough
NEXUS_AudioOutput_AddInput(
    NEXUS_SpdifOutput_GetConnector(spdifOutput),
    NEXUS_AudioDecoder_GetConnector(passthroughDecoder));
```

Depending on the hardware capabilities, much more complicated filter graphs can be constructed using Encoders, Pre and Post Processors, and multiple Mixers.

AudioDecoder

AudioDecoder take compressed digital data from a transport PidChannel, decodes the stream, then routes the uncompressed PCM data to an AudioMixer, any audio postprocessing block or an AudioOutput.

Transport's RAVE hardware is handled internally to Nexus. There is no Nexus public API for RAVE configuration.

The AudioDecoder has a trick mode interface that allows the Playback module to perform trick modes like 0.5x - 2.0x pitch-corrected playback.

Channel Change

Order of Events and Possible Optimizations

A channel change operation involves the following events:

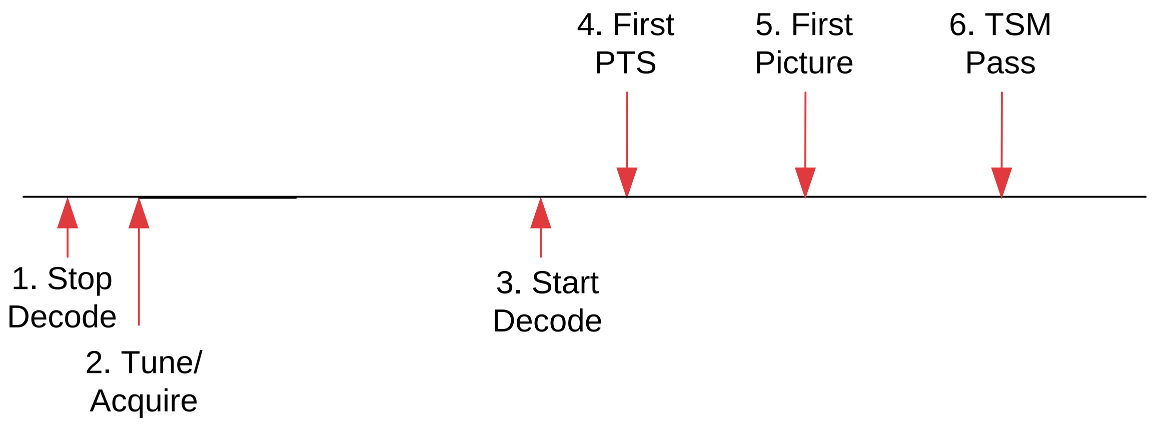


Figure 9: Order of Events for a Channel Change

1. Application stops decode. At this point, video stops moving.

   Speed up: The application can set NEXUS_VideoDecoder_ChannelChangeMode_eHoldUntilFirstPicture to continue showing the last picture until the First Picture of the next channel (5) or it can set eHoldUntilTsmLock to continue until TSM Pass for the next channel (6).

2. Application tunes and acquires the frontend for the new channel. The time between 1 and 2 should be negligible.

3. After application learns PID's and codecs, it can open PID channels and start decode. The time between 2 and 3 is the time to tune and acquire plus the time to find and parse PSI.

   Speed up: Some frontends have "fast acquisition" options. The application can cache PID and codec information. If PSI scan must be performed, the application can start it before the frontend is acquired so that the XPT MESG block sees data without the time to process an application callback.

4. The decoder sees the first PTS. The time between 3 and 4 is based on stream muxing. The HVD must see the next PES header to get the PTS. For live, the first PTS does not affect start of decode. For playback, the first PTS is used to set the STC for playback TSM.

5. The decoder sees the first fully decoded picture (i.e., RAP = random access point). The time between 4 and 5 is based on stream muxing. The HVD must see any required metadata (for

example, SeqHdr for MPEG, SPS/PPS for AVC) and must see an I-frame. This is the first picture that can be displayed without prediction errors.

Speed up: Set NEXUS_VideoDecoderSettings.prerollRate to display some pictures before TSM Pass (6).

6. The display receives the first moving pictures that are lipsynched. The time between 5 and 6 is based on the PCR/PTS difference determined by stream muxing, additional PTS offset added to the system for lipsync, and possible muting as SyncChannel does high precision lipsync. For playback TSM, the STC is set using the stream's PTS, therefore the delay for this step is minimal.

Speed up: Use "Fast Channel Change" (in the following section) to have adjacent channels already primed so that steps 1 through 5 are already done and step 6 is minimized. SyncChannel muting can also be disabled, but video or audio blips may be seen or heard.

Fast Channel Change

You can perform a fast digital channel change (FCC) by having your application "prime" adjacent channels in your channel map.

The steps in the process are:

1. Start decoding channel X.
2. Start VideoDecoderPrimer on channel X + 1. This requires having another tuner and parser band available, along with any required CA decryption.

   Optionally the app could start FCC primers on channel X – 1, X + 2, etc. For purposes of this example, we will use a single-direction, single step primer.
3. The user presses channel up (i.e., goes to channel X + 1)
4. Stop decode on channel X.
5. Call NEXUS_VideoDecoderPrimer_StopPrimerAndStartDecode on channel X + 1
6. Start VideoDecoderPrimer on channel X + 2.

The video primer gets the next channel ready by consuming data from the RAVE context, just like a video decoder would. The primer performs basic TSM (comparing PTS to STC) so that the buffer is very close to normal lipsync levels. When channel change occurs, the primed RAVE context is given to the video decoder but is not flushed when decode is started.

The video primer supports MPEG, AVC and HEVC video codecs.

RAVE Contexts and PCR_OFFSET Blocks shows the RAVE contexts and PCR_OFFSET blocks used in a FCC system.

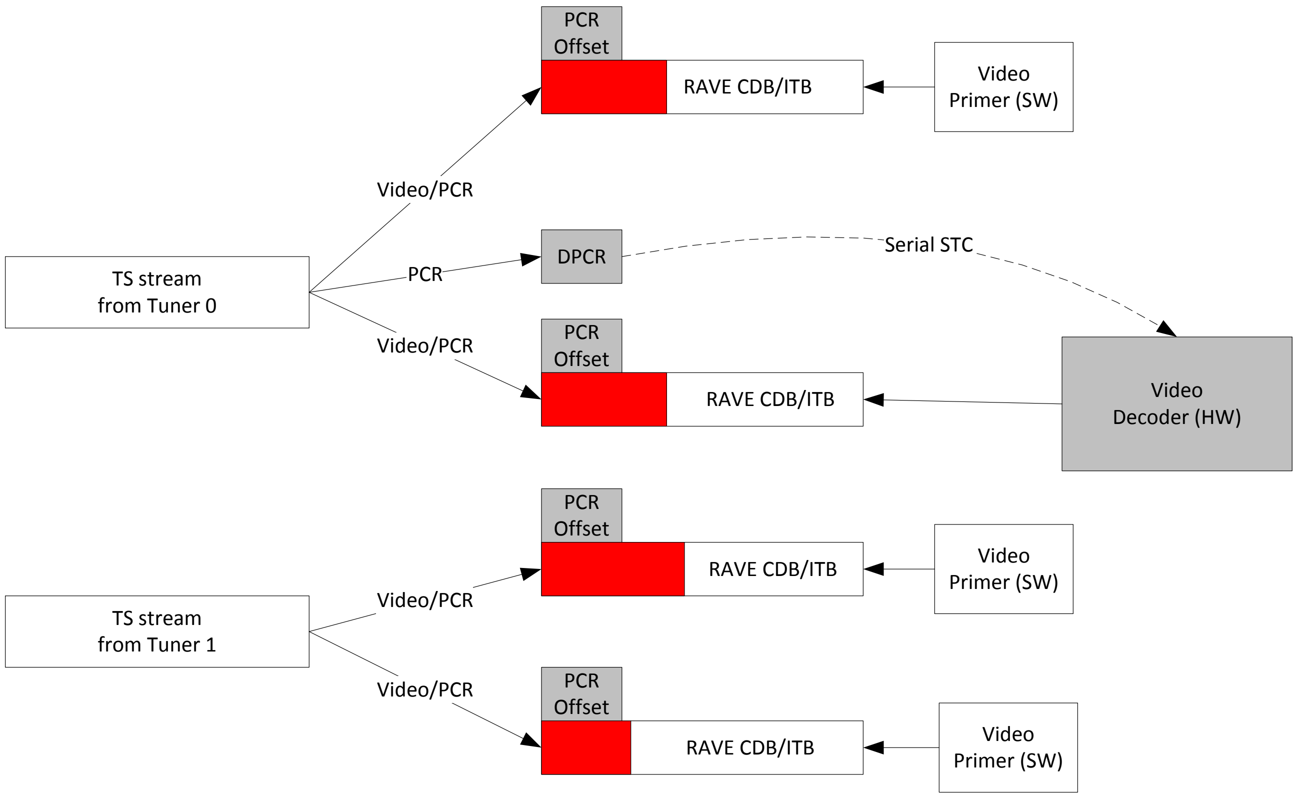


Figure 10: RAVE Contexts and PCR_OFFSET Blocks

When using the FCC primers, the application must consider that:

- The bandwidth through transport will be the sum of the currently decoding channel plus all primed channels. Use separate parser bands so that bandwidth is not exceeded.
- Conditional Access (CA) decryption must be performed on the currently decoding channel plus all primed channels.

See nexus/examples/video/video_primer.c for an example application.

Calling Sequence for Live Decode

- `NEXUS_VideoDecoder_Stop`
- `NEXUS_PidChannel_Close`
- `NEXUS_Frontend_TuneXxx` (for example, Vsb, Qam, etc.)
- `NEXUS_PidChannel_Open`
- `NEXUS_VideoDecoder_Start`

Calling Sequence for DVR

Starting and stopping DVR is a form of channel change.

To stop DVR:

- `NEXUS_VideoDecoder_Stop`
- `NEXUS_Playpump_Stop`
- `NEXUS_Playpump_ClosePidChannel`

To start DVR:

- `NEXUS_Playpump_OpenPidChannel`
- `NEXUS_Playpump_SetSettings` (some settings must be set before NEXUS_Playpump_Start)
- `NEXUS_Playpump_Start`
- `NEXUS_VideoDecoder_Start` (this must be called after starting the NEXUS_Playpump commands)

General Notes on Calling Sequences

`NEXUS_VideoWindow_RemoveInput` and `NEXUS_VideoWindow_AddInput` are only required when switching input types (for example, switching from analog to digital mode). This switch requires additional load time and should be avoided if possible.

`NEXUS_VideoDecoder_Close` should not be called on channel change. They are not required and will cause additional channel change time and possible memory fragmentation.

`NEXUS_Playpump_Close` should not be called on DVR change. It will cause memory fragmentation.

Lipsync

TSM

Time Stamp Management (TSM) achieves basic lipsync by having the decoder release pictures or audio frames only when the picture or audio frame's PTS matches the STC within a certain threshold. The TSM algorithm is run inside the Magnum XVD porting interface (for VideoDecoder) and the audio DSP (for Audio).

The XVD TSM algorithm is summarized in The XVD TSM Algorithm.

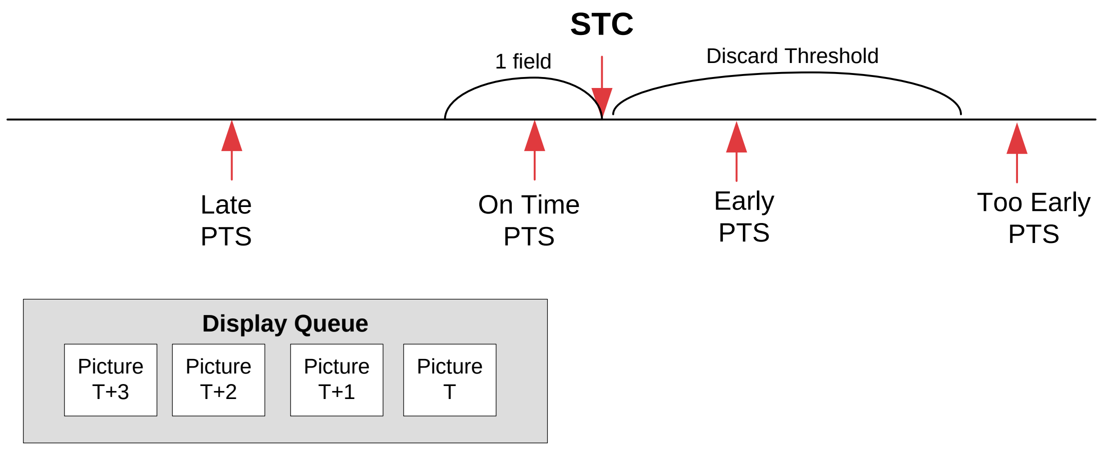


Figure 11: The XVD TSM Algorithm

1. PTS of next picture (Picture T) is compared with STC.
2. If PTS > STC, the picture is early. The decoder will wait for the PTS to mature.
3. If PTS > STC – one picture (i.e., either field or frame) and PTS <= STC, the picture is on time. It is displayed immediately. This one picture threshold is not programmable.
4. If PTS < STC – one picture, the picture is late. The decoder will walk through every picture in the display queue (for example, 5 for HD, 15 for SD) and find the least late picture. That becomes the current picture T and is displayed. All pictures before that picture will be discarded. This allows the decoder to never be stuck, regardless of how bad the stream may be.
5. If a late picture occurs three times in a row, a PTS error is raised. No action is required for this error. However, ASTM monitors PTS errors, along with PTS and PCR values and decides if it should switch to vsync mode. See ASTM documentation for more details.
6. The "discard threshold" defines when a picture is "too early". The low-level software defaults this to 2 seconds for MPEG; 10 seconds for VC1 and AVC. Nexus does not expose this value application control.

•The DPCR hardware block will absorb a certain amount of PCR jitter so that the STC remains smooth. The default for MPEG TS is 2.8 ms. See `NEXUS_TimebaseSettings.sourceSettings.pcr.maxPcrError` for fine tuning.

StcChannel

The StcChannel Interface provides the interconnections required for TSM (basic lipsync) for both audio and video. Each StcChannel manages a single PCR_OFFSET hardware block.

The StcChannel interface sets the System Time Clock (STC) using either the current PCR value (for live decode) or one of the decoder's current PTS value (for DVR). After setting the STC, the decoders will wait until their picture's or audio frame's PTS matches the STC within a certain threshold.

Live Decode Using a PCR PID illustrates a live decode using a PCR PID.

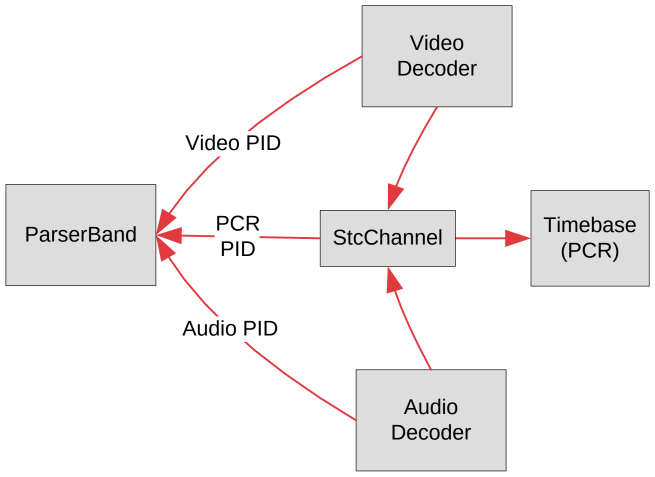


Figure 12: Live Decode Using a PCR PID

The following illustrates a DVR decode. No PCR is used and the Timebase is locked to the system clock (i.e., free run).

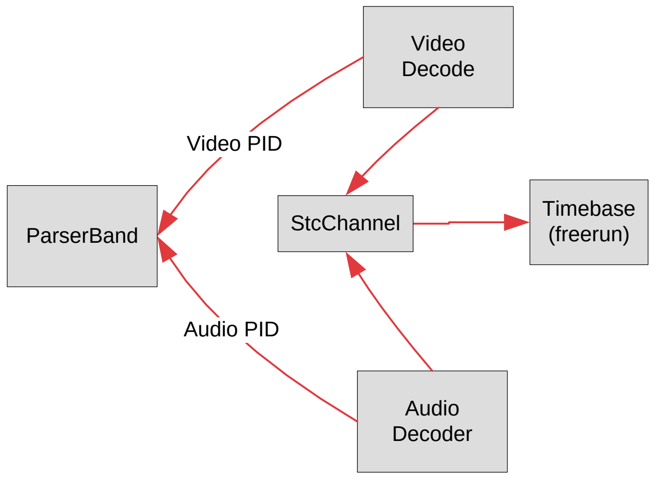


Figure 13: DVR Decode

Refer to nexus/examples/decode.c and nexus/examples/dvr/playback.c for an example of live and DVR basic lipsync.

SyncChannel

High-precision lipsync is accomplished by adding the SyncChannel Interface. StcChannel is required for basic TSM, then SyncChannel is optional in addition to that. SyncChannel monitors and adjusts the decoders and display/mixer settings in order to bring audio and video timing into very close proximity (for example, ±20 ms).

A typical digital decode system is shown in Typical Digital Decode System.

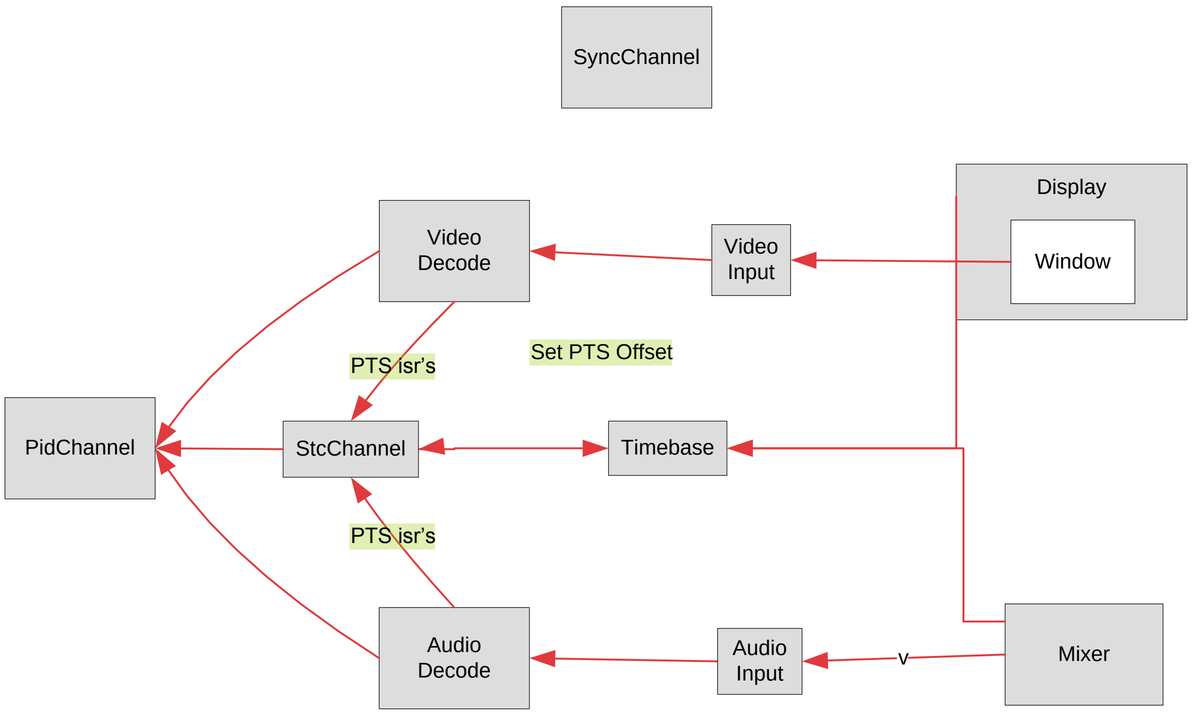


Figure 14: Typical Digital Decode System

The SyncChannel module provides high-precision lipsync by making small adjustments in audio, video, and display in response to a variety of feedback from those same components.

Refer to nexus/examples/lipsync/sync_channel.c for an example.

Adaptive System Time Management (ASTM)

The Adaptive System Time Management (ASTM) allows the system to adapt to bad TSM environments including bad PCRs, missing PCRs, bad PTS's, etc. ASTM monitors the system and moves the decoders into and out of TSM mode as needed. The goal is to keep seeing and hearing the audio and video with the best lispsync possible, even if TSM is not possible.

StcChannel is required for basic TSM, then ASTM is optional in addition to that. You can use ASTM with or without SyncChannel.

Typical Digital Decode System illustrates how ASTM relates to the rest of the system.

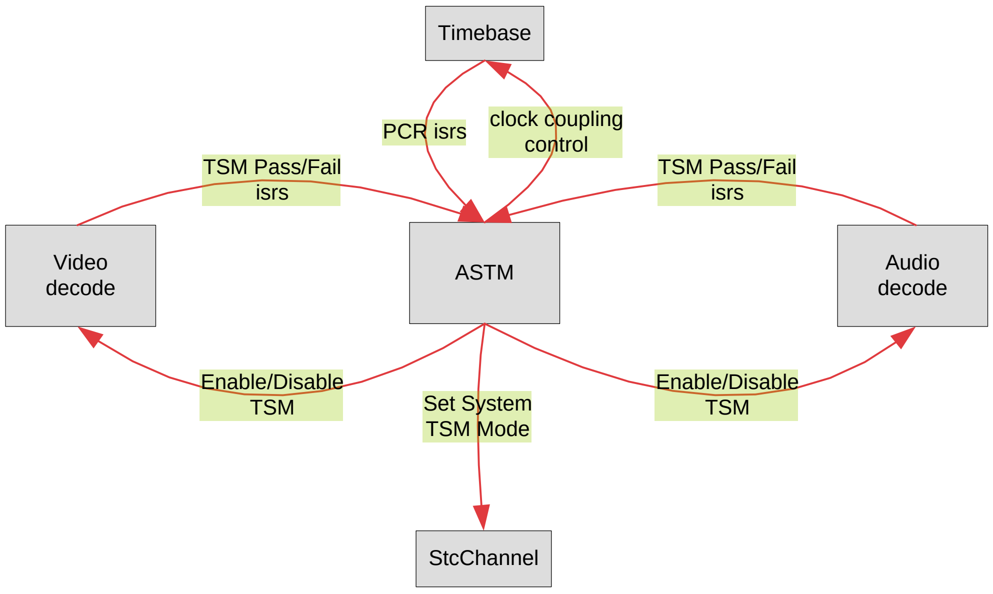


Figure 15: The Relationship of ASTM to the System

The ASTM algorithm can be summarized as follows:

- Presentation Control:
  - ASTM monitors both audio and video decoders looking for TSM failures.
  - A TSM failure is defined as a certain number of PTS errors. The programmable threshold defaults to ten.
  - If both decoders are failing, ASTM will try a PTS master mode first (i.e., video master or audio master).
  - If only one decoder is failing, ASTM will enter directly into non-TSM mode (i.e., vsync mode).
  - If either decoder is still failing after trying a master mode, it will switch to non-TSM mode.
- Clock Recovery (live decode only):
  - ASTM monitors the receipt of PCRs looking for high jitter

- . The jitter tolerance threshold and consecutive out-of-tolerance counter are programmable.
- If the incoming clock reference has high jitter, ASTM will decouple the timebase from the input clock (and couple it to the internal crystal)
- If the incoming clock reference returns to low jitter, ASTM will recouple the timebase to the input clock

If the application manually puts a decoder into vsync mode, ASTM will not move it into TSM mode. The application can also manually set the three controls ASTM provides: presentation control (PTS or output clock); STC source (PCR or PTS, with which PTS is used being a preference); and clock coupling (input or crystal). Any manual setting will not be overridden by ASTM.

ASTM is supported for both live decode and DVR.

Refer to nexus/examples/lipsync/astm.c for an example.

Low Latency Decode

Some applications need to get content from system input to system output with as little delay as possible. For example, a video game routed through a set-top needs minimal delay between user input and the on-screen result. Achieving low latency always involves trade-offs. The delays present in the system enable audio/video lipsync, high quality de-interlacing or frame rate conversion. Certain delays are also inherent in a pipelined architecture which requires minimal CPU interaction.

The following are some points of control:

The video decoder has several options in NEXUS_VideoDecoderExtendedSettings. It can be configured to not use prediction buffers (zeroDelayOutputMode), but the stream must be I-frame only. It can deliver a picture to the display before its writing is complete (earlyPictureDeliveryMode) or deliver a picture to the display before its PTS matures (lowLatencySettings.mode). These options compromise lipsync with audio. The “mode” setting has options to keep some delay to allow for higher quality frame rate conversion in high jitter contexts like IP streaming.

Transport can introduce a vsync of delay because HVD does not decode a picture until it sees the start code for the next picture or an end of sequence code. For AVC video, two AUD NAL (00 00 01 09) start codes followed by 188 bytes of zero padding can be inserted at the end of a picture to cause HVD to read immediately. No technique exists for MPEG.

The deinterlacer can be disabled (NEXUS_VideoWindowMadSettings.deinterlace) or its quality can be reduced with a spatial-only setting (NEXUS_VideoWindowMadSettings.gameMode). On some 40nm systems, BVN capture buffers can be disabled (NEXUS_VideoWindowSettings.forceCapture), but video must always be full-screen.

The audio decoder can be set into a low delay mode (NEXUS_AudioDecoderStartSettings.latencyMode) which compromises lipsync with video. A one-frame delay can be removed by telling the decoder to not wait for the next frame sync (NEXUS_AudioDecoderStartSettings.targetSyncEnabled) which compromises robustness.

See nexus/examples/video/decode_video_low_latency.c and nexus/examples/audio/audio_decode_low_latency.c for example code.

VBI and Userdata

Digital Userdata capture

The VideoDecoder interface provides a `GetBuffer`/`ReadComplete` interface for reading raw userdata from a video stream. This data can be parsed by an application for CC608/708 rendering, DVB subtitles, teletext, and other uses.

NEXUS_VideoDecoder_GetUserDataBuffer returns a pointer to heap memory which contains the raw userdata. The data in the buffer is always in blocks of header-plus-payload. The application parses the header, learns the payload size, and then parses the payload. NEXUS_VideoDecoder_UserDataReadComplete is called to consume both header and payload.

The application is responsible to read the userdata fast enough to avoid overflow. Any overflow will result in an ERR message printed to the console. The bitrate of most userdata is low, so this is not a major concern. The size of the userdata buffer can be increased if needed using `NEXUS_VideoDecoderOpenSettings.userDataBufferSize`.

Refer to nexus_video_decoder_userdata.h for the API. See nexus/examples/video/userdata.c for a working example.

Automatic VBI parsing and routing of Userdata

The VideoDecoder module can also internally parse userdata for EIA-608 and 708 closed captioning using the magnum/commonutils/udp library. Parsing and routing only works for EIA-608 data because only 608 data can be encoded in analog video VBI lines. The parsing-only mode works for EIA-608 and 708 data and is used for graphical rendering.

To enable automatic parsing and routing of EIA-608 data to the analog video VBI lines, two Booleans must be set: NEXUS_DisplayVbiSettings.closedCaptionEnabled enables the analog VBI encoding on the display, and NEXUS_DisplayVbiSettings.closedCaptionRouting enables the routing of EIA-608 from the decoder to the display.

The application also has an option of using the internal EIA-608/708 parser without routing to the display. The data can be retrieved using NEXUS_VideoInput_ReadClosedCaption. The application must set NEXUS_VideoInputVbiSettings.closedCaptionEnabled to enable this. nexus/examples/video/userdata.c demonstrates this mode as well. See the following section on graphical rendering of Closed Caption for a typical use case.

VideoDecoder Userdata Format Filter

Digital userdata can contain multiple formats of userdata. Internal parsing and rendering must select only one format; otherwise the resulting VBI data would be garbled. The Nexus VideoDecoder module provides as NEXUS_UserDataFormat enum as well as a NEXUS_VideoDecoder_SetUserDataFormatFilter function to manage the filtering processing.

If your application is having trouble filtering its data, or if the internal parser does not support your data, we recommend that you do raw userdata capture and write your own filter. This puts the application in full control of userdata and VBI.

Graphical rendering of Closed Caption data

The reference software includes a library, written above Nexus, which can assist with graphical rendering of EIA-608/708 data to the graphics plane.

See BSEAV/lib/dcc. See or request rockford/unittests/applibs/dcc/eia708_app.c as a single-process sample application. Also, see nexus/nxclient/apps/ccgfx for an example application which uses this library in a multi-process environment.

Application VBI Encode

The application can also write its own data to the display's VBI encoder. This can include closed caption, teletext, WSS, CGMS and others. Refer to the functions in nexus_display_vbi.h. The application should not be automatic routing from decoder to display in addition to writing its own closed caption data; the VBI buffers will likely overflow.

See nexus/examples/video/display_vbi.c for a sample app.

HDMI Input and Output

Chips with HDMI receivers (for example, the BCM7425) have an HdmiInput module. The HdmiInput module provides both a NEXUS_VideoInput and NEXUS_AudioInput connector by means of GetConnector functions. Call AddInput on the respective Display and Mixer to hear and see the HDMI input.

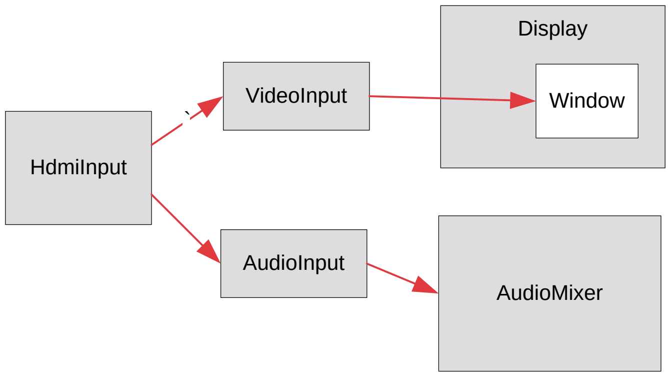


Figure 19: HDMI Input

Chips with HDMI transmitters (for example, the BCM7400 and BCM7405) have an HdmiOutput module. The HdmiOutput module provides both a NEXUS_VideoOutput and NEXUS_AudioOutput connector by means of GetConnector functions. Call AddOutput on the respective Display and Mixer to hear and see the HDMI output.

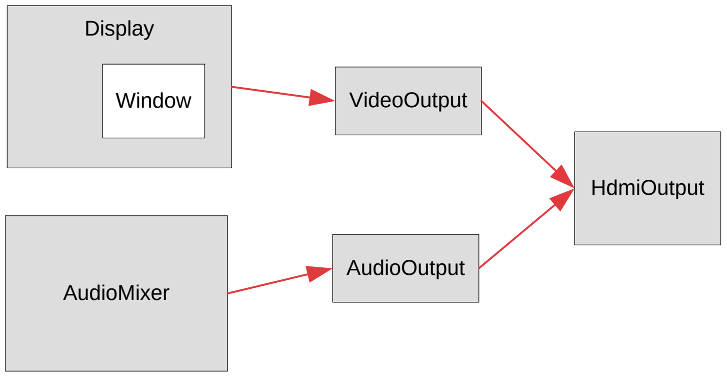


Figure 20: HDMI Output

Refer to nexus/examples/hdmi_input and nexus/examples/hdmi_output for sample code.

Dynamic Range and Mastering

The dynamic range of an image is the difference in brightness between its darkest part and its lightest part. The human eye can discern discrete objects over a large range of brightness (such as in bright sunlight or dim starlight). Newer recording equipment can capture increased levels of dynamic range in brightness, which yields better details in both brighter and darker areas of a scene. Along with improvements in recording have come improvements in rendering, and there are a number of newer digital televisions on the market today that can render with higher dynamic range than previously available.

Currently, most video content produced for broadcast is classified as Standard Dynamic Range (SDR). SDR is defined as light that was encoded to pixel values using only 8 bits per sample and using what is called a “conventional gamma curve” as defined in ITU-T Rec. 601 or Rec. 709. This allows a maximum difference in luminance of 100 candles per meter squared (cd/m2, also called “nits”).

Almost all digital televisions available now already offer a means of internally mapping SDR input video to a larger output dynamic range (e.g., 300 nits) before the light hits the viewers’ eyes.

While professional movie production companies have long used HDR technology, television content providers are now also starting to produce video that is classified as High Dynamic Range (HDR). With the increasing availability of HDR content and advances in display devices, television broadcasters are starting to broadcast HDR-encoded video content, HDR video spans a number of light-to-pixel encodings (called opto-electrical transfer functions, OETFs) and their inverse pixel-to-light decoding (called electro-optical transfer functions, EOTFs), including those defined by SMPTE and ARIB, and a few proprietary methods.

Standards

CEA 861.3 defines how to signal which EOTF was used in the display used by the artist who graded the content, so that the display receiving and presenting said content can try to reproduce the grader’s vision of the scene as closely as possible. CEA 861.3 also details how to signal eotf-specific metadata that might be helpful to the receiving display in its quest for accurate reproduction.

PQ / HDR10

SMPTE ST 2084 defines a different EOTF curve (sometimes called Perceptual Quantizer, or PQ, and used in CEA’s HDR10 Media Profile) which is not backward compatible with SDR, but allows for luminance levels up to 10000 nits, and higher contrast detail in the SDR region and below. Because it is not backward compatible, displaying a SMPTE ST 2084 video on an SDR TV will yield resulting images that are too dark and have lost a large amount of contrast detail.

SMPTE ST 2086 defines static metadata that gives more insight into the details of the grader's display and the video content itself, including the mastering display color volume (MDCV) and the content light level (CLL). These details can be used by the receiving display to tune the picture presentation.

HLG

ARIB STD-B67 defines an EOTF called Hybrid Log Gamma (HLG). This EOTF was designed to allow backward compatibility of an HLG-encoded image with a display device that does not support HLG (e.g., an SDR-only TV). This means that an HLG image presented on an SDR display should have brightness values that make the image look normal, while on an HLG-capable display, brightnesses above the SDR range are viewable with high contrast detail.

Because HLG is based on and compatible with SDR, it does not require any metadata be signaled for it to function.

NEXUS HDR Support

Pass-through

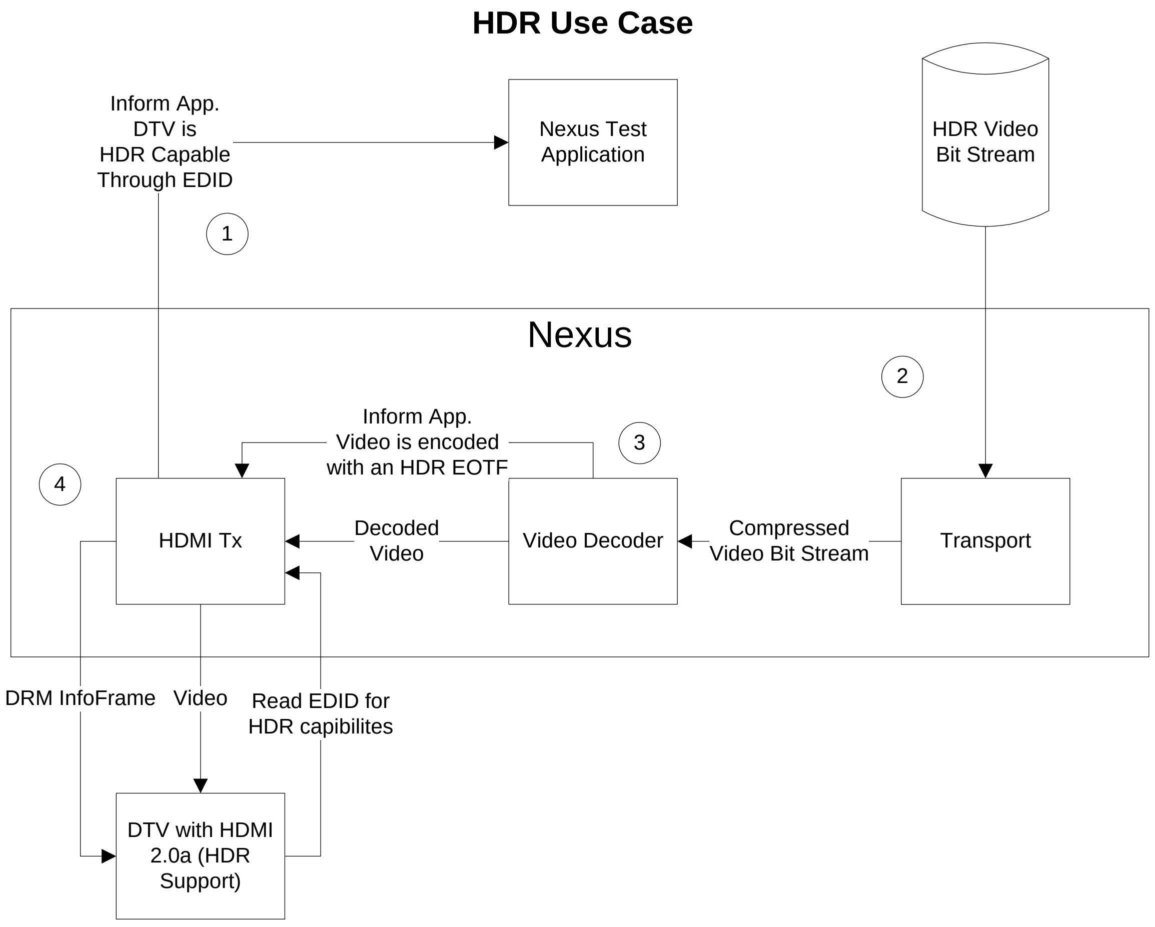
All Broadcom set-top box chips with 10-bit video pipelines currently support passthrough of HDR10 and HLG content. NEXUS will automatically determine the EOTF, mastering display color volume, and content light level specified in HEVC content and forward this information from the video decoder to the HDMI output. This information is also available to any application via NEXUS_VideoDecoderStreamInformation and its associated callback. The eotf is carried in the 'eotf' field, while the mastering display color volume is carried in the masteringDisplayColorVolume field, and the content light level is carried in the contentLightLevel field.

Once the HDMI output code internally receives the EOTF and its metadata, it looks at the EDID of the attached receiver to determine which EOTFs the receiver supports. If the input content EOTF is supported by the display, NEXUS will transmit the content as-is and create a Dynamic Range and Mastering (DRM) InfoFrame to send over the HDMI interface to the receiver to describe the EOTF and static metadata that is in the content.

If the receiver does not carry any HDR support, which is denoted by lack of an HDR DataBlock in its EDID, NEXUS will not send any DRM InfoFrame.

If the receiver does not support the specific EOTF that is used for the active content, NEXUS will signal the content's EOTF regardless, but will also present a warning on the console.

NxClient and NEXUS applications support pass-through natively. The following diagram shows a simple NEXUS test application usage mode.


HDR Use Case
Inform App. DTV is HDR Capable Through EDID
Nexus Test Application
HDR Video Bit Stream
1
Nexus
2
Inform App. Video is encoded with an HDR EOTF
3
4
HDMI Tx
Decoded Video
Video Decoder
Compressed Video Bit Stream
Transport
DRM InfoFrame
Video
Read EDID for HDR capibilites
DTV with HDMI 2.0a (HDR Support)

Operational Sequence in Nexus Test Application

1.Configure HDMI Tx as an output for the main display.

2.Configure and Start Transport and Video Decoder

-Configure to Transport demux bit stream

-Configure Video Decoder to decode demuxed video bit stream

-Start Transport and Video Decoder

3.NEXUS will determine if video bit stream is encoded with an High Dynamic Range (HDR) Electro-Optical Transfer Function (EOTF) encoding

-NEXUS uses content of transfer_characteristics SEI message to determine if video bit stream contains HDR encoding

-Extract Mastering Display Color Volume SEI message if available in stream

-Extract Content Light Level SEI message if available in stream

4.If video bit stream contains an HDR EOTF encoding, NEXUS will check the DTV's EDID to see if this form of HDR is supported, and if so it will automatically configure the HDMI Tx to transmit Dynamic Range and Mastering (DRM) InfoFrame. DRM InfoFrame conveys video's HDR EOTF encoding to DTV.

-Use the Mastering Display Color Volume and Content Light Level extracted in step 3 to fil-out DRM InfoFrame.

Conversion

Broadcom set-top box chips with HDR capabilities are incapable of converting video from SDR to HDR10 or HLG, or from HDR10 to SDR (HLG needs no conversion to SDR since it is backwards compatible). However, Broadcom set-top box chips do offer limited linear conversion of graphics from SDR to HDR10.

Broadcom EOTF conversion capability limitations imply the following corollaries:

1.Operators that embed SDR content with HDR content will be relying on the display device (and not the set-top box) to be able to handle signal switching between SDR and HDR and back. The set-top box has no control over how the display device accomplishes this, and it may involve extended periods of blanking or discoloration until the display adapts to the new EOTF.

2.User interfaces using large palettes will only be able to be tuned to have a small range of colors look as intended when converted from SDR to HDR10.

Application Override

NEXUS allows the application to override the information sent in the DRM InfoFrame, regardless of the connected receiver's reported capabilities. This is accomplished in direct NEXUS calls using the NEXUS_HdmiOutput_GetExtraSettings/SetExtraSettings function pair, setting the overrideDynamicRangeMasteringInfoFrame field to true, and then setting the contents of the dynamicRangeMasteringInfoFrame field accordingly.

For NxClient use the NxClient_GetDisplaySettings/SetDisplaySettings function pair by setting the hdmiPreferences.drmInfoFrame member fields to non-default values. For EOTF, the default is eMax. For other integer fields (e.g. MDCV and CLL fields), the default is -1. To return to the defaults, simply set the eotf field back to eMax, or the other parameters back to -1.

See nexus/examples/hdmi_output/eotf.c for an example of how to override using raw NEXUS, and nexus/nxclient/apps/setdisplay.c for an example of how to override using NxClient

DVR

Digital Video Recording (DVR), also known as Personal Video Recording (PVR), is supported with several interfaces, shown below:

Interface	Description
Playpump and Recpump	Low-level API for reading data from transport (Recpump) and writing data to transport (Playpump). Integrates the Media Framework for stream conversion. Integrates DMA and Security for DVR encryption and decryption.
File	File I/O support reading and writing stream and index data. Includes support for linear and circular mode recording.
Playback and Record	High-level DVR that extends Playpump and Recpump, integrates File I/O, and adds indexing and trick modes. Integrates the Media Framework for navigation and trick modes.
VideoDecoder	Trick mode interface
AudioDecoder	Trick mode interface
StcChannel	Needed for DVR TSM, STC trick modes, and to invalidate the STC on certain transitions.

Playpump and Recpump

The Transport module provides low-level DVR support by means of two “pump” interfaces: Playpump and Recpump. The caller must actively pump the data in or out of the interface. This is done by querying the state of the buffer, then writing data into empty space (Playpump) or reading data which is available (Recpump). The Playpump and Recpump interfaces have no file I/O, threading, or decoder control. The pump interfaces are only data feed interfaces.

These low-level interfaces allow a variety of high-level DVR options to be layered on top. Data may go to or come from a file, network stack, or memory. The user is responsible for efficient I/O outside of Playpump and Recpump. CPU-based memcpy should be avoided for any high bitrate streams. Options for efficient I/O include M2M DMA (see the NEXUS_Dma interface), SATA DMA (using Direct I/O), or using Playpump’s scatter-gather mode.

Both Playpump and Recpump are nonblocking interfaces with callbacks. The user cannot block and should not rely on polling[6]. Instead, the user should register for the data-ready callbacks and call when space becomes available or data becomes available.

[6]There are some cases where a time-based poll is helpful for record. When doing timeshifting, it’s important to evacuate the record buffer based on time, not buffer level. See the NEXUS_Record implementation for more information.

The basic state machine is shown in Playpump and Recpump State Machine.

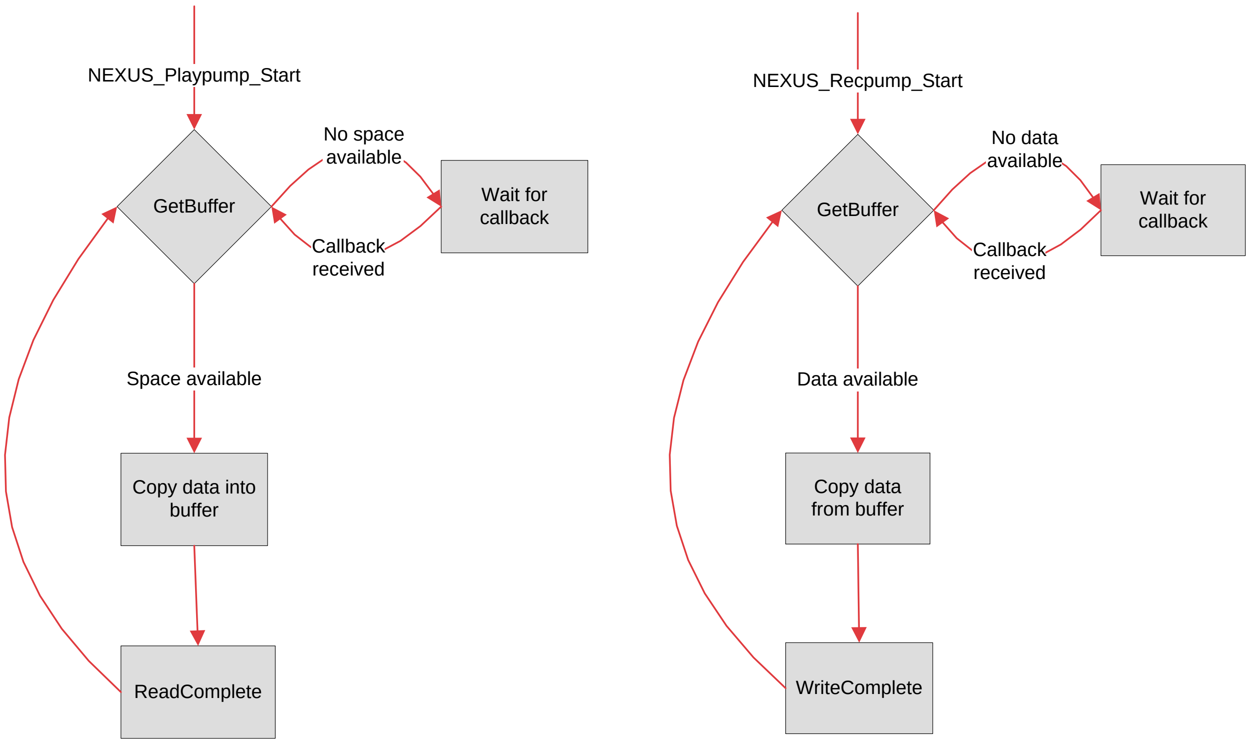


Figure 21: Playpump and Recpump State Machine

Recpump receives data from transport through a set of connected PidChannels. Playpump sends data to transport through a set of opened PidChannels. A Playpump PidChannel can be connected to any consumer including VideoDecoder, AudioDecoder, Recpump, Message, or Remux.

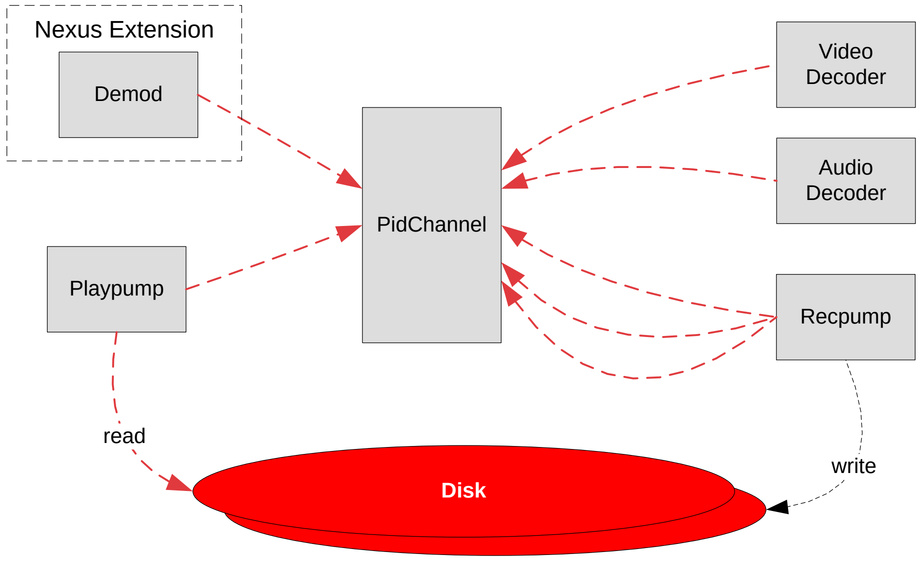


Figure 22: Playpump PidChannel Connections

You must use `NEXUS_Playpump_OpenPidChannel` to get a playpump PID channel. Even though live and playback PID channels are the same in hardware, their software configuration is different[7].

Playpump includes integration with the Media Framework to support conversion from certain stream formats (e.g., AVI, ASF) to MPEG-2 PES, which can be handled by transport hardware. See Media Framework.

Playpump and Recpump include integration with the Nexus Dma and Security modules for DVR encryption and decryption. See `NEXUS_PlaypumpSettings.securityDma` and related comments.

Recpump uses RAVE to capture stream data (using the CDB) and index data (using the ITB). The stream data is always MPEG-2 Transport. The index data is Start Code Table (SCT) entries generated by the Start Code Detection (SCD) block. Therefore Recpump has two different but related `GetBuffer`/`ReadComplete` APIs.

NEXUS StcChannel has a mode for DVR TSM, which is different from live TSM.

[7]Also, the live and playback hardware parser bands are different.

High-Level File-Based DVR

Nexus contains the Playback, Record, and File modules that provide a simple and powerful file-based DVR[8]. These modules connect as shown in High-Level File-Based DVR.

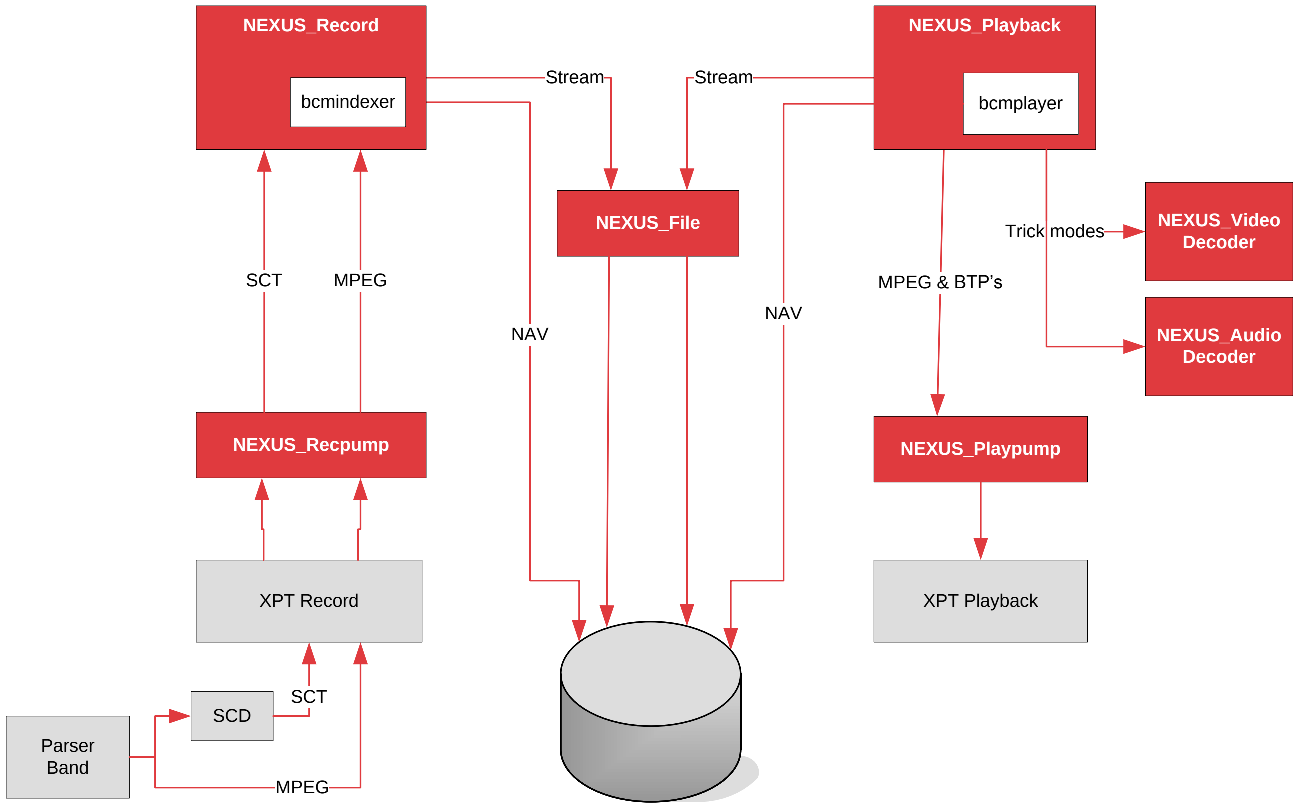


Figure 23: High-Level File-Based DVR

In High-Level File-Based DVR, the gray boxes represent hardware blocks and the red boxes represent software modules.

The Playback, Record, and File modules always run in Linux user mode, even when the rest of Nexus runs in kernel mode. Their calls to Transport, VideoDecoder, or Audio are made using the normal Nexus proxy. This allows File to use normal posix system calls or stdio library calls.

[8]There is a Nexus IP playback application library in nexus/lib/playback_ip that is also built on top of Playpump.

File

The File module supports file read and write access for DVR. The user typically only calls Open and Close calls, then passes the handle to the Playback and Record interfaces.

Threading

The File module creates two internal threads for sending I/O requests to the operating system. These two threads are shared across all playback and record instances. All I/O requests from playback and record are performed asynchronously. A callback notifies the caller when its read or write is complete. The reason for two threads is to ensure that a file I/O request is already posted in the OS when the previous I/O request completes. Additional threads do not add greater efficiency to the system. This centralized file I/O threading allows File to make system-wide priority decisions.

Priority

The File module prioritizes reads and writes across the entire Nexus system. The priority scheme is as follows:

- Record is prioritized above playback.
- Full record buffer is prioritized above an empty record buffer.
- Empty playback buffer is prioritized above a full playback buffer.
- Normal playback is prioritized above trick mode playback.

The goal for this priority scheme is:

- Never overflow a record buffer (which leads to a permanent error in the recorded stream).
- Never underflow a normal playback (where the stuttering would be obvious).
- Do trick modes with whatever I/O bandwidth is left over.

File I/O

For stream data, the File module uses Direct I/O for optimal performance. Direct I/O is supported in Linux for SATA and USB devices. Direct I/O requires that all reads and writes must be 4K page-aligned in memory and on disk. The Playback and Record modules perform this alignment, even during trick modes. The application is required to provide a Direct I/O capable file system (for example, ext2). DVR systems that do not use Direct I/O will usually have performance problems as they scale. If Direct I/O fails, Nexus will automatically convert to standard I/O and will notify the user with an error on the console.

For index data, the File module uses libc-based stdio. This allows for CPU-based buffering, which is more efficient for the low bitrate index data.

The File module uses the `bfile` library located in BSEAV/lib/bfile. The `NEXUS_File` handles and `bfile` handles are related using a simple form of polymorphism.

Playback

The Playback module is a high-level DVR interface for file-based playback. The user gives Playback a single File handle for the stream and index data when it calls `NEXUS_Playback_Start`. This prompts the state machine to query and interact with the Playpump and File interfaces, as shown in Playback State Machine.

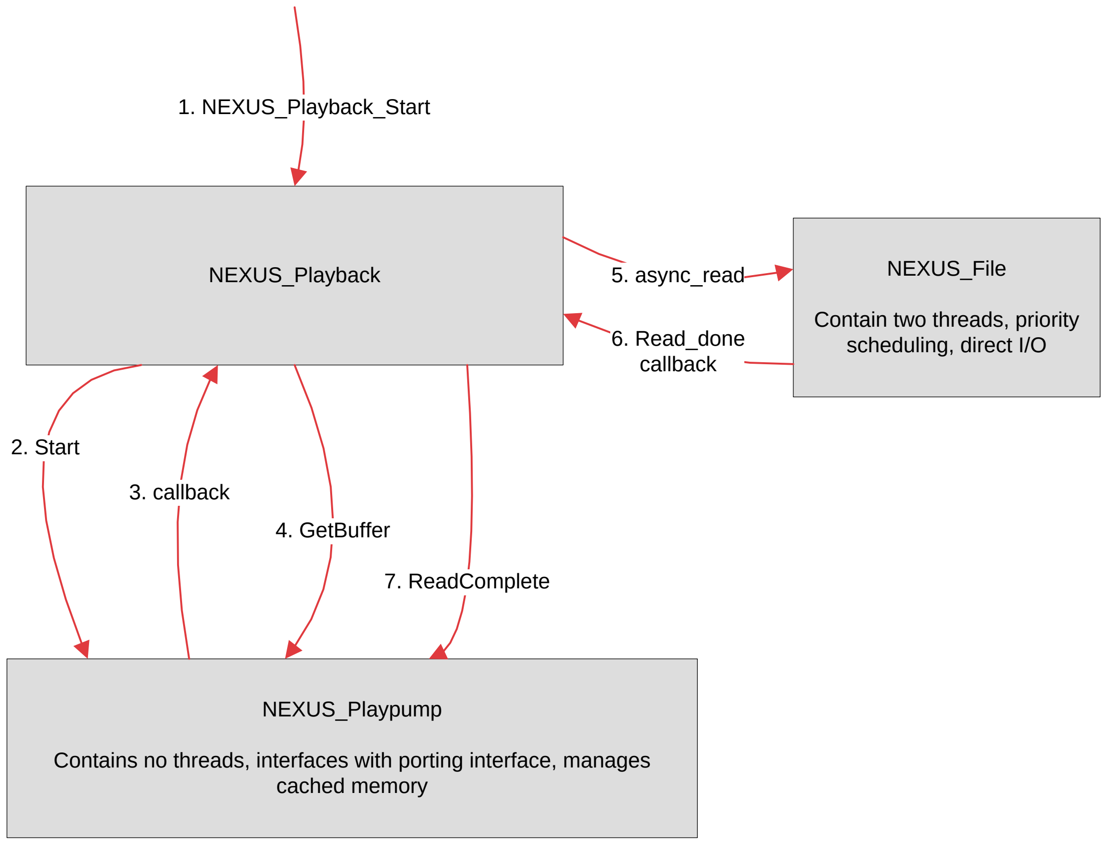


Figure 24: Playback State Machine

The Playback module creates no threads. It uses transport interrupts and the File module's threads to drive the state machine. User calls into the Playback module require synchronizing with its internal state machine, which creates a certain amount of complexity in the code: a threadless Playback implementation is more scalable and efficient because it removes unnecessary context switches.

After Playback has started, the user must start the decoders. Once decode has begun, the user can allow normal playback to continue or can perform trick modes using `NEXUS_Playback_TrickMode`, `NEXUS_Playback_Pause`, and `NEXUS_Playback_FrameAdvance`. Playback Connections with File, Playpump, VideoDecoder, and AudioDecoder shows how Playback interacts with File, Playpump, VideoDecoder, and AudioDecoder Interfaces.

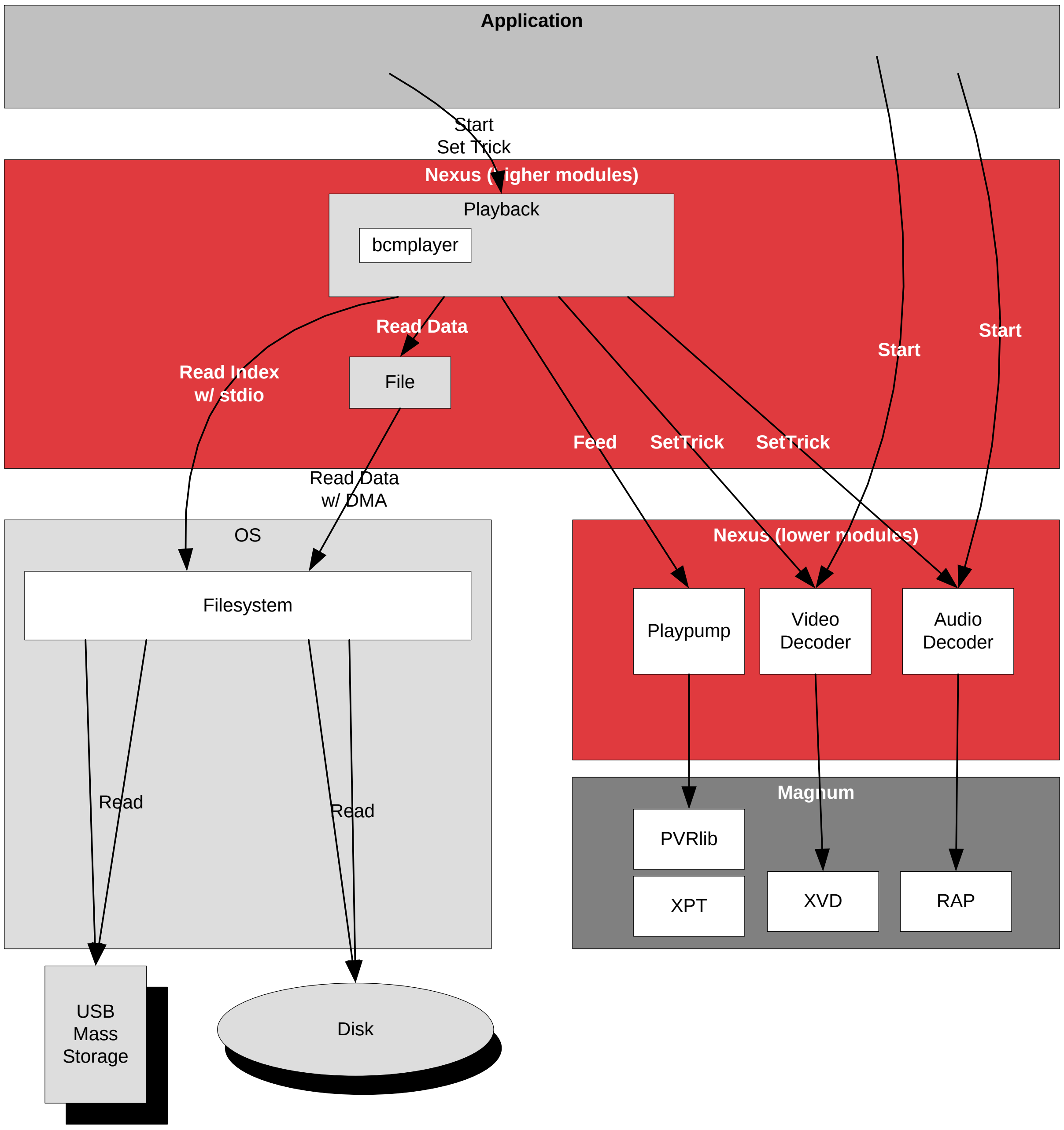


Figure 25: Playback Connections with File, Playpump, VideoDecoder, and AudioDecoder

After giving the respective handles to Playback, the application cannot also call `NEXUS_VideoDecoder_SetTrickState` or `NEXUS_AudioDecoder_SetTrickState` or `NEXUS_Playpump_SetSettings`.

Trick Modes

The Playback Interface provides a set of high- and low-level trick modes. These are performed by a combination of transport, decoder, and display functions.

The Playback API has a simple trick mode interface. The user specifies the rate for the trick mode in units of `NEXUS_NORMAL_PLAY_SPEED`. For instance:

```
NEXUS_PlaybackTrickModeSettings trickModeSettings;

/* 2x fast forward */
trickModeSettings.rate = NEXUS_NORMAL_PLAY_SPEED * 2;

/* 1/4x slow motion */
trickModeSettings.rate = NEXUS_NORMAL_PLAY_SPEED / 4;

/* 10x rewind */
trickModeSettings.rate = NEXUS_NORMAL_PLAY_SPEED * -10;
```

The actual implementation of the trick mode is handled inside Nexus. This can vary per codec. For MPEG, the host is able to drop B and P pictures. For AVC, only I and IDR trick modes may be available and their frequency may vary greatly over the length of the stream.

Internal trick mode techniques involve a combination of the following:

- Host manipulation of the stream (for example, dropping discardable pictures)
- Repeating pictures in the video decoder
- Changing the rate of the STC
- Remapping PTSes in the stream to achieve a desired rate
- (host-paced trick modes)

Playback supports both STC-based trick modes and decoder-based trick modes for pause, frame advance, and slow motion. To use STC trick modes, set `NEXUS_PlaybackSettings.stcTrick = true` before calling `NEXUS_Playback_Start`. STC trick modes are required for 2.0x/0.5x audio trick modes.

Refer to nexus/examples/dvr/playback_trick.c.

Media Framework

Playback and Playpump integrate the Media Framework. The Media Framework allows for detection, navigation, and conversion of a variety of container classes. It is implemented with an extensible plug-in architecture. Container classes currently supported include ASF, AVI, MP4, and MKV. The Media Framework has the following components:

- Media Probe - to detect the format of a file by reading the header
- Media Player - to feed a file to Playback using an index. The media player component implements trick modes.
- Media Filter - to convert container format to MPEG-2 PES format that can be processed by hardware.

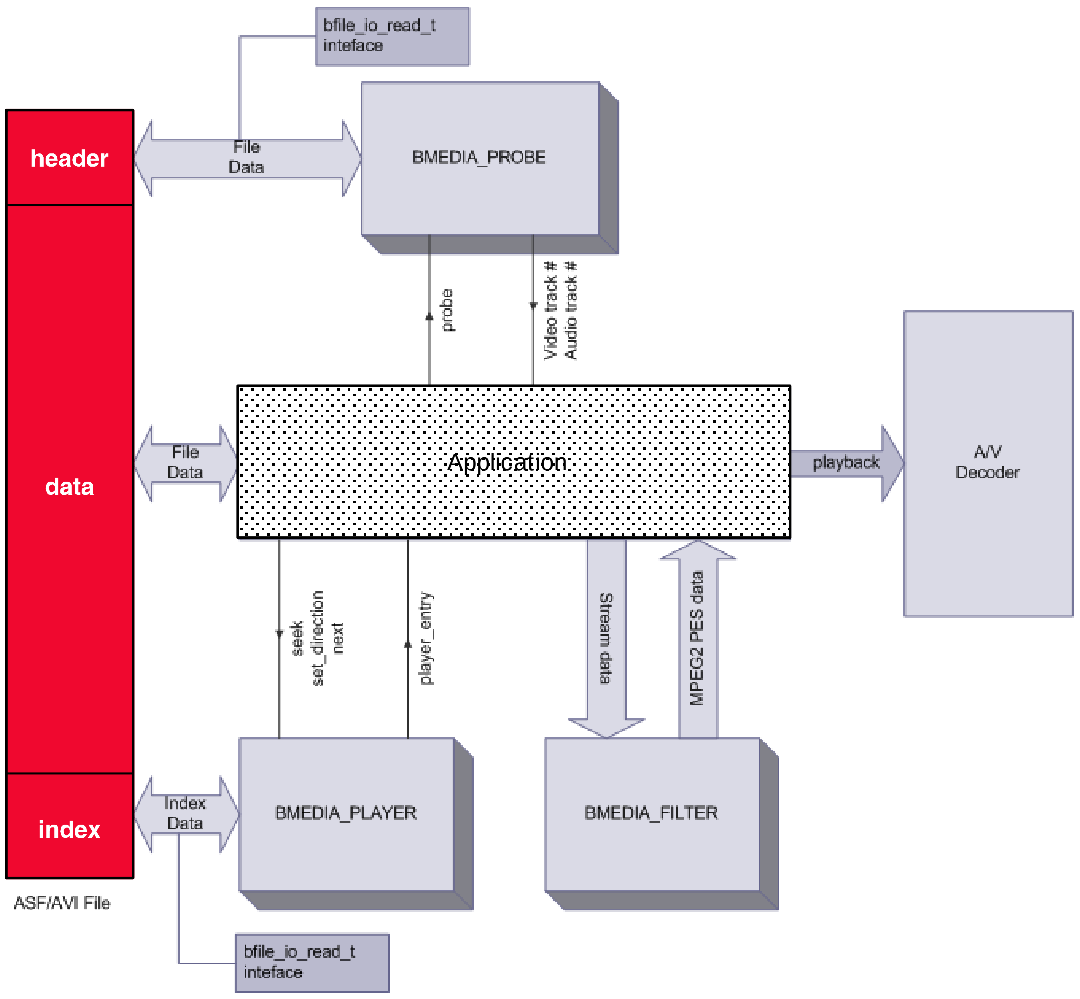


Figure 26: Media Framework Implementation

The Media Filter implementation is done per container/codec pair. The following table is a summarized and partial list of supported pairings. For a full list, please request the latest copy of SupportedMediaContainer.xls from your Broadcom representative.

Container	Video	Audio
AVI	VC1, DivX	MPEG, AC3, WMA
ASF	VC1, DivX, AVC, MPEG2	MPEG, AAC, AC3, WMA
MP4	AVC	AAC, AC3
MKV	VC1 Advanced, AVC, MPEG2	MPEG, AAC, AC3

Playback uses Playpump's scatter-gather API to minimize data copy when converting to MPEG-2 PES.

The media framework fits into the Nexus stack as shown in Media Framework in the Nexus Stack.

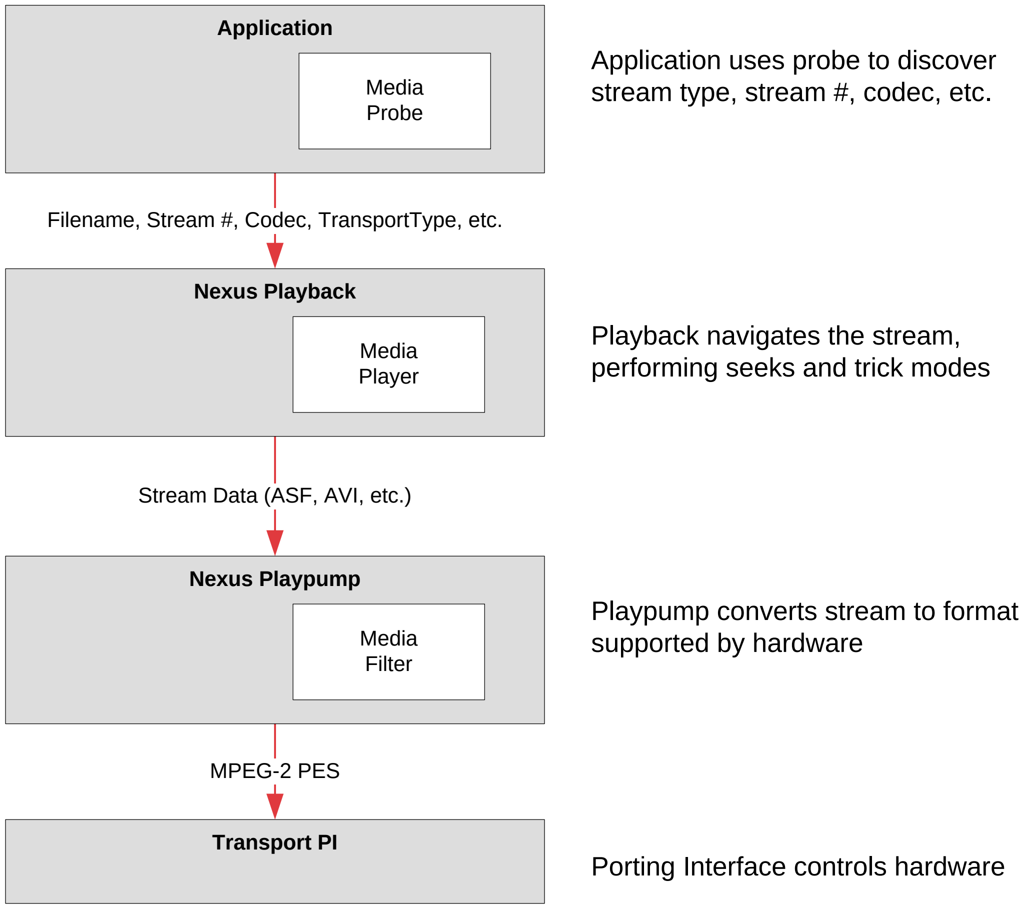
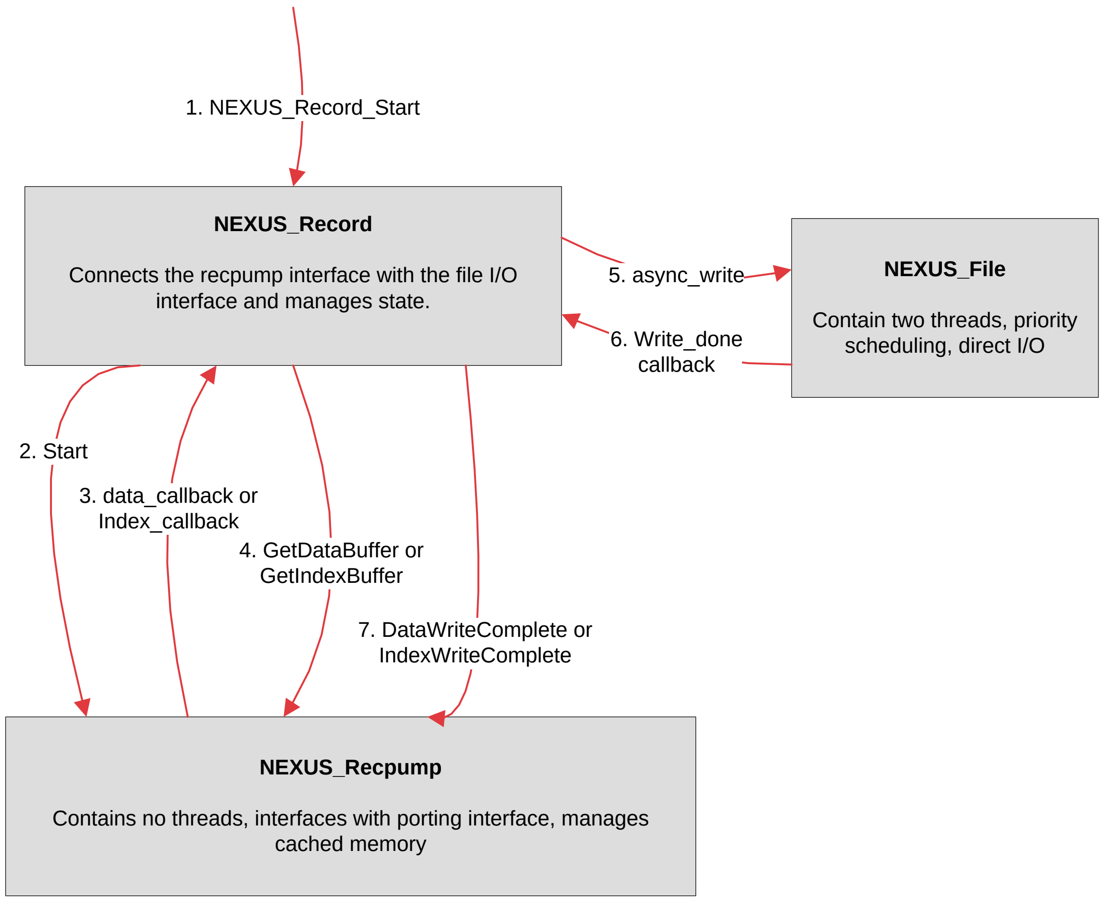

Application
Media Probe
Application uses probe to discover stream type, stream #, codec, etc.
Filename, Stream #, Codec, TransportType, etc.
Nexus Playback
Media Player
Playback navigates the stream, performing seeks and trick modes
Stream Data (ASF, AVI, etc.)
Nexus Playpump
Media Filter
Playpump converts stream to format supported by hardware
MPEG-2 PES
Transport PI
Porting Interface controls hardware

Figure 27: Media Framework in the Nexus Stack

AVI and ASF can be fed directly to Playpump where they are parsed by the media filer. MP4 and MKV require Playback to perform random access and generate a stream that Playpump can convert to PES.

Record

The Record module is a high-level DVR interface for file-based record. The user gives Record a single File handle for the stream and index data when it calls `NEXUS_Record_Start`. This prompts a state machine to query and interact with the Recpump and File interfaces, as shown in Record Module.



1. NEXUS_Record_Start
NEXUS_Record
Connects the recpump interface with the file I/O interface and manages state.
5. async_write
6. Write_done callback
NEXUS_File
Contain two threads, priority scheduling, direct I/O
2. Start
3. data_callback or Index_callback
4. GetDataBuffer or GetIndexBuffer
7. DataWriteComplete or IndexWriteComplete
NEXUS_Recpump
Contains no threads, interfaces with porting interface, manages cached memory

Figure 28: Record Module

The Record module creates no threads. It uses transport interrupts and the File module's threads to drive the state machine. This is essential because extra context switches will not allow a system to scale to three or more high bitrate records.

The Record module captures data from Transport, generates an index, and writes to disk, as shown in Record Module Components.

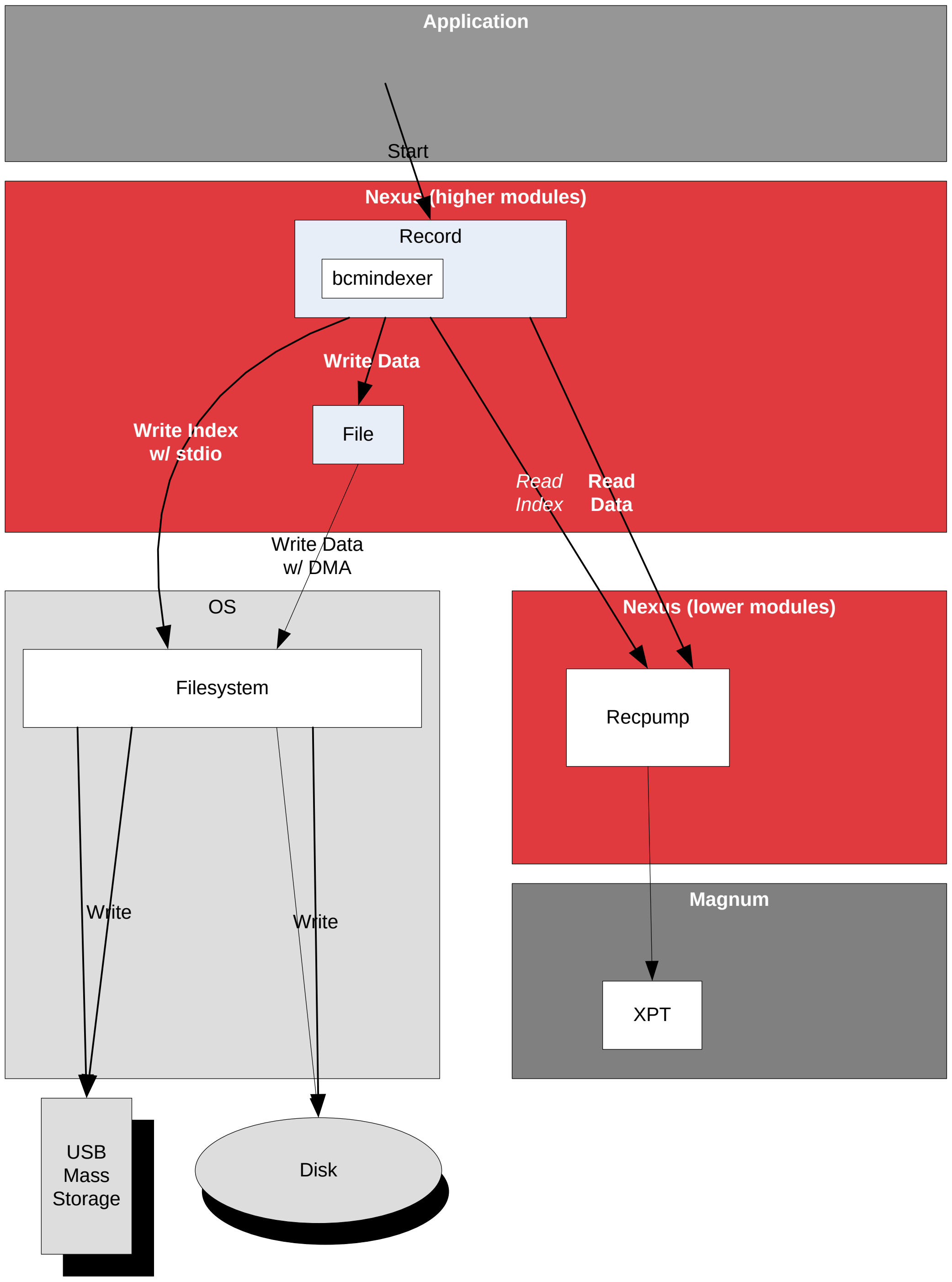


Figure 29: Record Module Components

Indexing

The SCT index generated by hardware indexes start codes and PES headers. It includes the following:

- Picture start codes
- Sequence headers
- PES headers (that provide PTS's)

The Record module uses `bcmindexer` to generate a NAV index based on the SCT index data. The NAV index has the following properties:

- One entry per picture[9].
- A bcmindexer-generated monotonically-increasing 32 bit timestamp (unlike the PTS, which can be discontinuous). This can be used for time-based navigation across any stream with any frame rate. This 32 bit timestamp is subject to overflow for timeshifting. See below for strategies.
- The PTS, sequence header offset, and I-frame offset are stored in every entry for more efficient trick mode processing.
- Bcmindexer has the ability to index MPEG, AVC, and VC1[10] start codes. The NAV format differs depending on the codec. The SCT captures up to 8 bytes of payload in the clear for indexing. AVC requires VLC decode of that payload and we must assume worst case encoding; therefore the number of fields we can parse is limited.

Refer to BSEAV/lib/bcmplayer/utils for offline utilities to print and create NAV indexes.

Overflow

If a stream or index overflow occurs, the user will be notified with a callback. No response is required to this callback; it is only advisory. When an overflow occurs, the RAVE hardware will drop both stream data and index data together. This means that stream data and index data captured after the overflow, even without restarting record, should be valid. In a well-architected system, overflow will never occur. Causes for overflow include:

- Inefficient file I/O, usually because File dropped out of Direct I/O mode
- Improperly blocking code (i.e., high-level code was written with fast code waiting on slow code)
- Some external input overwhelming the system, like a network storm

[9]The field-encoded AVC streams, `bcmindexer` cannot distinguish between field pictures and frame pictures. Bcmplayer includes some support for doing I-frame trick modes with field-encoded AVC stream.

[10]VC1 support has limited testing. Most VC1 streams are contained in ASF containers, not MPEG-2 TS.

You can flush index data while trying to recover from an overflow, but you cannot flush stream data without invalidating the remainder of your index.

Buffer Sizing

The Record module works with the Recpump module to perform Direct I/O. The hardware provides data in 188-byte packets. Direct I/O requires that data be consumed in 4096 byte pages. In order to satisfy both alignment requirements, the CDB must be sized correctly so that a properly aligned wraparound can occur. See nexus_recpump.h for suggestions regarding correct sizing.

Timeshifting

DVR timeshifting records use a continuous recording file (i.e., a circular file or file FIFO). Nontimeshifting records use a linear recording file. The difference is that timeshifting can continue forever because the beginning of the file is trimmed to keep the file at a maximum size.

The File module supports both linear and continuous modes. Refer to nexus_file.h for linear mode and nexus_file_fifo.h for continuous mode. Both APIs produce a `NEXUS_FilePlayHandle` and `NEXUS_FileRecordHandle` that are passed to Playback and Record. Playback and Record are agnostic to the type of File implementation.

For continuous mode, the user can manipulate the record settings to manage the size of the file fifo. The user can also manipulate a virtual beginning and ending of the file at playback time.

When playing back the file FIFO, the application may now have Playback `beginningOfStream` actions when paused or in slow motion. At the end of the file, the application may choose to switch to live decode so that there's no gap[11]. When pausing live TV, the application will actually stop live decode and let the last picture stay on the screen. Nexus does not automatically handle the switch to or from live time and timeshifting playback.

In timeshifting, the Timebase should be locked to the PCR even though the StcChannel is in DVR mode. This prevents drift in the timeshifting buffer.

If the user wants to convert a finite continuous file to an infinite permanent recording, you can increase the bounds to infinity[12]. However, the file format will always remain as a File FIFO.

Refer to nexus/examples/dvr/timeshift.c for an example.

[11]If the app does not switch to live, it will need to leave a gap of about 3–5 seconds from live to avoid playback underflow. Switching to live is the recommended method.

[12] Set NEXUS_FifoRecordSettings.interval = 0xFFFFFFFF, which is 4294967295 seconds or 136 years.

NOTE: The NAV timestamp is 32 bits in units of miliseconds, so it will overflow in about 49 days (2^32/1000/3600/24 ~ 49 days). Switching to 64 bit could not be abstracted from applications, so we require the app to stop the recording before overflow.

If timeshifting, the app can start a second, parallel timeshift buffer then switch over to the second buffer when it is ready. Ideally this can be done when no HDMI output is connected, or in the middle of the night, etc. But if the app gets close enough to overflow, it must simply be done.

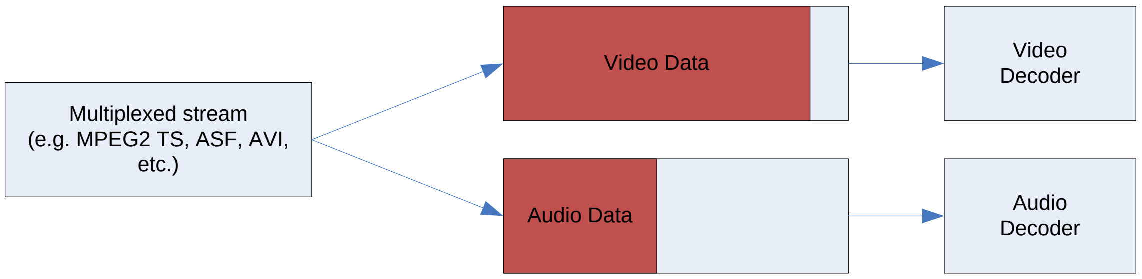
If linear recording, which is very unlikely, it must simply be stopped. Another recording can be started and the app must splice them together during playback.

Audio Decoder Primer

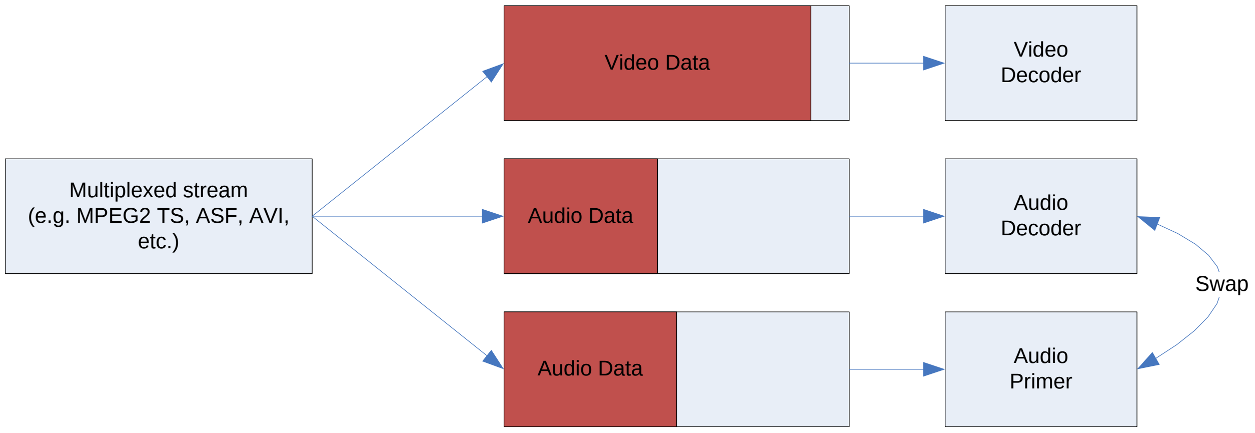
There are two common cases when an application will stop and restart the audio decoder while continuing decode of the video program:

- Changing audio between main and PIP programs
- Changing audio tracks between a primary and secondary audio track in a single program

Because digital data is pipelined, all data in the audio decoder's FIFO is lost when this transition is made. There can be a long delay before audio is heard again. For live decode, the delay is equal to the PCR/PTS difference in the stream. For playback, the delay is equal to the time represented by the data in the video FIFO. For a low bit rate video stream, this could be very large.



If you use an AudioDecoderPrimer on the second track, it will prepare the FIFO so that it is ready for immediate decode. Then you can swap between the two FIFO's between the AudioDecoder and AudioDecoderPrimer with minimal delay. This technique works for all codecs and stream formats.



See nexus/examples/dvr/playback_pip.c for an example.

Linux User and Kernel Modes

Comparison of Modes

Nexus reference platforms support Nexus and Magnum in both Linux user and kernel modes. There are various reasons to prefer one or the other.

Reasons for using user mode include:

- User mode code is easier to develop and debug.
- If applications crash, in user mode the system does not have to be rebooted.
- More debugging tools are available.
- User mode code avoids any GPL considerations related to linking with the Linux kernel.

Reasons for using kernel mode include:

- Kernel mode has smaller worst-case interrupt latencies. The average case interrupt latency in user mode is not much more than kernel mode. The difference (about 1 ms) is well below our hard real time requirements (about 16 ms for video). However, excessive interrupt processing in the kernel can prevent user mode code from being executed. Under heavy Ethernet network traffic or during some filesystem mounts and checks, user mode latencies can be 100 ms or more.

One common approach is to carry out initial development in user mode, then switch to kernel mode for final integration and testing. The proxy layer makes this possible. However, making any large changes can expose latent bugs, so it is highly recommended that the switch between modes is made with ample time to allowed for testing and debugging.

Linux User Mode

By default, Nexus runs in user mode. This means that all of Magnum and Nexus run in user mode and a small driver (bcmdriver.ko) runs in kernel mode in order to handle interrupts.

In user mode, the following files are created:

- libnexus.so—a shared library that contains Magnum and Nexus
- bcmdriver.ko—a kernel driver that services interrupts
- `nexus`—a script that insmods the driver and sets the LD LIBRARY_PATH for loading ./libnexus.so

Linux Kernel Mode With Proxy

Nexus can also be configured to run in kernel mode with an autogenerated proxy. This means that all of Magnum and Nexus run inside the Linux kernel as a driver. This driver is compiled as a Linux module. Nexus will also generate a small user mode `proxy` library that provides an implementation of every Nexus call; however, the implementation of these Nexus calls is simply to call an `ioctl` in the driver.

In kernel mode, the following files are created:

- libnexus.so —a thin proxy that marshals Nexus calls into the kernel mode driver
- nexus.ko—a kernel mode driver that contains Magnum and Nexus
- nexus—a script that insmod's the driver

You can build in kernel mode by defining the environment variable `NEXUS_MODE=proxy`.

To run in kernel mode, use the `nexus` script. It will insmod the appropriate driver.

User mode environment variables cannot be read directly by the kernel mode driver. Instead, Nexus supports the use of the `config` environment variable to pass multiple variables in a space-delimited, semicolon-delimited or colon-delimited list. The `nexus` script accomplishes this using `insmod` parameters.

```
export config="msg_modules=BXVD,BVDC force_vsync=y magic=42"
export config="msg_modules=BXVD,BVDC;force_vsync=y;magic=42"
```

You cannot have both bcmdriver.ko and nexus.ko installed at the same time. The `nexus` script will `rmmod` either driver and `insmod` the required driver for your build.

Because the libnexus.so proxy provides every Nexus call in user space, the same application binary can run with user mode or kernel mode Nexus.

Linux Kernel Mode Without Proxy

Nexus also supports the option of running in kernel mode with no proxy. This means that all of Magnum and Nexus are run inside the Linux kernel as a driver, but no proxy is generated. The user is responsible to write additional driver code to call Nexus.

The advantage of this approach is that the user is in complete control of Nexus and can provide any API to the application (for example, a custom ioctl layer).

You can test kernel mode without a proxy by doing the following:

```
# build
cd nexus/legacy/examples/linuxkernel
make

# run on ARM systems
BOXMODE=`cat /proc/device-tree/bolt/box`
insmod linuxkernel_example.ko config="B_REFSW_BOXMODE=$BOXMODE"

# run on MIPS systems
insmod linuxkernel_example.ko
```

This will perform a simple decode in kernel mode with no proxy. Other Nexus examples will not run in this mode because there is no user mode Nexus API available.

Build System Variables

Refer to the nexus/docs/Nexus_Development.pdf for a description of the nexus build system.

Nexus requires that application's include nexus/platform/$(NEXUS_PLATFORM)/build/platform_app.inc into their Makefile in order to compile and link with Nexus. This Makefile include supports a variety of inputs and produces a variety of outputs. The inputs are either pulled implicitly from the environment or can be explicitly set by the application as Makefile variables. Most of the input variables have defaults. The outputs are Makefile variables which are set and have required or optional usage.

Inputs

The following lists the most common Nexus build options (i.e., inputs) available[13].

Build Option	Description	Default
NEXUS_PLATFORM	Selects which directory in nexus/platforms to build.	No default; must be set.
BCHP_VER	Chip revision to compile for (e.g., A0, B0, B1, etc.)	A0
B_REFSW_ARCH	Specifies the type and mode of the CPU. Value s include: arm-linux, mipsel-linux or mips-linux (big endian)	arm-linux or mipsel-linux
B_REFSW_OS	Specifies the OS being compiled for. Supported values are: linuxuser, linuxkernel, ucos, wince, vxworks Setting NEXUS_MODE=proxy or driver will default B_REFSW_OS=linuxkernel as needed.	linuxuser
LINUX	Linux only. Selects the Linux kernel source directory Only applies of B_REFSW_OS=linuxuser or linuxkernel	/opt/brcm/linux
LINUX_INC	Linux only. Select the include directory for preconfigured Linux source.	$(LINUX)/include

[13]This table does not list Atlas or Trellis build variables.

NEXUS_BIN_DIR	Set the location for Nexus binaries	obj.97xxx/nexus/bin
NEXUS_HEADERS	Run "make nexus_headers" as part of the build. See section below for details.	
NEXUS_MODE	Set to **proxy** to build Linux kernel mode with a proxy. Set to **driver** to build Linux kernel mode without a proxy. Set to **client** to build Linux usermode multiprocess client. Leave unset to build Linux user mode.	
B_REFSW_DEBUG	Set to **n** to build in release mode. Also support **minimal** and **no_error_messages**.	y
B_REFSW_DEBUG_LEVEL	If B_REFSW_DEBUG=y, then set this to **err** for only BDBG_ERR messages, or set to wrn for only BDBG_ERR and _WRN messages.	
B_REFSW_VERBOSE	Set to **y** to see verbose compilation output.	
NEXUS_POWER_MANAGEMENT	Set to **n** to disable dynamic power management in Nexus and Magnum	y
MAGNUM_SYSTEM_DEFINES	Add compilation options for Magnum builds. Useful for debug.	
B_REFSW_SHAREABLE	Set to **n** to build nexus as a static libraries. The default of "y" builds nexus as a shared library.	y
NEXUS_EXTRA_CFLAGS	Extra CFLAGS for Nexus builds	
NEXUS_EXTRA_LDFLAGS	Extra LDFLAGS for Nexus builds	

The Nexus build system includes backward-compatibility support for the variables `DEBUG`, `ARCH`, `OS`, `PLATFORM`, `MODE`, `VERBOSE`, and `SHAREABLE`. If your Makefile defines one of these, you will need to explicitly define the `B_REFSW_` or `NEXUS_` variant. For example, if your Makefile defines DEBUG to something other than y/n, then you must explicitly define `B_REFSW_DEBUG=y/n` to prevent Nexus from picking up the incompatible DEBUG default.

Outputs

Build Output	Description	Typical Usage
CC, LD, CXX, AS, AR, NM, STRIP, OBJCOPY, OBJDUMP, RANLIB, MKDIR, PWD, MV, PERL, CPP, CP, RM, SORT, SED, TOUCH, AWK	Standard toolchain variables. Set based on B_REFSW_ARCH and B_REFSW_OS setting. See nexus/build/os/${B_REFSW_OS}/os_tools.inc for a complete listing.	Use them to build your application. app: app.c $(CC) -o $@ $< $(CFLAGS)
NEXUS_APP_DEFINES	Defines needed to compile application code in a compatible way with Nexus	CFLAGS += $(addprefix -D,$(NEXUS_APP_DEFINES))
NEXUS_APP_INCLUDE_PATHS	Include paths needed for Nexus public API header files	CFLAGS += $(addprefix -I,$(NEXUS_APP_INCLUDE_PATHS))
NEXUS_CFLAGS	CFLAGS requires to compile code that will link correctly with Nexus	CFLAGS += $(NEXUS_CFLAGS)
NEXUS_LDFLAGS	Linker flags needed to link with Nexus	LDFLAGS += $(NEXUS_LDFLAGS)
NEXUS_ENDIAN	Set to either BSTD_ENDIAN_BIG or BSTD_ENDIAN_LITTLE. This is determined by B_REFSW_ARCH.	No use required. The require magnum –D is already added to NEXUS_CFLAGS.
NEXUS_LD_LIBRARIES	Linker flags for Nexus single-process and server apps	LDFLAGS += $(NEXUS_LDFLAGS) $(NEXUS_LD_LIBRARIES)
NEXUS_CLIENT_LD_LIBRARIES	Linker flags for Nexus client apps	LDFLAGS += $(NEXUS_LDFLAGS) $(NEXUS_CLIENT_ LD_LIBRARIES)

Another form of output is the defaults assigned to unset input variables. For instance, if your application does not define B_REFSW_OS, Nexus will set `B_REFSW_OS=linuxuser`. Your application can use this setting as an output.

make nexus_headers

If you run “cd nexus/build;make nexus_headers”, or if you build with export NEXUS_HEADERS=y, then nexus will copy all public API header files to obj.97xxx/nexus/bin/include. It will also generate a static platform_app.inc. You can bundle the Nexus binaries along with these header files and the Makefile .inc to make a binary package for Nexus.

Nexus will also aggregate all of the public API into a single nexus.h, then run that through the C preprocessor using the platform_app.inc CFLAGS, storing the output as obj.97xxx/nexus/bin/include/nexus.cpp.h. You can compare one platform’s nexus.cpp.h with another platform’s to see if the Nexus API’s are binary compatible.

Graphics

This topic is addressed in detail in nexus/docs/Nexus_Graphics.pdf.

Memory Management

This topic is addressed in detail in nexus/docs/Nexus_Memory.pdf.

Power Management

This topic is addressed in detail in nexus/docs/RefswPowerManagementOverview.pdf.