NxClient

Broadcom
Corporate Headquarters: San Jose, CA
Web: www.broadcom.com

Revision History

Revision	Date	Change Description
0.1	12/3/12	Initial draft
0.2	2/25/13	Rename refsw_server to nxserver
0.3	3/12/13	Build libnxclient.so
0.4	11/25/13	Revise local nxclient
1.0	1/15/14	Revise document to describe use cases
1.1	4/11/14	Add "External App" section Revise "HDMI Input" section Add "NxServer and libnxserver.a" section Add "HDMI output" section Add "Stereoscopic TV (3DTV)" section
1.2	5/20/14	Fully-mapped client heap is now optional Add "4K and 10 bit decode and display" section Describe serial STC management
1.3	8/27/14	Change required relationship between NxClient_Alloc and Connect
1.4	10/10/14	Add section on DMA
1.5	1/20/15	Add sections on PQ, FCC, audio decoder primer
1.7	4/21/15	Clarify "Compiling NxClient into your application" section
1.9	9/1/15	NXCLIENT_SUPPORT build and compile options
2.0	11/9/15	Expand intro. Use ". nxclient.sh" for setting client runtime environment.
2.1	1/5/16	Expand "External App" section
2.2	3/30/16	Add HDCP state machine in HDMI output section
2.3	6/21/16	Revise "Timebases and Serial STC's" section
2.4	2/14/17	Update 3D graphics, Picture decode, Audio/video decode
2.5	5/5/17	Removed two sections

Table of Contents

Introduction

NxClient is a client library and resource-managing server which runs above Nexus to facilitate multi-process applications. This document shows how to use NxClient along with regular Nexus API's to design a multi-process system.

After calling NxClient to obtain resources, much of the work in the application is still calling regular Nexus API's. When porting to NxClient, most application code on the front half of the system (tuners, demods, parser bands, playbacks, records, graphics rendering, etc.) remains unchanged. The application code on the back half of the system (video and audio decoders, display, audio outputs) must change in order to share those resources between multiple clients.

NxClient includes a re-usable server (called nxserver) which does the resource management of these back half resources. It supports a small set of NxClient API's to configure share resources, like changing display format, adding audio post processing, or adjusting picture quality.

NxClient relies on the SimpleDecoder and SurfaceCompositor modules which are part of Nexus. The application calls the client interfaces of SimpleDecoder and SurfaceCompositor while nxserver calls server-side interfaces. The application specifies the input pids and codecs and starts decode, while the server configures the outputs in response to those inputs. This division of labor makes writing client applications with these shared resources simpler.

Directory structure

NxClient is stored in the Nexus directory tree under nexus/nxclient. Its sub-directories are:

Index	Description
nexus/nxclient/include	NxClient API and client-side implementation
nexus/nxclient/build	Makefile for building libnxclient.so
nexus/nxclient/server	NxServer application
nexus/nxclient/examples	Simple example applications
nexus/nxclient/apps	More complex utilities and integrated applications
nexus/nxclient/apps/utils	Helper code for the client applications
nexus/nxclient/apps/resources	Images, fonts and config files for the client applications
nexus/nxclient/docs	Documentation

Example applications

Build the server and client apps as follows:

```
cd nexus/nxclient
make
```

NxClient builds for user mode unless export NEXUS_MODE=proxy directs it to kernel mode. Binaries are copied to nexus/bin or NEXUS_BIN_DIR if defined.

Run the server and client applications as follows:

```
nexus nxserver &

. nxclient.sh
play FILENAME
blit_client
playpcm < PCMFILENAME
setaudio –mute on
transcode INPUTFILE
```

Most utility apps in nexus/nxclient/apps support "--help" for command line options.

"make clean" will clean both NxClient and Nexus. If you want to clean only NxClient, run "make clean_apps". Building with "export PLAYBACK_IP_SUPPORT=y" is required to compile in playback_ip to some client apps.

See nexus/nxclient/README.txt for another form of this "quick start" documentation.

Compiling NxClient into your application

In addition to compiling for Nexus, there are a few extra lines to add NxClient:

```
NEXUS_TOP=<location of "nexus" folder>

include $(NEXUS_TOP)/platforms/$(NEXUS_PLATFORM)/platform_app.inc
CFLAGS += $(NEXUS_CFLAGS)
CFLAGS += $(addprefix -I,$(NEXUS_APP_INCLUDE_PATHS))
CFLAGS += $(addprefix -D,$(NEXUS_APP_DEFINES))

include $(NEXUS_TOP)/nxclient/include/nxclient.inc
CFLAGS += $(NXCLIENT_CFLAGS)
LDFLAGS += $(NXCLIENT_LDFLAGS)

nxclient:
    make -C $(NEXUS_TOP)/nxclient/build

app: nxclient
    $(CC) -o $@ app.c $(CFLAGS) $(LDFLAGS)
```

NOTE: NXCLIENT_LDFLAGS already includes NEXUS_CLIENT_LD_LIBRARIES, but NXCLIENT_CFLAGS does not include NEXUS_CFLAGS, NEXUS_APP_INCLUDE_PATHS and NEXUS_APP_DEFINES. You must explicitly include the latter.

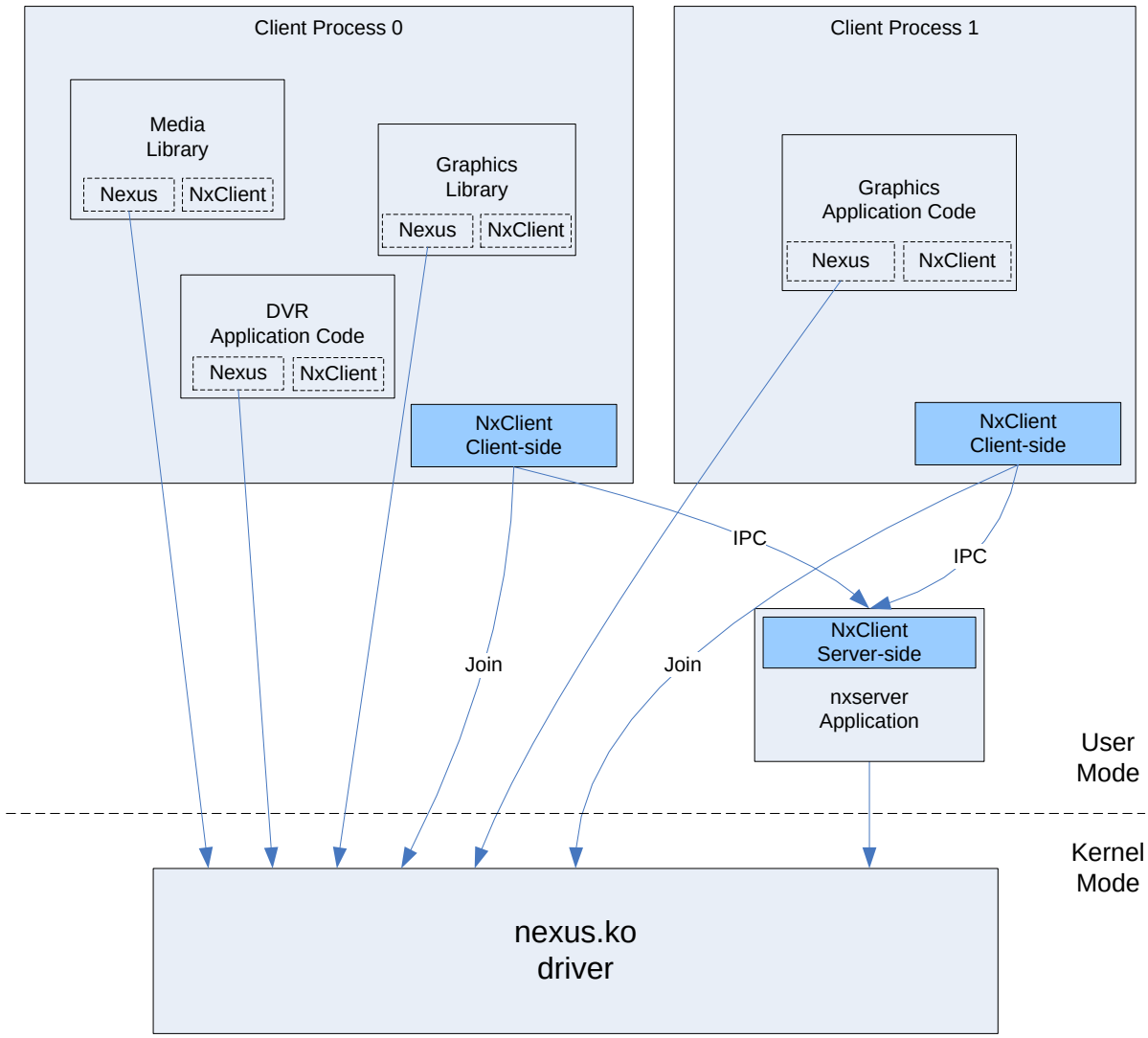
Running "make" in nxclient/build causes libnxclient.so to be built. The shared library is placed in nexus/bin (or wherever NEXUS_BIN_DIR points).

You can also follow nexus/nxclient/examples/Makefile as a working example.

If you want your application to build with and without NxClient support, use "export NXCLIENT_SUPPORT=y" as a standard build option and "#if NXCLIENT_SUPPORT" for conditional compilation.

System Diagram

A typical NxClient system looks like the following:



Each process has two connections: an NxClient "IPC" connection to the nxserver application and a direct "Join" connection to the Nexus driver. Within each process, there can be multiple blocks of library code that access NxClient and Nexus. NxClient allows the code to dynamically register itself and acquire resources. The server application receives those requests and grants the resources. Once the resources are granted, the code can call Nexus directly and use those resources.

NxServer implements the NxClient API. It owns the display and the regular video and audio decoders. It creates and configures resources on behalf of the clients. In order to make the client API's simple, the server application is necessarily complex. However, it is written in a generic way that is intended to be used in a variety of systems.

A client must call NxClient_Join first. This initializes the nxclient library, connects to NxServer and allows connects to Nexus. A client disconnects by calling NxClient_Uninit. NxClient_Join can be called multiple times from the same process, but subsequent calls are only reference counted. There is only one nxclient connection per process. This reference counting allows libraries to be written independently of each other. Each library can join and do work independent of other libraries.

However, NxClient does not require IPC. You can link a client directly with nxserver as a library. See nexus/nxclient/apps/Makefile for an example. If you build with NXCLIENT_LOCAL=y, the client app will link with the server. Only one client can run as a local client, but the IPC mechanism will still be available for other clients.

Client heaps

The purpose of a multi-process system is increased stability through process isolation. Process isolation must include memory isolation. One client should not be able to scribble on the server's memory and bring down the system.

However, embedded systems with real-time memory scheduling constraints also have very complex memory configurations. It is difficult to reconfigure the Nexus memory layout without having to understand a wide range of internal issues.

NxClient strikes a balance between complete process isolation and no process isolation for heap management. When NxServer starts, it passes a subset of heaps to the client. The default client heap is the "graphics heap". Optionally, the heap can receive a second "fully mapped" heap, but the default is for the client to share the main fully mapped heap with the server. Clients also get access to the DRM secure heap and a secondary graphics heap if it exists. The list of heaps is the following:

Client heap[] index	Typical Size	NEXUS_MemoryType (memory mapping)	Use Case
0	Large (~256MB)	eApplication	VC4/VC5/M2MC graphics, record
1	Medium (~100MB)	eFull	Clients need a fully mapped heap for packet blit, playback, DRM, and audio PCM playback
2	Varies	eSecure	DRM
3	Large	eApplication	Secondary graphics heap for M2MC, but not VC4/VC5
4	Varies	On-demand	Dynamic heap. Must use NxClient_GrowHeap to expand; heap will shrink when client exits.
5	Varies	None	SVP XRR heap
6	Varies	None	SVP secure graphics heap

All clients work in these heaps. Clients have no access to other heaps. So, a client can scribble over the heap memory of another client, but not the server.

The client heap numbering of 0 and 1 is not the same as the server numbering. The server heap numbering varies considerably. A client can query its heaps using NEXUS_Platform_GetClientConfiguration, then calling NEXUS_Heap_GetStatus on the handles returned by NEXUS_ClientConfiguration.heap[].

However, the client should not have to be aware of this numbering. By default, Nexus interfaces will default NEXUS_HeapHandle parameters to NULL. When the Nexus module sees

the NULL, it will request a default heap for that client. This will be a heap to which the client has memory mapped access. If the heap must have NEXUS_MemoryType_eFull mapping (for example, for packet blit or playpump) the Nexus module will request that.

When starting nxserver, you can make adjustments to the default heap sizes. Picture buffer heaps are adjusted using Nexus Memconfig. Some nxserver command line options (like -memconfig) will modify those memconfig settings. If you want full control of the memconfig API, you should write a replacement for the default nxserver.c wrapper of nxserverlib. You can also make manual adjustments to other heaps using the "-heap" option.

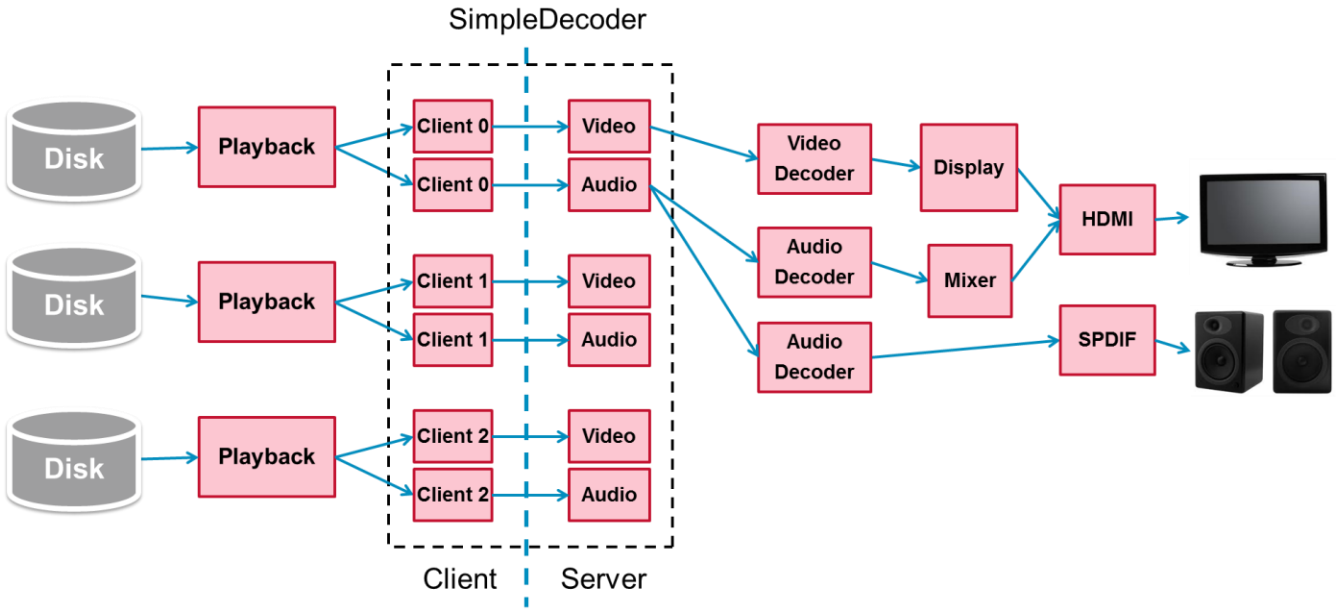
Use Cases

This section presents a set of typical scenarios and how to use Nexus and NxClient to write applications. This document tries to avoid duplicating application logic, so the comments are minimal – just enough to explain what the example apps are doing. You should run and study the example apps to get a complete picture of how to use each feature.

Video and audio decode

See nexus/nxclient/examples/playback and tune_qam.c for simple decode applications. See nexus/nxclient/apps/play.c and live.c for more complex apps.

The SimpleVideoDecoder and SimpleAudioDecoder interfaces provide a higher-level client/server interface for decode. The same interfaces are used for live and playback decode. They allow multiple clients to share resources as follows:



The typical series of NxClient calls are:

- NxClient_Alloc - have the server allocate simple video/audio decoder handles for the client. These handles will serve as containers for the regular VideoDecoder and AudioDecoder resources.

- NEXUS_SimpleVideoDecoder_Acquire and NEXUS_SimpleAudioDecoder_Acquire – acquire the handle from Nexus using the ID provided by NxClient. This handle is owned by the client and will never be removed.

- NxClient_Connect – ask NxClient to connect the client handle to the underlying decoder resource. This may involve stealing the regular decoder from another client. The current policy is "last one wins".

- NEXUS_SimpleStcChannel_Create – Creates a handle used to link video and audio decoders for lipsync. The regular StcChannel interface and its associated stcIndex and timebase properties will be created and configured on the server based on system requirements.

- NEXUS_SimpleVideoDecoder_SetStcChannel and NEXUS_SimpleAudioDecoder_SetStcChannel – Connect the decoders together for lipsync. This must be done before starting either decoder. This allows higher-level lipsync operations like Astm and SyncChannel to have all information before any one is started.

- NEXUS_SimpleVideoDecoder_Start and NEXUS_SimpleAudioDecoder_Start – specify the PID and codec and start. The underlying decoders and backend filter graphs will be automatically configured based on the codec and server-side configuration.

The simple decoder interfaces keep the client simple by moving configuration logic into the server. It allows multiple client apps to share decoders with a pre-emptive resource allocation scheme. Each client has SimpleVideoDecoder and SimpleAudioDecoder handles which are exclusive owned, so even if underlying resources are grabbed, the client application will not crash.

Audio and video decoders which will be lipsynced using SimpleStcChannel should be connected using the same NxClient_Connect call for minimal resources. See *Appendix C* for more detailed requirements.

Dual video decode (typically main and PIP) must come from separate NxClient_Alloc and NxClient_Connect calls. NxClient_ConnectSettings has a parameter to request NxClient_VideoWindowType_ePip, otherwise the default is the main video window.

Mosaic video decode requires a single NxClient_Alloc and NxClient_Connect call for all mosaic decoders which are in the group (and no other decodes apart from that set of mosaics should be added into the NxClient calls). After the SimpleVideoDecoder handles have been acquired and connected as a group, they are started and managed independently. They can have separate sources and trick modes. However they cannot use STC trick modes because they share a single HW serial STC. They can be sized and overlapped independently, but their total size cannot exceed the area of one regular decode window.

Resizing graphics and video windows

NxClient assumes that a client does not have direct access to the position of its graphics on the screen and that client video is always a child of its graphics. This is necessary for several reasons:

- The video window is always placed within the context of a GUI. If the GUI is resized, moved or hidden, video should follow.
- Nexus is configured to always place graphics above video. This allows alpha-blended graphics to be overlaid on video. If a client wants to place its video above another client's graphics, this can only be done by having a surface with a transparent alpha plane punch a hole for video. That is, graphics is almost always required on top of video, even if the only purpose of that graphics is to set alpha to 0.
- Nexus SurfaceCompositor was designed so that the server can reposition a client without the client's permission or knowledge. This allows Nexus to take an application which thinks it is full screen and run it as one client among many. This can only be done by preventing the client from positioning itself on the screen.

To resize graphics, the client cannot work directly with SurfaceCompositor. It must ask NxClient to reposition the surface. This is done with NxClient_SetSurfaceClientComposition. See nexus/nxclient/apps/blit_client and its "-rect" parameter for an example.

NxClient_Config_SetSurfaceClientComposition can used to reposition another client's graphics. See nexus/nxclient/apps/config and its "-rect" and "-vrect" parameters for an example. Be aware that a client may open any number of SurfaceClients, so there's no automatic way for a third party to know which SurfaceClient should be repositioned without having additional information.

However, a client can resize its video window directly. This is necessary because the client renders its own GUI and will need to position the video in that GUI. The server can't know where the video should be positioned. This is done as follows:

- Call NxClient_Alloc and NEXUS_SurfaceClient_Acquire to get a top-level graphics window.
- Call NEXUS_SurfaceClient_AcquireVideoWindow to create a child SurfaceClient for the video.

- Call NxClient_Connect for the video decoder and specify NxClient_ConnectSettings.simpleVideoDecoder[0].surfaceClientId for the top-level graphics window and set .windowId to the ID used in AcquireVideoWindow.
- Call NEXUS_SurfaceClient_SetSettings to reposition the video window.

If you have two or more video windows (main and PIP, or mosaic), each window needs its own child SurfaceClient handle. They are controlled separately. The window ID's given to SurfaceCompositor and NxClient can be anything the client chooses. They are relative to the top-level graphics SurfaceClient.

See nexus/nxclient/apps/play.c and the "-pig" option for an example of resizing video within graphics.

See nexus/docs/Nexus_SurfaceCompositor.pdf for more details on virtual display coordinates and video window control.

4K and 10 bit decode and display

NxClient applications can configure 4K and 10 bit display using NxClient_DisplaySettings. Set NxClient_DisplaySettings.videoFormat to one of the 4K enums. 10 bit, including 4:2:0 color space control is set with NxClient_DisplaySettings.hdmiPreferences.colorDepth and .colorSpace. However, Nexus will automatically select the highest quality possible color depth and color space.

See how nexus/nxclient/apps/setdisplay handles "-display_format 3840x2160p60" for an example.

NxClient applications can decode 4K and 10 bit sources using SimpleVideoDecoder. The app should request a 4K/10-bit capable decoder using NxClient_ConnectSettings.simpleVideoDecoder[].decoderCapabilities.maxWidth, .maxHeight, and .colorDepth. When starting decode, set NEXUS_SimpleVideoDecoderStartSettings.maxWidth and .maxHeight. Also, set NEXUS_VideoDecoderSettings.colorDepth before doing the connect. See nexux/nxclient/apps/utils/media_player.c for sample code. nexux/nxclient/apps/play.c uses media probe to automatically handle 4K and 10 bit. No command line options are required.

Frontend

See nexus/nxclient/examples/tune_qam.c for a simple application with hardcoded values. See nexus/nxclient/apps/live.c for a more complete, more complex application that includes PSI scan, multi-demod, and fast channel change.

First, the application gets a frontend handle by using NEXUS_Frontend_Acquire. This is a dynamic allocator that works by capabilities or by platform index. Once the frontend handle is acquired, it is exclusively owned by the client.

Then the application allocates a parser band using NEXUS_ParserBand_Open(NEXUS_ANY_ID). Because of transport bandwidth requirements, one parser band should be used for each decode or fast channel change primer. A single parser band can be shared between decode and record.

Finally, the application decodes audio and video from that parser band by opening pid channels and using NEXUS_SimpleVideoDecoder and NEXUS_SimpleAudioDecoder.

Multi-process support for pid channels and message filtering are the same as single-process.

DVR playback

See nexus/nxclient/examples/playback.c for simple decode applications. See nexus/nxclient/apps/play.c for more complex apps.

DVR playback uses SimpleVideoDecoder and SimpleAudioDecoder just like live decode. The only difference is that the pid channels come from playback, not a parser band.

Multi-process support for playback, file module, media framework and playpump are the same as single-process.

Use "-crypto" to decrypt using CPD. The default keys for play match the default keys for record.

IP client playback

See nexus/nxclient/apps/play.c for an integration of IP playback with NxClient. It will do HTTP streaming if given a URL instead of a file name.

The media_player.c library is a helper library and is subject to API change. It shows how a client can use the playback IP library instead of the NEXUS_Playback module to do playback.

The use of NxClient and Nexus multi-process is the same.

DVR record

Multi-process record is the same as single-process record. After the client has acquired and tuned a frontend handle, it configures the ParserBand, Recpump and Record in the same way.

Use nexus/nxclients/apps/record to record from frontend, streamer or from playback.

Use "-crypto" to encrypt using CPS. The default keys for record match the default keys for play.

Timebases and Serial STC's

NxClient automatically selects and configures timebases and serial STC's in the system. Clients can do manual timebase configuration using NEXUS_Timebase_Open(NEXUS_ANY_ID) and

assigning to NEXUS_SimpleStcChannelSettings.timebase. Manual configuration is useful to do clock skewing for IP streaming or to do clock recovery from a timeshifting record.

NxClient always uses timebase 0 for display and audio outputs. Any live decode on the main video window with low jitter will do PCR clock recovery using timebase 0, which will drive the display and audio output clock. If the live decode is for PIP or if it's reported as high jitter, another timebase will be allocated. Any playback decode will use timebase 0 in its current state; it will not reconfigure timebase 0. Any encode will use a dynamically allocated timebase.

Nexus also allocates serial STC's for decode and encode. The serial STC is represented by NEXUS_StcChannelSettings.stcIndex in the Nexus API. The client should not call NEXUS_StcChannel_Open because there is no way to dynamically allocate a serial STC and avoid a collision with NxClient. Instead, the client calls NEXUS_SimpleStcChannel_Create and connects lipsync'd decoders together. In the server, an StcChannel is allocated and a serial STC is selected.

2D graphics

See nexus/nxclient/examples/client.c for a simple graphics example. See nexus/nxclient/apps/blit_client.c, animation_client.c and tunneled_client.c for more complex examples.

Because Graphics2D is virtualized, each client should open their own Graphics2D handle. Surface heap memory will be allocated by default from client heap[0], which can be very large because it has NEXUS_MemoryType_eApplication mapping.

After graphics have been rendered locally in the client using a combination of CPU and Grahpics2D writes, the final surface can be submitted to SurfaceCompositor for display. SurfaceCompositor is used as follows:

- NxClient_Alloc - have the server allocate one or more SurfaceClient handles for the client.
- NEXUS_SurfaceClient_Acquire – acquire the handle from Nexus using the ID provided by NxClient.
- NEXUS_SurfaceClient_SetSurface or PushSurface – Submit graphics to SurfaceCompositor to be displayed.

A client app is not limited to one SurfaceClient. A complex app could have multiple windows on the screen.

See nexus/docs/Nexus_SurfaceCompositor.pdf for additional information.

3D graphics

See BSEAV/lib/gpu/applications/nexus/cube for a simple 3D graphics example. If the OpenGL-ES library is compiled with "export NXCLIENT_CLIENT=y" it will use NxClient and submit surfaces using NEXUS_SurfaceClient_PushSurface. If not, OpenGL-ES is single process and will submit surfaces to the graphics framebuffer directly.

See nexus/docs/OpenGL_ES_QuickStart.pdf for additional information.

User input

See nexus/nxclient/apps/desktop.c for a simple example.

By default NxServer opens all user input devices including IR input, Keypad as well as Linux devices like USB keyboard and mouse. NxServer will post all incoming events to InputRouter. Clients can receive those events using InputClient as follows:

- NxClient_Alloc - have the server allocate an InputClient handle for the client.
- NEXUS_InputClient_Acquire – acquire the handle from Nexus using the ID provided by NxClient.
- NEXUS_InputClient_SetSettings – set any client-side mask for input events. The simplest mask is 0xFFFFFFFF for all events and 0x0 for no events.
- NEXUS_InputClient_GetCodes – get input events

See nexus/nxclient/apps/utils/remote_key.c for a helper library which converts a set of input devices into a set of generic codes. This makes writing general purpose example apps easier.

The performance of NxClient IPC calls may not be sufficient for mouse events. A client may want to go directly to Linux for mouse support. NxClient doesn't add much value.

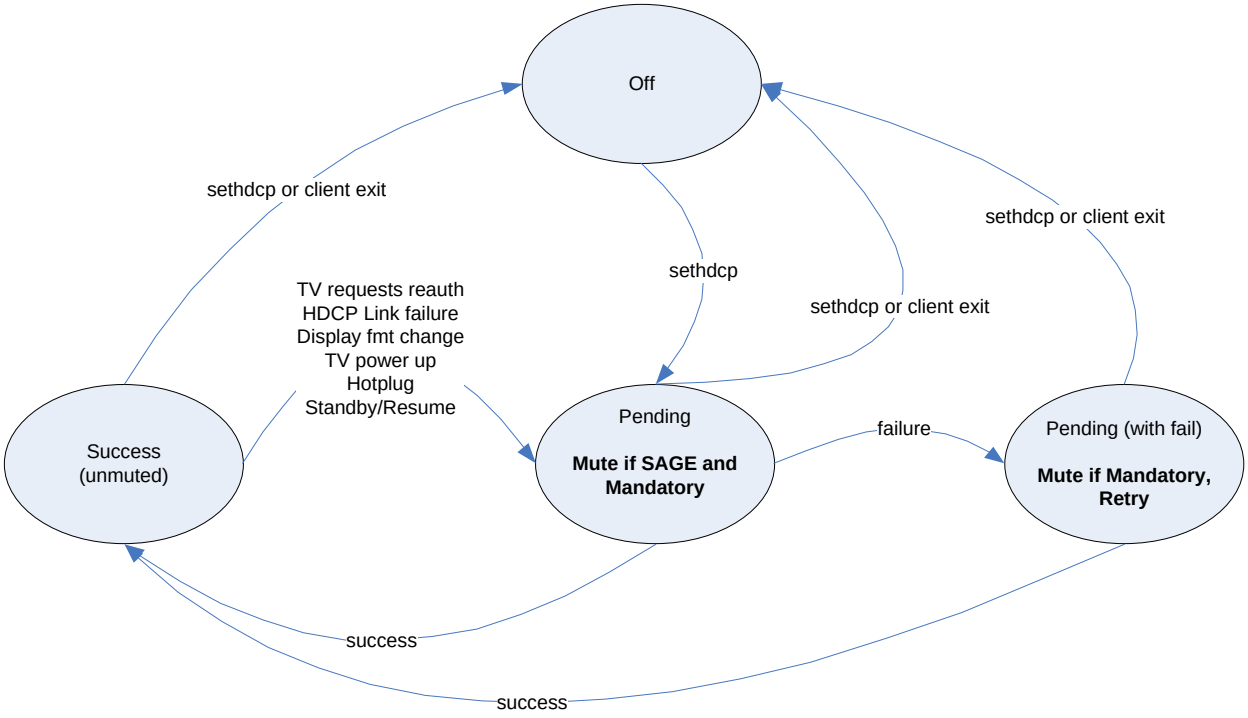
HDMI output

NxServer opens and manages the HDMI output handle. There are several ways a client can interact with HDMI:

- Use NxClient_SetDisplaySettings to control display settings. NxClient_DisplaySettings.hdmiPreferences has HDMI specific settings.
- Use NxClient_GetDisplayStatus for HDMI and HDCP status.
- Open a read-only alias of the HDMI output for reading status. See nexus/nxclient/apps/setdisplay.c for example code.

- Get HDMI and HDCP callbacks using NxClient_GetCallbackStatus (polling) or NxClient_StartCallbackThread (async).

NxClient apps can enable HDCP by setting NxClient_DisplaySettings.hdmiPreferences.hdcp. NxServer will aggregate the HDCP settings from all clients and decide if HDCP should be off or on. If HDCP is on and authentication fails, the server will notify clients (see above for callbacks and status). The clients are responsible for displaying graphics to notify the user of the failure. If the aggregate HDCP level for the system is "Optional", audio/video will continue even after the failure. If the level is "Mandatory" the server will mute HD video until HDCP can be authenticated. The HDCP state machine in nxserver works as follows:



NxServer supports dual HDMI outputs in multi-session configurations. Session 0 uses HDMI 0 and session 1 can use HDMI 1. See the "-session0" and "-session1" command line options.

HDMI input

See nexus/nxclient/apps/hdmi_input.c for an example.

The basic calls are:

- NxClient_Alloc - have the server allocate simple video and audio decoder handles for the client.

- NxClient_Connect – ask NxClient to connect a video window and audio decoder to the simple decoder handles.

- NEXUS_HdmiInput_Open – the client opens the HDMI input interface directly. All settings, status and callbacks are available.

- NEXUS_SimpleVideoDecoder_StartHdmiInput – route video from HDMI input to the display.

- NEXUS_SimpleAudioDecoder_StartHdmiInput – route audio from HDMI input to the audio outputs.

A SurfaceClient must be allocated if the application wants to control window position or z-order.

See nexus/nxclient/apps/transcode.c for encoding the HDMI input. Run the app with the "-hdmi_input" option.

Changing the display format

See nexus/nxclient/apps/setdisplay and the "-format" command-line option for an example.

NxClient handles display format change seamlessly for clients. Client graphics and video are resized automatically. HDMI EDID preferences and requirements can be followed or ignored.

NxServer and SurfaceCompositor go through a protocol which waits for blits to complete, temporarily disables SurfaceCompositor, resizes framebuffers, then re-enables composition.

Clients can request a callback from SurfaceClient so that graphics can be re-rendered to avoid scaling. But this is optional.

Stereoscopic TV (3DTV)

NxClient supports stereoscopic/3DTV sources and displays. Source and display are controlled separately with the NxClient API and with the reference apps in nexus/nxclient/apps. The orthogonal mapping is as follows:

	3D source	2D source
3D display	3D video	2D video by replicating the source to left and right eye
2D display	2D video by only displaying left eye	2D video

The display can be switched in and out of 3D mode at runtime using nexus/nxclient/apps/setdisplay. The following are example command lines:

Use case	App
Enable full-res 3D display	setdisplay –format 1080p3D
Enable half-res 3D display	setdisplay –format 1080p -3d lr setdisplay –format 1080p -3d ou
Enable half-res 3D display with no hardware assist (required for 65nm silicon, but available for all silicon)	nexus nxserver –native_3d off setdisplay -3d lr
Disable 3D display (go back to 2D)	setdisplay -3d off

You can also start nxserver in a 3D mode (with either the –display_format or –lr command line options) and still switch it later with the setdisplay app.

Nexus supports auto-detection of a 3D source using stream meta-data. However, many 3D streams lack that metadata. The alternative is for the app to explicitly state that a stream is 3D. The "play" app does this with the "-3d" option. Be aware that this "-3d" option only applies to the source, not the display.

SurfaceCompositor supports a client submitting stereoscopic graphics. It can submit a single surface which contains both left and right eye graphics. See nexus/nxclient/apps/3dtv_client. The mapping of 3DTV graphics to 3D and 2D sources is the same for graphics and video, as noted above.

The nxclient API's for controlling 3DTV are as follows:

- Use NxClient_DisplaySettings.format for full-res 3D. There are special NEXUS_VideoFormat values for full-res 3D.

- Use NxClient_DisplaySettings.display3DSettings.orientation for half-res 3D. You do not need to set this full-res 3D.
- There is no API for setting HDMI Vendor Specific Information (VSI) for 3D. NxServer does this automatically.
- Use NEXUS_SurfaceClientSettings.orientation for 3D graphics.
- Use NEXUS_VideoDecoderSettings.customSourceOrientation and .sourceOrientation to explicitly set a source as being 3D.

NOTE: NxClient does not currently support picture-in-graphics 3D video on 65nm (that is, non-native 3D and less-than-fullscreen video). If you are doing 3D video on 65nm, it must only be full-screen.

Userdata capture and Display VBI

Vertical blanking interval (VBI) information includes closed captioning, teletext, (copy generation management system) CGMS, (Widescreen signaling) WSS and others.

Unlike single-process Nexus, closed caption data is only provided to clients as raw user data. All user data parsing is handled in the application. This makes parsing very flexible. Without this, the regular VideoDecoder had various filter options to deal with multi-protocol streams.

The nexus/nxclient/apps/ccgfx application shows how the magnum/commonutils/udp library can be compiled into an application for parsing and how the BSEAV/lib/dcc can be integrated using SurfaceCompositor for graphics rendering of EIA608 and 708 data.

If the client sets NEXUS_SimpleVideoDecoderClientSettings.closedCaptionRouting to true, Nexus will parse userdata for closed captioning and automatically route to a VBI capable display (usually the SD display).

The client can write its own closed caption, teletext, CGMS and WSS data to a VBI capable display using NxClient_Display_WriteClosedCaption, NxClient_Display_WriteTeletext, NxClient_Display_SetWss, NxClient_Display_SetCgms and NxClient_Display_SetCgmsB.

Audio PCM playback

See nexus/nxclient/apps/playpcm.c for an example. The calls are as follows:

- NxClient_Alloc - have the server allocate a SimpleAudioPlayback handle for the client.
- NEXUS_SimpleAudioPlayback_Acquire – acquire the handle from Nexus using the ID provided by NxClient.
- NxClient_Connect – have the server connect the SimpleAudioPlayback handle to an AudioPlayback resource which is connected to the AudioMixer.

- NEXUS_SimpleAudioPlayback_Start – start audio playback
- NEXUS_SimpleAudioPlayback_GetBuffer/WriteComplete – push PCM data into audio

Depending on the HDMI EDID capabilities, HDMI/SPDIF configuration, and last decoded codec, the HDMI and SPDIF outputs may be configured for compressed output. In that case, PCM playback will not be heard. To force HDMI or SPDIF to PCM output, guaranteed that PCM playback is heard, call NxClient_SetAudioSettings with NxClient_AudioSettings.hdmi.pcm and .spdif.pcm set to true.

Picture decode (with SID)

See nexus/nxclient/apps/picviewer.c and thumbnail.c for examples.

NEXUS_PictureDecoder is a virtualized interface. Each client can open its own instance and use it directly. See nexus/nxclient/apps/utils/picdecoder.c for a helper library for using PictureDecoder.

Many applications use software image decode libraries like libjpeg and libpng instead of SID.

Still decode (with HVD)

See nexus/nxclients/apps/playviewer.c for an example.

NEXUS_StillDecoder is a virtualized interface. Each client can open its own instance and use it directly. See nexus/nxclient/apps/utils/thumbdecoder.c for a helper library for using StillDecoder.

Transcode

See nexus/nxclient/apps/transcode.c for an example. It decodes a file and re-encodes it using other codecs or at a different resolution or bitrate. The transcode app uses the SimpleEncoder and SimpleVideoDecoder and SimpleAudioDecoder interfaces.

The app defaults to non-realtime encoding. Real time can be selected using the "-rt" command line option.

HDMI encoding can be done using the "-hdmi_input" option and no input file. This automatically selects real time encoding.

NxClient allows transcode to be performed by dynamically allocating decoders, encoders and displays from a pool. A system can transition from dual decode to single decode + single transcode, or from dual decode + dual transcode to quad transcode.

The client does playback and decoder just like a normal video decode app. But SimpleEncoder routes that decode away from the main display to a transcode display. The client also opens a

Recpump interface and registers that with SimpleEncoder. In the simple_decoder module, Nexus will create the StreamMux instance, build the PAT and PMT tables, and connect all transcoded and generated output to the recpump interface. After starting decode and encode, the client just works with the recpump interface to capture the stream data. That data can be written to file or sent over the network.

Display encode

See nexus/nxclient/apps/encode_display.c for an example. This application encodes the video and audio output for a session and writes to a file.

Display encode is different from transcode because it encodes the video and audio output of all clients in its session. This is used for Miracast or remote UI applications. The Broadcom Trellis software library contains a Miracast implemtnation built on top of NxClient.

The display and audio mixer interfaces must be allocated with the session, not with the encode client. The encode client simply starts the encode and captures data in the same way as the transcode client.

There are notable RTS limitations for display encode. Not all transcode displays have sufficient bandwidth to encode video and graphics. Please work with your FAE to determine those RTS limitations and what configurations are possible.

Multi-session

NxClient supports running multiple set-top instances on a single set-top. Each instance is called a session and has its own display and audio output. A dual session box could drive separate interfaces on dual HDMI outputs. A multi-session box code also drive one session on a local display and additional sessions streamed over the network. These encoded or streaming sessions are also known as miracast sessions or remote UI sessions.

Local sessions are created when NxServer starts. By default, session 0 is local. Additional local sessions can be started, or session 0 can be converted to an encoded session using the "-session0" command line parameter.

Encoded sessions are created on demand as encoder and display resources are available.

By default, all clients are on session 0. Clients can connect to other sessions by specifying NxClient_JoinSettings.session.

Also, the NXCLIENT_SESSION environment variable will automatically set the NxClient_JoinSettings.session variable. This allows are parent process to start in a session and all its child processes will start in the same session. For example:

```
# start server
nexus nxserver &

# start session 0 desktop
. nxclient.sh
desktop &

# start session 1 desktop
NXCLIENT_SESSION=1 desktop &
```

AudioCapture

See nexus/nxclient/apps/audio_capture.c for an example.

NxClient will create an AudioCapture interface which allows a client to get a PCM stream of audio for all clients in its session. This can be used for streaming audio to a Bluetooth headset.

The steps are as follows:

- NxClient_Alloc – have the server create an AudioCapture interface and connect to this session's mixer. This will require stopping and restarting all audio decodes and playbacks, which may result in a noticeable delay.

- NEXUS_AudioCapture_Open – open the AudioCapture interface as an alias for client use. The client calls Start, GetBuffer and ReadComplete. It can assume the interface is already connected.

Standby

See nexus/nxclient/apps/standby.c for an example.

A client can request that nxserver put the system into standby. The steps are as follows:

- NxClient_SetStandbySettings – Client requests standby mode along with its wake-up devices
- The server will notify clients by setting NxClient_StandbyStatus.
- Clients are responsible to poll NxClient_GetStandbyStatus. If the system is going into standby, the clients should shut down cleanly then call NxClient_AcknowledgeStandby.
- A client can tell the server that it will be ignoring standby notifications by setting NxClient_JoinSettings.ignoreStandbyRequest.
- After all clients have acknowledged, or are ignoring, or after a certain time threshold has been reached, nxserver will put Nexus into standby.
- Then then standby client calls Linux to put the system into standby.

DMA

The NEXUS_Dma interfaces works in client apps just like single-process application. However, one difficulty you may encounter is the memory mapping of the default heap. When the application calls NEXUS_Memory_Allocate or NEXUS_MemoryBlock_Allocate with default settings, the heap used for clients is the graphics heap, which only has eApplication mapping. In kernel mode this heap has no driver-side memory mapping.

If there is no driver-side memory mapping of device memory, the server or driver cannot flush on behalf of the client. Therefore, the app cannot set NEXUS_DmaJobBlockSettings.cached = true, which requests the driver to flush on behalf of the application.

Instead, the application should flush itself, either before the DMA read or after the DMA write. This turns out to be the most efficient method anyway; the driver-side method usually does extra flushes because it doesn't know if a flush can be skipped when there is no CPU write or read.

If you want to use NEXUS_DmaJobBlockSettings.cached = true, you must use a heap with driver-side mapping.

See nexus/nxclient/examples/dma.c to see how cache management should be done with DMA in a multi-process environment. It also demonstrates other methods, including the invalid method.

Fonts

Nexus and NxClient do not include font support. That is usually handled by applications using open source libraries like FreeType. However, nexus/nxclients/apps/utils has a bfont.c library which provides simple font support using pre-rendered fonts used by Brutus, Atlas and BSEAV/lib/bwin. These fonts are provided only for example use.

Callbacks

See nexus/nxclient/apps/setdisplay and the "-wait" option for an example.

NxClient has callbacks for HDMI output hotplug, HDMI output HDCP and display settings changes. The underlying mechanism is uni-directional and requires client polling. We also provide start/stop functions which creates a thread and provides asynchronous callbacks.

The NxClient callback mechanism is designed to be lightweight and to avoid complex synchronization when unregistering callbacks.

Picture Quality (PQ)

There are two places to manage picture quality, either for the whole display (including all video windows and graphics) or for a single video window.

Display-level PQ is controlled using NxClient_SetPictureQualitySettings. It currently only supports NEXUS_GraphicsColorSettings.

VideoWindow-level PQ is controlled using NEXUS_SimpleVideoDecoder_SetPictureQualitySettings. It supports color space conversion (CSC), digital noise reduction (DNR), analog noise reduction (ANR), de-interlacing (MAD) and scaler settings.

See nexus/nxclient/apps/setdisplay for example client code.

Fast Channel Change (FCC)

Fast Channel Change allows live video streams to start faster by priming them before they are started. See nexus/docs/Nexus_Usage.pdf for a general description. FCC is for live video only and does not apply to playback or audio.

In a multi-process environment, it is supported using the NEXUS_SimpleVideoDecoderPrimer interface which uses the NEXUS_SimpleVideoDecoderHandle.

See nexus/nxclient/apps/live for a functioning example. You will need to read utils/live_decode.c to see how the primer is used.

Audio Decoder Primer

Audio Decoder Primer allows playback streams to start audio without long delays. It is useful for main/PIP audio switches or switching between alternative audio streams without disrupting video. See nexus/docs/Nexus_Usage.pdf for a general description. Audio Decoder Primer is for playback audio only and does not apply to live.

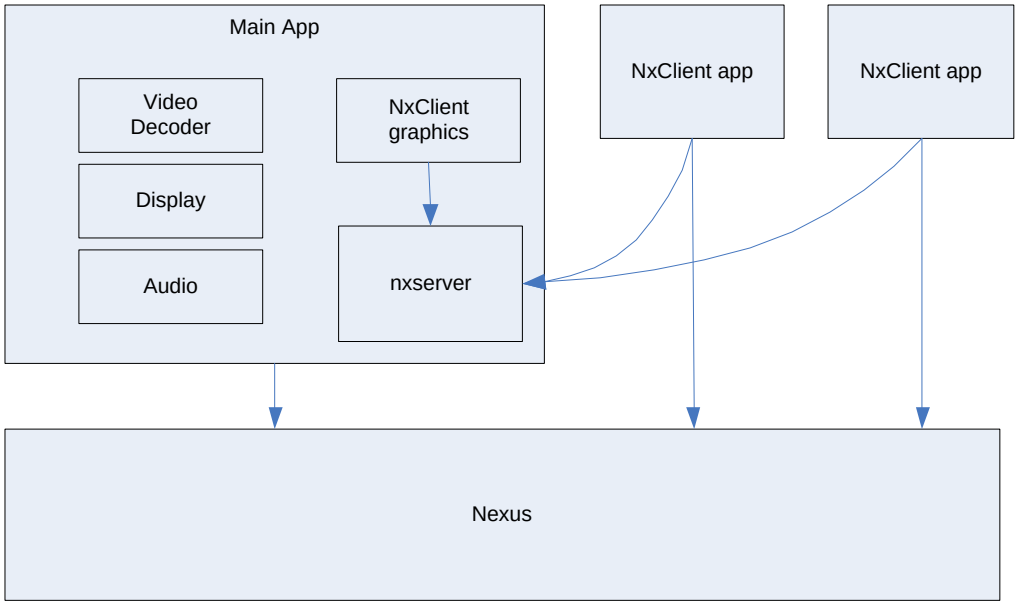
In a multi-process environment, it is supported by setting NEXUS_SimpleAudioDecoderStartSettings.primer.pcm = true or .compressed = true. After being started, if the regular AudioDecoder is not present or is removed, the audio stream will be primed. Then, when the regular AudioDecoder becomes available or is assigned, the audio stream will be able to be heard almost immediately.

See nexus/nxclient/examples/pip_swap.c for an example.

External App

NxClient provides an option for linking an external application with nxserverlib. This is a transition path for non-NxClient systems for integrating NxClient applications without a complete rewrite of the original, main application.

For a working example, see nexus/nxclient/apps/switched_decode.c.



The main app must link with libnxserver.a.

After opening NEXUS_Display, it calls nxserverlib_init() and passes the display handles. By passing the display handles, the main app gives up the ability to call any destructive NEXUS_Display call which nxserver or SurfaceCompositor will call. That includes NEXUS_Display_SetGraphicsSettings, NEXUS_Display_SetGraphicsFramebuffer, NEXUS_Display_SetSettings, NEXUS_Display_SetVsyncCallback and (of course) NEXUS_Display_Close.

NOTE: NEXUS_Display_SetVbiSettings, NEXUS_Display_WriteTeletext, NEXUS_Display_SetWss and NEXUS_Display_SetCgms are called by nxserver, so the main app should use NxClient instead. But nxserver could be reworked to avoid them.

Nxserver avoids calling NEXUS_Display_AddOutput and NEXUS_Display_RemoveOutput when external app mode is enabled.

This means the main app must always use NxClient for graphics, display format change, and VBI encoding. The main app can still call NEXUS_Display to manage outputs, video windows and picture quality.

The main app must also call nxserver_ipc_init() to enable IPC from client apps. After calling nxserverlib_init() and nxserver_ipc_init(), NxClient is running in graphics-only mode and the main app is free to do audio and video decode with its existing code. Both the main app and NxClient applications can do graphics using NxClient and SurfaceCompositor.

When the main app wants to enable audio and video decode using NxClient, it does the following transition:

- The main app must close all decoder resources. This includes VideoDecoder, VideoWindow, StcChannel, AudioDecoder, AudioMixer, AudioPlayback and all other audio filter graph nodes other than the audio outputs that come from NEXUS_PlatformConfiguration.

- It calls nxserverlib_allow_decode(server, true) to allow NxClient to permit decode.

- NxClient apps can now use decode resources.

- When the app wants to reclaim control of the decoders, it should notify clients that they should release their decode resources.

- It then calls nxserverlib_allow_decode(server, false) which closes all decode resources in NxClient. Any client that was using decode resources will lose them immediately.

- The main app can then reopen and use decode resources.

NxServer and libnxserver.a

NxServer can be run as a ready-to-use application or compiled into another application as a library.

The default build creates the "nxserver" application which supports a set of command-line options. This is tested and ready to run on Broadcom reference platforms.

When building the standard "nxserver" application, nxserverlib.a is also generated. This library can be linked into a custom application. The header file for the library is nxserverlib.h. These API's are not proxied; they must be called locally. There are two API's available:

- nxserver_init and nxserver_uninit. These are very simple API's the preserve all the command line options and default NEXUS_Platform_Init, but allow linking into another application. See nxserver_main.c.

- nxserverlib_init and nxserverlib_uninit. These API's bring up the nxserver state machine, but require the calling application to bring up Nexus and the IPC library externally. This gives the application more customization options. See nxserver.c.

Another option is to customize nxserverlib itself. This is permitted, but discouraged. NxServer is complex code that is still being updated often. It will be difficult to take updates if you've made major changes.

Appendix A: Serialized global settings

When the same settings are exposed to multiple clients, race conditions can exist. NxClient has a simple method for serializing this access with minimal overhead.

Other alternatives that were considered and not used are:

- Each client has its own settings and the server combines them. This guarantees serialization, but is complex for both the server (how to combine) and the client (how to know which settings were applied and why).
- Clients must acquire a mutex before getting and setting settings. This would pose a risk to the system if a client acquired a mutex and did not release. The client would have to be killed before the rest of the system could continue.
- All settings are split into individual get/set functions. This removes any race condition for any one setting, but puts a burden on the server to manage intermediate states.

NxClient manages global settings with Get/Set functions with structures that have dependent settings. These settings structures also have a global sequence number. For example:

```
typedef struct NxClient_DisplaySettings
{
    unsigned sequenceNumber;
    NEXUS_VideoFormat format;
    NEXUS_DisplayAspectRatio aspectRatio;
} NxClient_DisplaySettings;
```

The sequence number is incremented on every successful Set call. If a Set call is made with a number that does not match the current number, then two clients have interleaved their Get/Set calls and so the mismatched Set call will fail. The client should Get/Set again. For example:

```
do {
    NxClient_DisplaySettings settings;
    NxClient_GetDisplaySettings(&settings);
    settings.format = NEXUS_VideoFormat_e720p;
    rc = NxClient_SetDisplaySettings(&settings);
} while (rc == NXCLIENT_BAD_SEQUENCE_NUMBER);
```

This allows the clients to naturally merge their settings without any server logic. If the call succeeds, then the client can know that its settings were applied.

If the client implements the loop, then the logic is “last one wins.” If the client doesn’t implement the loop, then the logic becomes “first one wins.”

Appendix B: IPC mechanism

Multi-process support is implemented using an embedded inter-process communication (IPC) mechanism. NxClient uses nexus/lib/ipc which is built on top of Unix domain sockets and code-parsing Perl scripts.

Because the IPC mechanism is hidden under the NxClient API, you can substitute your own IPC mechanism and still reuse the NxClient API as well as most of the nxserver implementation.

NxClient does not support "attr{}" hints like nelem or memory=cached. Therefore, NxClient does not support passing of virtual addresses (you must pass offsets). It does not support variable sized arrays.

Appendix C: Relationship between NxClient_Alloc and NxClient_Connect resources

The NxClient Connect and Alloc API's can adapt to a variety of configurations, but there are practical limitations due to the current server implementation. Some of them are:

- The ids used by Connect can come from multiple Allocs as long as multiple ids of the same type come from the same Alloc. So, simpleVideoDecoder[].id can come from one Alloc and simpleAudioDecoder.id can come from another Alloc, but simpleVideoDecoder[0].id and simpleVideoDecoder[1].id must come from the same Alloc.

- Audio and video that will be lipsync'd should come from a single connect call, but we support separate connect calls. If you use separate connect calls, one serial STC's will be allocated for video and one for audio, but only one will be used. A resource is wasted.

- One connect only supports one audio decode. So, main and PIP (dual decode) require separate alloc and connect calls.

- One connect only supports simpleEncoder[0], not simpleEncoder[1] or greater. If you want to do multiple encodes, use multiple connect calls.