Nexus

Multi-Process

Broadcom Corporation
5300 California Avenue
Irvine, California, USA 92617
Phone: 949-926-5000
Fax: 949-926-5203

Web: www.broadcom.com

Revision History

Revision	Date	Change Description
0.1	10/8/10	Initial draft
0.2	10/29/10	Clarify trusted and untrusted clients
1.0	2/17/11	Release of user mode multi-process support
2.0	9/13/11	Change client modes to unprotected/protected/untrusted. General refactoring.
3.0	9/23/11	Separated usage and internal design documents
3.1	10/6/11	Added description of handle tracking vs. verification, table of interfaces and new API's, typical system diagrams
3.2	10/21/11	Expanded bipc, revised typical scenarios
3.3	8/7/12	Client heaps Dynamic allocator for frontend Virtualization of I2C interface Add NEXUS_ClientConfiguration for limiting interface instances used by untrusted clients
3.4	2/5/13	Add verified mode
3.5	9/24/13	Describe SimpleStcChannel Deprecate unprotected mode
3.6	10/1/13	Add Non-Root Access section
3.7	1/9/14	Add "Aliased Interfaces" and NEXUS_ALIAS_ID. Add SimpleEncoder
3.8	3/6/14	Remove unprotected mode
3.9	8/12/14	Add LXC section
3.10	1/20/15	Rework "Handle Sharing" section
3.11	1/29/16	Rework "Concepts" section

Table of Contents

Introduction

Complex systems are often designed to run software in multiple processes. Each process has its own memory address space and its own interface with the OS, which provides for greater stability. The approach is used to manage software complexity, for instance, not having to integrate separately-developed pieces of software. Another use is to isolate untrusted, secondary processes from the server process.

Nexus (and, below that, Magnum) is inherently a singleton. It must run in only one process or only one driver. Therefore, any multi-process support requires marshalling Nexus public API functions from one or more client processes to the server process. The Nexus coding convention makes this marshalling possible with an auto-generated IPC thunk.

Nexus multi-process support is not a single technique or single software layer. It is a set of techniques which help enable a multi-process system design. Customers may use only a portion of these techniques. However, we have found that these techniques often have a mutual dependence.

This proposal does not address system-level security issues outside of Nexus. It is focused on enabling Nexus to work in a properly secured system. It is assumed that the OS ensures that untrusted client processes are unable to cause harm through either direct memory or register access or through OS system calls. It is also assumed that all Magnum and Nexus code and firmware is trusted. The only code that is untrusted is running in client processes.

The easiest way to get started with Nexus multi-process is to use NxClient. It contains a ready to use server application and a set of client applications. See nexus/nxclient/docs/NxClient.pdf for documentation.

Concepts

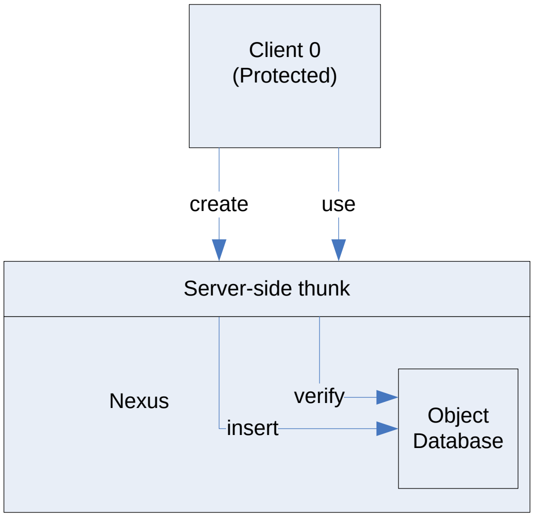
The secure, multi-process architecture is built around two things:

- An IPC mechanism for marshalling calls from client to server (including marshalling callbacks from server to client)
- a set of policies for the server to protect itself from poorly behaved or malicious clients

When Nexus is running in kernel mode, the auto-generated IOCTL proxy allows each client application to communicate with the driver (nexus.ko). When Nexus is running in user mode, the auto-generated socket proxy allows each client to communicate with the server process (linked with libnexus.so). This document does not describe the internal workings of these IPC mechanisms.

Nexus protects itself from poorly behaved or malicious clients by managing resources through handles. Each resource used by a client is associated with a handle which is opened, used and closed. A client can only use handles that it has opened itself, unless a “shared handle” exception is made. If the client calls NEXUS_Platform_Uninit or its process terminates without calling NEXUS_Platform_Uninit, all handles which are left open are cleaned up by the server (i.e. garbage collection). There is no run-time garbage collection.

Nexus implements handle tracking on the server using an object database (b_objdb). After each handle is opened, it is inserted into the object database with its handle value, handle type and the client that opened the handle. On every use of that handle, the value, type and client is verified. If verification fails, the Nexus function is automatically failed. When the handle is closed, either manually or with automatic cleanup, it is removed from the object database.



Given this simple resource management system, Nexus provides various techniques to share resources. These techniques include virtualization, dynamic allocation and read-only aliasing. See “Nexus Interface Multi-Process Capabilities” for details.

Client Modes

Nexus allows clients to operate in three modes: NEXUS_ClientMode_eVerified, eProtected and eUntrusted. We recommend that you run in either Protected or Untrusted mode. The default mode is Protected mode. Running in Verified mode permits unmanaged handle sharing that can be negative implications, described below.

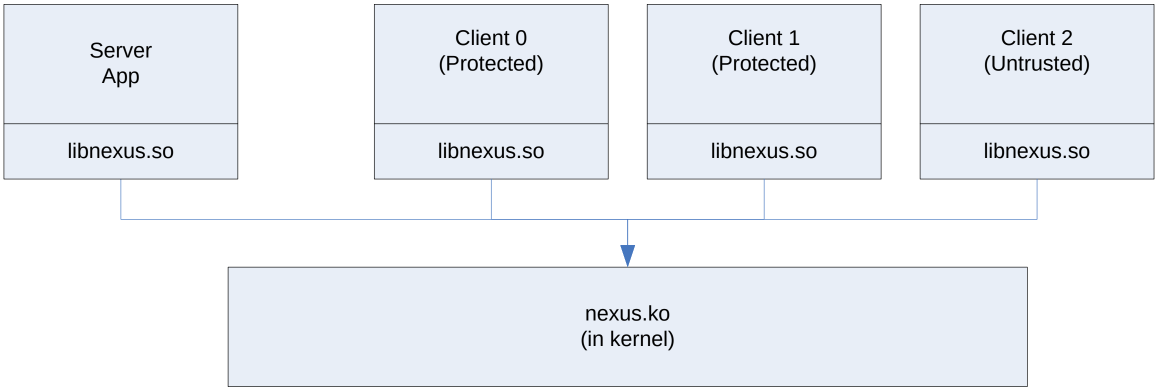
We recommend that you determine your client mode at the beginning of your development. If your client runs in protected or untrusted mode (as described later in this document) you will end up discovering all of Nexus' security requirements of Nexus multi-process by implication. This document will explain why those requirements are there, and may relieve some frustration when you hit them, but the only necessary condition is just setting the mode.

If you try to switch from a less restrictive mode to a more restrictive mode (for instance, from verified to protected), it will likely require extensive system redesign. Instead, by setting your desired mode in the beginning, Nexus will inform you if certain handles or functions are inaccessible and you can adjust your design right away.

The three client modes are summarized as follows:

Feature	Verified	Protected	Untrusted
Nexus API	Full access	Full access	Limited access
Handle tracking	Yes	Yes	Yes
Handle verification	Yes, but not for handle owner	Yes	Yes
Garbage collection	Yes	Yes	Yes
Client can crash?	Yes	Yes	Yes
Client can be malicious?	No	No	Yes
Heaps	All	Granted by server	Granted by server
Recommend for clients	No	Yes, but only for limited cases	Yes, in all cases

A system can run with a mix of client modes, as follows:



Untrusted Mode

Untrusted clients have all the same safe guards as protected clients, but there are limitations placed on which API's can be used and how they are used. A protected client can open/create/acquire all resources; therefore it is not inherently safe. For instance, a client which performs live decode will need direct access to tuners, transport, frontend and security. Nexus can recover resources from a crash, but the resources themselves should not be exposed to untrusted code.

Access is limited in two ways: first, by limiting which interfaces an untrusted client may call; second, by limiting the instances of those interfaces that an untrusted client may use, either by limiting to a total number or by only allowing specific ID's to be used.

The limitation of interfaces is enforced with an access control list (ACL) defined in nexus/build/tools/common/nexus_untrusted_api.txt. The ACL is used to generate proxy code which enforces this in the server. For example, an untrusted client may use the Playpump interface, but may not use the Timebase interface. The ACL lists every header file which contains functions that an untrusted may call; all other functions are restricted.

The limitation of interface instances is controlled by a server application. The server sets the NEXUS_ClientResources structure for each client. The NEXUS_ClientResources structure allows limitation by count or by ID list, depending on the nature of the interface.

```
typedef struct NEXUS_ClientResources
{
    NEXUS_ClientResourceIdList simpleAudioDecoder;
    NEXUS_ClientResourceIdList simpleVideoDecoder;
    NEXUS_ClientResourceIdList surfaceClient;

    NEXUS_ClientResourceCount dma;
    NEXUS_ClientResourceCount graphics2d;
    NEXUS_ClientResourceCount graphicsv3d;
```

```
        NEXUS_ClientResourceCount pictureDecoder;
        NEXUS_ClientResourceCount playpump;
        NEXUS_ClientResourceCount recpump;
        NEXUS_ClientResourceCount simpleAudioPlayback;
        NEXUS_ClientResourceCount stcChannel;
        NEXUS_ClientResourceCount surface;
    } NEXUS_ClientResources;
```

NEXUS_ClientResources can be set in three ways:

1. using NEXUS_ClientSettings.configuration when calling NEXUS_Platform_RegisterClient
2. using NEXUS_PlatformStartServerSettings.unauthenticatedConfiguration when calling NEXUS_Platform_StartServer
3. using NEXUS_Platform_SetClientResources at run-time. However, this API does not revoke already used resources. If the client already has resources which exceed the new limits, SetClientResources will fail. The server must remove the resources or unregister the client before calling SetClientResources again.

There are defaults for the "Count" interfaces, but no defaults for the "IdList" interfaces. The application must grant access to specific ID’s for the “IdList” interfaces.

If you try to open an instance that hasn't been granted, you'll see an error like this on the server console:

```
### 00:00:24.294 nexus_client_resources: surfaceClient acquire rejected:
ID 4 not allowed
!!!Error rc(0xa) at
nexus/modules/surface_compositor/src/nexus_surface_client.c:463
```

Untrusted mode is recommended for potentially malicious clients. A malicious client could be an Internet-downloaded binary which is actually trying to crack or disable your system. Ultimately, any non-malicious bug could end up attempting the same thing as malicious code, but it is more unlikely.

Handle Sharing

One common approach used for multi-process system design is to share resources by sharing handles to those resources between different processes. We do not recommend this approach, but Nexus does support the technique with the following limitations.

A client which uses a handle it did not open must be registered as NEXUS_ClientMode_eVerified. This bypasses the handle ownership check. NEXUS_ClientMode_eProtected and eUntrusted mode will prevent handle sharing. Another option is for the owner of the handle to call NEXUS_Platform_SetSharedHandle to bypass the ownership test on each handle it intends to share.

Even if another process is allowed to use someone else's handle, the process that opened the handle remains the owner. When the owner process exits (in either a clean exit or a crash) any handle that it owns which is still opened will be automatically closed. If another process has that handle, any use will be invalid. If that process uses the handle in a "SetSettings" or "GetStatus" call which has a return code, it will get an error. If that process makes a "GetSettings" call which returns void, it will just get uninitialized data. Handle sharing really only works if the process that opens the handle is guaranteed to never crash.

If two processes are using the same handle, they must coordinate their use of that handle. If they call the same Get/Set calls, they must use some IPC semaphore to serialize access. That is, some IPC above Nexus must ensure the calls arrive as Get/Set/Get/Set (which is a proper incremental change) and not Get/Get/Set/Set (where the second Set destroys the first Set).

Also, Nexus does not support callbacks (that is, NEXUS_CallbackDesc) to more than one process within the same Settings struct. So, if a Settings struct contains even one NEXUS_CallbackDesc, it must only be called by one process; serializing with an IPC semaphore will not help. If you want to pass the exclusive use of a handle from one process to another, be sure to clear your callbacks by setting NEXUS_CallbackDesc.callback = NULL before allowing the other process to call the same SetSettings call.

The rest of this document describes techniques that help you avoid these pitfalls by not doing handle sharing. Instead, we have designed virtualized resources, added dynamic allocation, and created higher-level abstractions which provide client/server interfaces.

Client Authentication

The server has two ways of allowing clients to connect. It can explicitly register individual clients to join or it can allow any unauthenticated client to join.

Authentication is not the same as trust. Authenticated only has to do with knowledge of a client's identity. We recommend unauthenticated clients only for simple, closed systems. nexus/examples/multiprocess/blit_server uses unauthenticated clients; nexus/nxclient only uses authenticated clients.

Authentication is set up on the server using NEXUS_Platform_RegisterClient. Unauthenticated client options and defaults are set on the server using NEXUS_PlatformStartServerSettings.

Client Heaps

Heap management is also done through handle verification. The server grants each client access to a set of heaps (or no heaps). The client is able to mmap through the server only those heaps it has been given access to. The server ensures that only the client's granted heap handles are used in the API.

The heap index on the client is not the same as the heap index on the server. If the client application does not specify a heap when using an interface (which is normal), the server will pick the default heap, which is generally the client's heap[0], not the server's heap[0].

Also, the server has the ability to pick a default heap based on memory mapping requirements. Some interfaces (like graphics2d packet blit of playpump) require a heap with NEXUS_MemoryType_eFull mapping, which may not be the client's heap[0]. The server will then pick the first client heap that has eFull mapping.

Nexus Interface Multi-Process Capabilities

The following is a summary of most Nexus interfaces and their multi-process capabilities. These capabilities are defined on the following page.

Interface	Capability	New API for multi-process systems
AudioCapture	Aliased	Unchanged
AudioDecoder	Exclusive and dynamically-allocated	NEXUS_ANY_ID support added
AudioMixer and its outputs	Exclusive and statically-allocated	SimpleAudioDecoder wrapper
Display and its outputs	Exclusive and statically-allocated	SimpleVideoDecoder and SurfaceCompositor wrappers
Dma	Virtualized	Unchanged
File	Local	Unchanged
Frontend	Exclusive and dynamically-allocated	NEXUS_Frontend_Acquire
Graphics2D	Virtualized	Unchanged
Graphics3D	Virtualized using OpenGL-ES	Unchanged
HdDviInput	Aliased	Unchanged
HdmiInput	Exclusive and statically-allocated	Unchanged
HdmiOutput	Aliased	Unchanged
I2c	Virtualized	Unchanged
InputBand	Static and untracked	Unchanged
IrInput, UhfInput, Keypad, Gpio, etc.	Exclusive and statically-allocated	InputRouter
KeySlot	Exclusive and dynamically-allocated	Unchanged
Message	Exclusive and dynamically-allocated	Unchanged
ParserBand	Exclusive and dynamically-allocated	NEXUS_ParserBand_Open
PictureDecoder	Virtualized	Unchanged
PidChannel	Exclusive and dynamically-allocated	Unchanged
Playback	Local	Unchanged
Playpump	Exclusive and dynamically-allocated	NEXUS_ANY_ID support added
Record	Local	Unchanged
Recpump	Exclusive and dynamically-allocated	NEXUS_ANY_ID support added
Spi	Exclusive and statically-allocated	Unchanged
StcChannel	Exclusive and dynamically-allocated	SimpleStcChannel wrapper
StillDecoder	Virtualized	Unchanged
Surface	Exclusive and dynamically-allocated	Unchanged
Timebase	Exclusive and dynamically-allocated	NEXUS_Timebase_Open
VideoDecoder	Exclusive and statically-allocated	SimpleVideoDecoder wrapper
VideoEncoder	Exclusive and statically-allocated	SimpleEncoder wrapper

The preceding multi-process capabilities are as follows:

Local Interfaces

If an interface does not correspond to a hardware resource, is not called by a server-side module and calls no private API's, then it is run locally in the client. Examples include Playback, Record, and File. These are not multi-process capable, but they have no need for multi-process functionality.[1]

Virtualized Interfaces

Some Nexus interfaces expose limited hardware, but the software implementation is able to virtualize the access so that it appears *as if* multiple users have exclusive access. Examples include Graphics2D, Dma and PictureDecoder, and I2c.

For these virtualized interfaces, the optimal implementation is for each client to have its own handle. It allows each context to have a separate data FIFO and callback, which gives better performance. We do not recommend that these resources be shared.

Aliased Interfaces

Some interfaces allow for multiple opens using NEXUS_ALIAS_ID. The server opens the master with the regular ID (for example, NEXUS_HdmiOutput_Open(0)). Then, a client opens a slave as an alias on that master by adding the NEXUS_ALIAS_ID macro (for example, NEXUS_HdmiOutput_Open(0 + NEXUS_ALIAS_ID)).

For HdmiOutput, the alias has read-only access to status and EDID information. The server opens HdmiOutput and connects it to the display.

For AudioCapture, the alias has read-write access to Start/Stop/GetBuffer/ReadComplete. The server opens AudioCapture and connects it to the mixer.

[1] Astm and SyncChannel call private API's, so they are server-side. The Surface module is used by server-side modules like Display and Graphics2D, so it is server-side. Also, some server-side interfaces have client-side implementations of specific functions. NEXUS_Surface_Flush and NEXUS_Memory_FlushCache have local implementations. This is noted with the attr{local=yes} attribute.

Exclusive and Dynamically-Allocated Interfaces

Some Nexus interfaces expose hardware where there are plenty of identical resources available. These can be dynamically allocated. Once the resource is obtained, the underlying hardware is exclusively used by the client. Examples include Message, PidChannel, ParserBand, Timebase[2], Recpump, Playpump and Frontend.

Message and PidChannel were already dynamically allocated in the original Nexus API. We added a macro called NEXUS_ANY_ID which, when used as the index, will dynamically allocate an unused resource.

For pid channels, no hardware index is specified by default. But if you specify a specific hardware PID channel, a conflict may result.

For message filters, no index is given. Nexus will manage the dynamic allocation.

Frontend uses a dynamic allocator to select a frontend handle based on capabilities. See NEXUS_FrontendAcquireSettings. This is especially useful for frontends with full-band capture digital tuners and demods with 8 or more channels. You can also select a frontend based on an index number.

Exclusive and Statically-Allocated Interfaces

These interfaces represent hardware that is for the exclusive use of the client, but the resource is not obtained dynamically. The client must know which resource it wants.

The Spi interface is a one-to-one connection that is opened by a specific index. If any sharing is needed, the client should open it, use it, then close. If the open fails, the SPI bus is already in use and is not available.

Interfaces like VideoDecoder and Display are exclusive and statically-allocated. If you chose not to use the new client/server API's (documented later in the document), then clients can open them in this fashion. Any sharing requires closing the interface in one client and reopening it in another client.

Static and Untracked Interfaces

Interfaces like NEXUS_InputBand are not handle based. They are enum based. There is no tracking and no dynamic allocation. Untrusted clients are prevented from using them.

[2] ParserBand and Timebase were originally conceived of as enum-based interfaces. They have been changed to handle-based interfaces but with backward compatibility for enum-based usage for existing apps.

Challenging Interfaces

Display

The most difficult problem in a multi-process system is the management of the display. Most other resources can be easily closed in one client and reopened in another. Some memory fragmentation risk can occur, but it is minimal and can be mitigated with a memory heap scheme. But closing the display means losing sync on all video outputs.

Also, most multi-process systems allow multiple clients to be rendered to the display at the same time. So, the display needs to be opened by one process and somehow shared with all of the clients. However, only one process can open the display. Therefore no other protected client can use that display handle. This means that those processes which did not open the display cannot set the graphics framebuffer, or control VBI output, or connect a video decoder to a video window.

Also, graphics is highly performance intensive. Any multi-process graphics solution must be designed for optimal performance.

AudioMixer

Just like the Display, the AudioMixer is an aggregator. The performance requirements are not as high, but the handle-sharing issues are the same.

VideoDecoder and AudioDecoder

The regular VideoDecoder and AudioDecoder were not designed to be client/server capable. You can use them in a client, but only if you open/close on each use and if you provide application IPC to make the connection to the display.

User Input

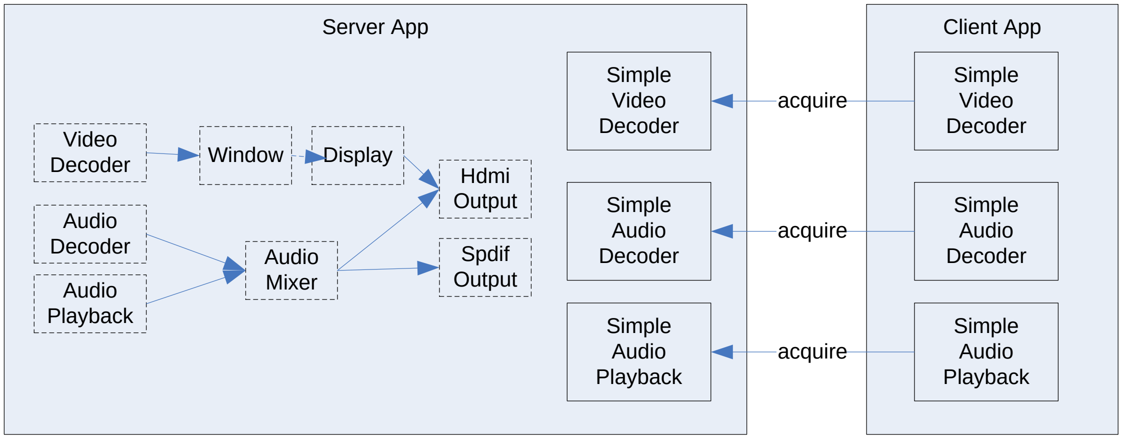
User input devices like IrInput or UhfInput must be managed by the many server application and routed to the appropriate client based on “focus”.

New Client/Server Interfaces

To address the problems related to sharing the display, audio mixers and decoders, we have created several new Nexus interfaces and modules.

Simple Decoder

The Simple Decoder provides a client/server interface on top of the standard nexus VideoDecoder and AudioDecoder API's. It makes it easy for clients to acquire a decoder, use it, and then release it after use.



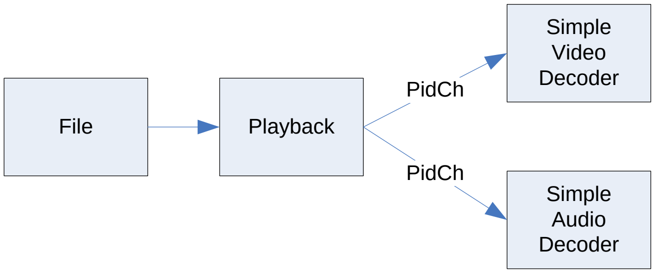
If you do not use the simple decoder, there are two other options, but both have some risk. First, the client can open the decoder before each use and close it after each use[3]. The application will need to provide some application IPC to the server so that a connection from the VideoDecoder can be made to the VideoWindow, or the client can run in NEXUS_ClientMode_eVerified and share the VideoWindow or Display handle. Second, the client can run as NEXUS_ClientMode_eVerified and get a connected decoder from a handle cache.

Even if you have access to the display or are single-process, you may just want a simpler decode API. If the Simple Decoder meets your needs, your app will be a whole lot easier to write and maintain.

The essence of the simple decoder design is that it automatically performs downstream configuration with system-level resources. This includes automatic configuration of lipsync, video HD/SD simul mode and audio mixing. The server application creates the simple decoder, populated with lower-level resources (like a container), and it becomes available to clients to acquire.

[3] This will cause some memory fragmentation, but it can be mitigated by using heaps.

The following shows how the client configures the frontend (from file, network or memory) and simple decoder handles the decode and output.



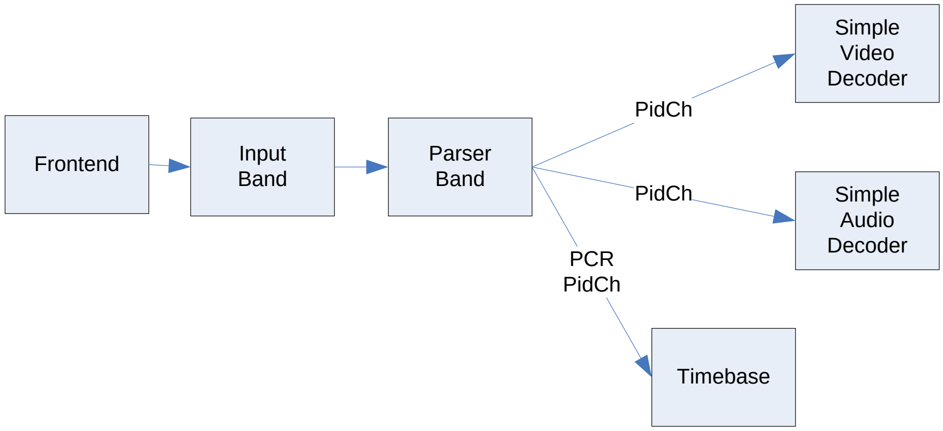
These simple decoder interfaces have the following advantages:

- Protected clients can acquire/release decoders that are safe to use.
- The client does not manage TSM interconnections, including basic TSM, SyncChannel and ASTM. The client sets an enum specifying whether the TSM will be PCR or DVR based. The required configuration happens internally.
- Audio decoder filter graph is automatically configured based on codec. The server determines how AC3, DDP, WMA, etc. will be routed to DAC, SPDIF, and HDMI outputs. The client simply starts decode.
- Client does not have to configure audio mixing. All audio post-processing and mixing is managed by the server. The client simply acquires and uses the simple decoder and playback interfaces.
- Regular video decode and mosaic video decode are abstracted by the server. This allows a client to be written in a generic way.
- Video decoder capabilities (i.e. max source size) can be specified at start-decode time. The normal nexus video decoder API requires a more complex interaction to reconfigure this.

See nexus/examples/multiprocess/decode_server and decode_client for an example.

Live decode with Simple Decoder

This also applies to live decode which requires different upstream interfaces, but the downstream is the same:



ParserBand and Timebase have been given new Open/Close functions.[4]

Frontend also has new NEXUS_Frontend_Acquire/Release function.

InputBands are not protected resources. They are simple enums. The client must know which input band it should use and whether it has access to it.

See nexus/examples/multiprocess/decode_server and live_client for an example.

SimpleStcChannel

SimpleDecoder includes a high level interface for managing lipsync. Instead of using NEXUS_StcChannel, clients should create a NEXUS_SimpleStcChannel. The underlying StcChannel can then be assigned or removed as needed as underlying decoder resources are assigned.

The assignment of SimpleStcChannel to SimpleVideoDecoder and SimpleAudioDecoder must be done before either decoder is started. This allows internal, high-precision lipsync algorithms to work optimally.

[4] In order to have the statically used enums work with dynamically allocated pointers, Nexus has an internal conversion and lookup feature. For instance, if NEXUS_ParserBand_e0 is used via enum, it will never be dynamically allocated.

Timebase management with Simple Decoder

The server manages timebases for a variety of configurations like main and PIP decode, high-jitter, and transcode. The client can specify that its content is high-jitter by setting a high jitter mode along with the normal pcr settings, like this:

```
NEXUS_SimpleStcChannelSettings stcSettings;
NEXUS_SimpleStcChannel_GetSettings(stcChannel, &stcSettings);
stcSettings.mode = NEXUS_StcChannelMode_ePcr;
stcSettings.modeSettings.pcr.pidChannel = pcrPidChannel;
stcSettings.modeSettings.pcr.offsetThreshold = 0xFF;
stcSettings.highJitter.mode = NEXUS_SimpleStcChannelHighJitterMode_eDirect;
NEXUS_SimpleStcChannel_SetSettings(stcChannel, &stcSettings);
```

Based on that high jitter enum, Nexus will set recommended values for low-level settings like maxPcrError, trackRange, jitterCorrection, offsetThreshold, and more.

For a Main/PIP system, SimpleStcChannel assigns timebases as follows:

0 = main (normal)
1 = main (high jitter)
2 = pip (normal)
3 = pip (high jitter)
4 = first transcode
5 = second transcode
etc.

For a system without PIP, SimpleStcChannel assigns timebases as follows:

0 = main (normal)
1 = main (high jitter)
2 = first transcode
3 = second transcode
etc.

Simple Encoder

The SimpleEncoder interface supports transcode and display-encode using the video and audio encoders.

Transcode means converting a file from one codec and bitrate to another.

Display encode means encoding the graphics, video and audio from a display so it can be streamed to a remote device.

See nexus/nxclient for functioning examples of both.

SimpleEncoder is not a required interface for client/server encode. If the decode and display resources for encode are not shared with the server or any other client, the client can open them directly. SimpleEncoder was created to share those decode and display resources with other clients, either for other encoding uses or for regular decode and display.

Surface Compositor

Surface Compositor provides a client/server interface to composite multiple surfaces into a single framebuffer. Because protected clients cannot access the display handle, some server-side module must provide the service.

Reasons you may want to use SurfaceCompositor are:

- You are using a multi-process capable GUI like DirectFB, but also want to integrate non-DFB clients.
- You want a high performance graphics solution. You may be using application-based IPC to communicate to a main graphics renderer. But that will be slow because it requires a context switch. When Nexus is compiled for kernel mode, SurfaceCompositor runs in the kernel. Calls from every client only require a user->kernel mode switch, which is very fast[5].

See nexus/examples/multiprocess/blit_server and blit_client for an example. See nexus/docs/Nexus_SurfaceCompositor.pdf for more documentation.

Input Router

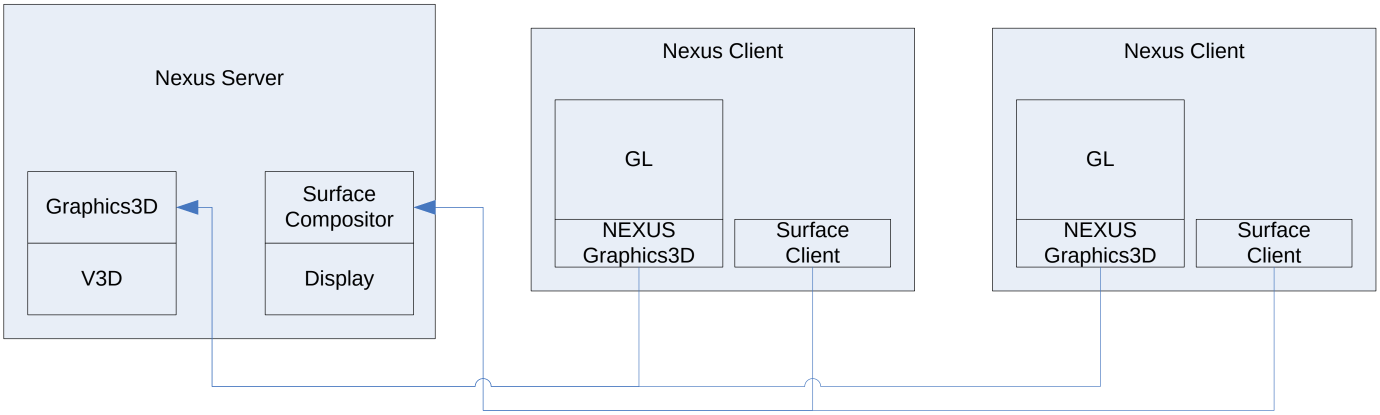
InputRouter is a simple client/server library for routing user input (including keyboard, mouse, IR, UHF, keypad) to various clients.

See nexus/examples/multiprocess/input_router for an example.

OpenGL-ES 3D graphics

OpenGL-ES 2.0 has multiprocess support. Each client creates its own instance of GL. Internally, GL communicates to the Nexus server using the NEXUS_Graphics3D API. The GL reference applications include examples of integrating GL with SurfaceCompositor.

[5] If Nexus is compiled for user mode, SurfaceCompositor will run in the server's user mode thread. Performance will not be as good, but it is supported.


Nexus Server
Graphics3D
V3D
Surface Compositor
Display
Nexus Client
GL
NEXUS Graphics3D
Surface Client
Nexus Client
GL
NEXUS Graphics3D
Surface Client

Non-Root Access

Clients

Nexus clients can run as non-root users. In kernel mode, the server application creates device nodes (/dev/nexus_*) and sets permissions to 0666, which gives read/write access to all users. In user mode, the server application binds Unix domains sockets (/tmp/nexus_*) and sets the same 0666 permissions. This allows Nexus clients to connect as any user id.

The Nexus driver (either nexus.ko or bcm_driver.ko) allows the client to map device memory within server-specified bounds.

The system operator can set different user/group permissions on those device nodes or socket files. Those permissions operate at a module level and are enforced by the operating system. They work in addition to the NEXUS_ClientMode security, which is function based, not module based.

In kernel mode, the device nodes can be created with desired permissions before the server is run; Nexus will not recreate the device nodes. In user mode, Nexus must recreate the socket files, so desired permissions must be set after the server has started. The device nodes and socket files vary by what modules are compiled into your driver. Run Nexus once to discover those nodes.

Dropping privilege in the server

The Nexus server application (or single process application) must start as root in order to map memory and connect to interrupts. This is enforced in the driver (either nexus.ko or bcm_driver.ko). However, to give an additional level of security, Nexus can drop privilege after performing those initial operations. See NEXUS_PlatformSettings.permissions.userId and .groupId.

However, even if the Nexus server is not running with operating system root privilege, it still has direct access to registers and other unbounded hardware access. So the server application must still be trusted.

Linux Containers (LXC)

Nexus multi-process runs in an LXC environment in both user mode and kernel mode configurations.

The only difficulty is the location of the Unix domain socket files for user mode. Which means this applies to Nexus built in user mode or for NxClient running on top of Nexus in either kernel or user modes.

The default location for Nexus and NxClient Unix domain sockets is /tmp, but this directory is not the same for each LXC client. Nexus has a run-time environment variable to redirect the socket files to a shared location, as follows:

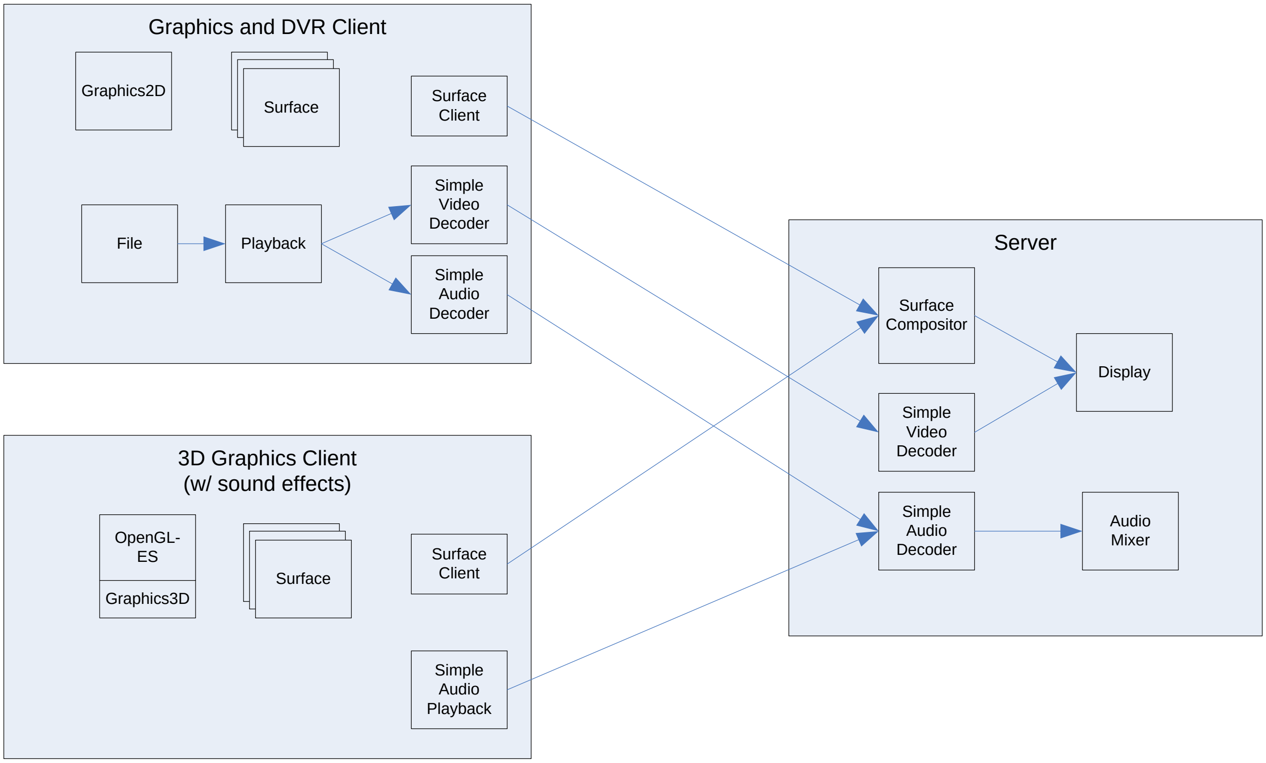
```
    export nexus_ipc_dir=/shared
```

This must be set in the server and client environments.

Typical Systems

The following diagrams show how Nexus interfaces are separated between client and server control in typical usage scenarios.

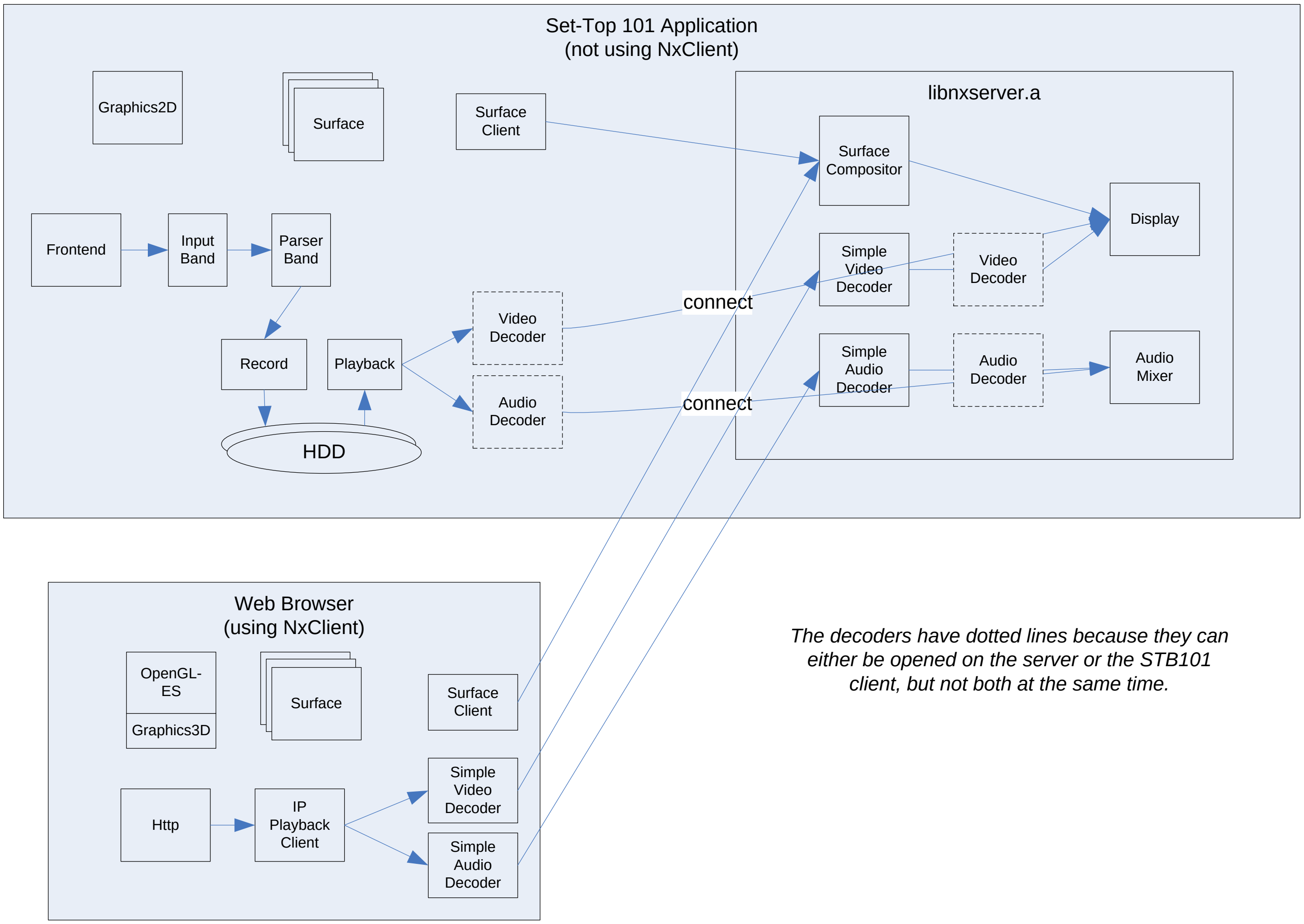
Two Graphics Clients



This shows a combination of DVR and non-DVR clients. Each client renders graphics independently and submits them to the surface compositor via the surface client interface. Audio mixing is supported in the SimpleAudioDecoder via the client's SimpleAudioPlayback interface.

Legacy Set-Top Application and new web browser client

This diagram shows a “Set-Top 101” main application. The main app is usually mature and already deployed in the field. The new web browser must be added with minimal impact on the main application.



The decoders have dotted lines because they can either be opened on the server or the STB101 client, but not both at the same time.

This mode is supported through NxClient and its “External Application” mode. See nexus/nxclient/docs/NxClient.pdf for details.

Writing your Multi-process Application

Set your client mode at the beginning

We highly recommend that you select your client mode at the beginning of your design and program it at the beginning of development. Foreseeing all of the implications of a multi-process design is very difficult, so by setting the correct mode you will learn many of those implications as you attempt to run. See the “Concepts” section for a full discussion of this.

By default, all Nexus clients are protected. The typical expectation is that client applications can freely crash and not compromise the server. Because the API is limited and all handles are verified, you need to factor this into your system design from day one. Common issues you may encounter:

- You cannot share handles between applications using shared memory. The untrusted client cannot use a handle it has not opened or acquired itself.
- You should not access the display from an untrusted client. That should be managed by the server application and/or a client/server graphics library like SurfaceCompositor.
- Live decode (tuning, input bands and parser bands) is supported from a protected client, but not an untrusted client.

Test crash scenarios early and often

When writing your applications, you need to test abnormal termination right from the beginning. This can be easily done by hitting Ctrl-C from the console at random places. You can also test using random numbers as parameters or handles. The client will crash, but the server should be uncompromised.

Do not leave crash testing to the later phases of development. If this capability is not baked into the system, it is very hard to add it late in development.

Expect Application IPC

Nexus provides an IPC mechanism for its API. But this is rarely the only IPC needed in the system. You will likely need some application-level coordination above Nexus. This can be done using sockets, shared memory, message queues, etc.