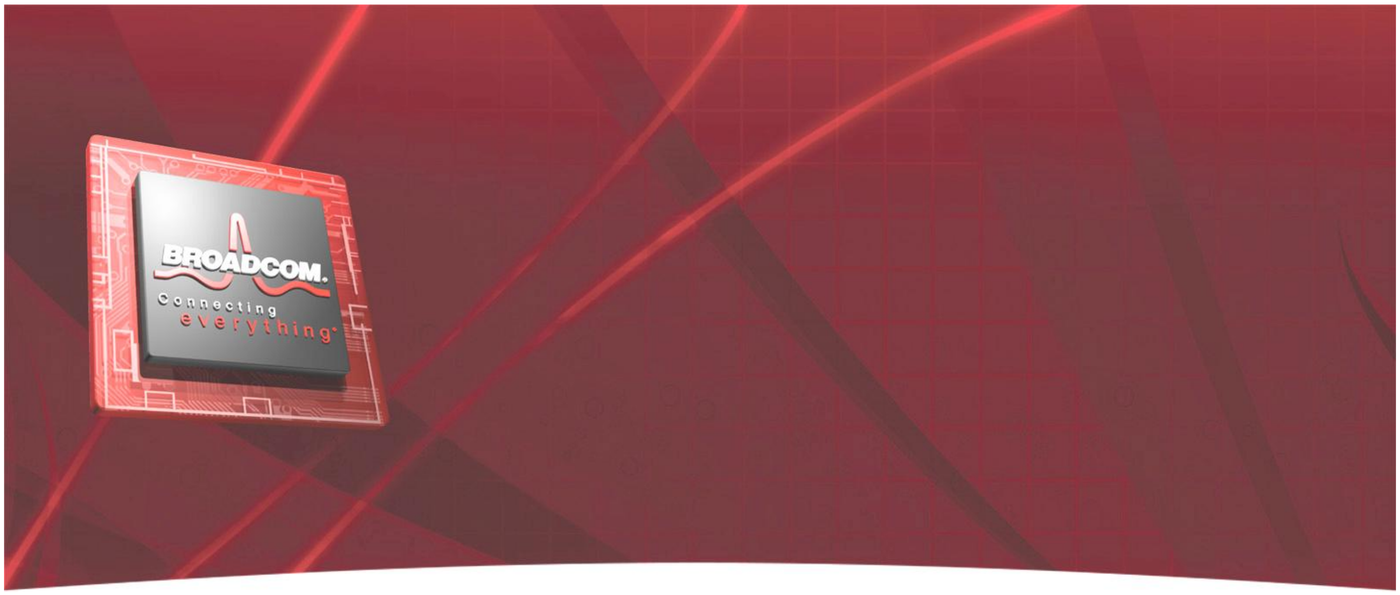


Nexus

Multi-Process Usage

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

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 and can run in an unprotected mode. The only code that is untrusted is running in client processes.

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

The IPC mechanism is an internal, implementation detail which is documented in Nexus_MultiProcessDesign.doc. This document describes the policies Nexus establishes to secure the system. These policies must be understood by the Nexus user.

Nexus allows clients to operate in three modes: unprotected, protected and untrusted. We recommend that you run in either protected or untrusted mode. Running in unprotected mode will work – if the client never crashes and is always well-behaved. But one of the unspoken requirements of multi-process architecture is that the system can survive misbehaving or malicious clients.

We also 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 unprotected 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.

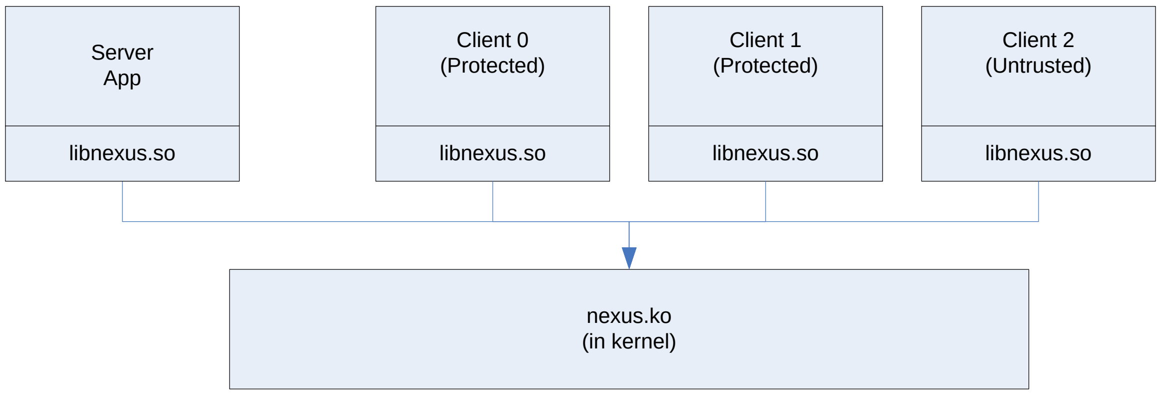
Overview of Client Modes

The three client modes can be summarized as follows:

Feature	Unprotected	Protected	Untrusted
Nexus API	Full access	Full access	Limited access
Handle tracking	Yes	Yes	Yes
Handle verification	No	Yes	Yes
Garbage collection	Limited	Yes	Yes
Client can crash?	No	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

The client does not choose its mode. The server determines what mode each client is in. This can be done explicitly for authenticated clients or implicitly for unauthenticated clients (authentication is described below).

The following diagram shows a server app and multiple clients (in different modes) in a kernel mode, multi-process system:



Handle Tracking and Handle Verification

Handle tracking means that when any process opens a handle, Nexus remembers the handle, the handle type and the client that opened the handle. When the client terminates, any tracked handle will be automatically closed. Handle tracking is unconditionally enabled for all client modes[1].

Handle verification means that Nexus checks the parameters of every function being made and verifies that the handle is a known value, with the correct type, and is being used by the correct client. If verification fails, the function returns a failure immediately. Handle verification is only enabled for protected and untrusted mode.

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.

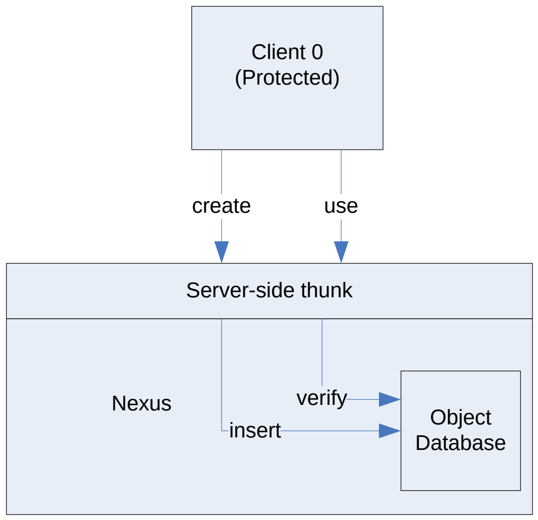
These two techniques are implemented in the server-side thunk, so they are guaranteed across the entire Nexus API in the same way that module synchronization is guaranteed[2]. Handle tracking and verification enable the server to protect itself in a multi-process environment.

[1] Nexus handle tracking is similar to how an operating system manages user mode processes. When a process dies, all resources that it has are automatically freed.

[2] See nexus/docs/Nexus_Architecture.pdf for the original discussing of auto-generated thunking code.

Protected Mode
Protected mode implements handle tracking and verification across the entire Nexus API. It is the default mode of clients.

The main implication for applications is that only handles that have been opened, created or acquired by the client through Nexus can be used by the client.



In protected mode, the server is protecting its software state from an ill-behaved client. It does not protect the client from itself. For instance, if a client creates multiple threads which try to simultaneously use the same handle, bad things will happen, but the server will not crash.

Untrusted Mode
Untrusted clients have all the same safe guards as protected clients, but we also limit the API for an extra level of safety.

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.

The limited access is done by means of an access control list (ACL). The untrusted client ACL is defined in nexus/build/tools/common/nexus_untrusted_api.txt. It is enforced by the server thunk.

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.

Unprotected Mode

In unprotected mode, handle tracking is enabled, but handle verification is not enabled.

Garbage collection is limited because the client may have obtained resources through non-tracked means. In those cases, there is no way for Nexus to clean up those resources when the client dies. For instance, if an unprotected client gets a decoder handle from shared memory and starts decode, when the client dies, decode will continue going.

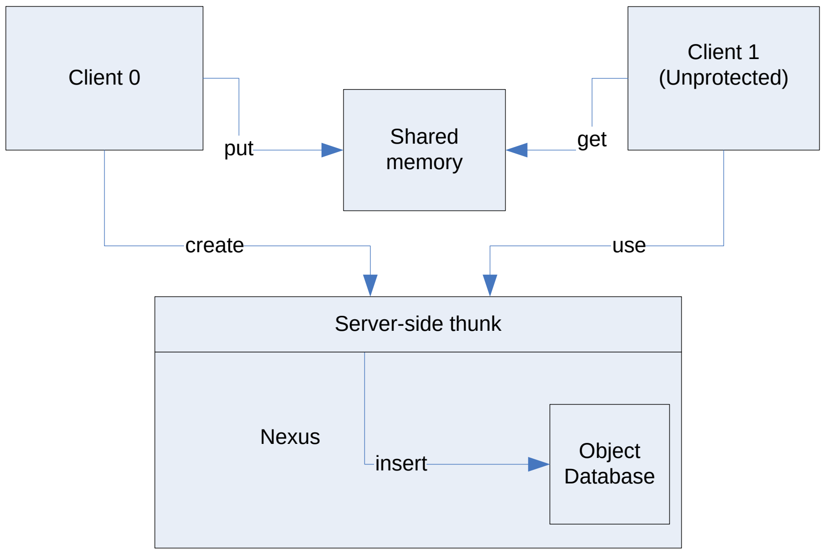
We do not recommend that any client be run in unprotected mode or with root privileges. Clients are expected to crash, but Nexus cannot guarantee recovery from an unprotected client crash. We do support clients in unprotected mode and will attempt to do resource cleanup, but it is limited what can be done.

The server runs in unprotected mode and must run with root privileges. There is no reason for the server to run in a protected mode. In protected mode, any violation will be caught by Nexus and the call will be rejected. When a call is rejected, it usually results in a client crash. But the server is not a client and by definition it cannot crash. Therefore there is no reason to protect the server from itself. Server applications must be written correctly and not make invalid calls to Nexus.

Handle Caches

One common approach used for multi-process system design is to create a handle cache. The idea is that handles are opened and put into the cache (or the cache is application-specific and actually opens the handles itself). Then, clients in various processes can check out handles and use them. The client will then check the handles back in. If the client crashes the handles will be automatically checked back in by the handle cache.

This approach is not supported by Nexus' protected or untrusted modes. If you must use the handle cache approach, your client must run in unprotected mode. We do not recommend unprotected mode clients, but it can be done.



In unprotected mode, Nexus cannot protect against the following attacks:

- Retrieve a handle from the cache and call Nexus to close it
- Retrieve a handle from the cache, but call Nexus with the wrong handle
- Retrieve a handle from the cache and never return it

One common scenario is sharing graphics surfaces between clients. Sharing a single surface handle between clients is not safe because one client may close the handle and another client may attempt to use it. Instead, we recommend that the clients share the offset/size information about the surface, then each client creates its own surface which aliases the same heap memory. The application is responsible to manage the life cycle of the heap memory. It should only close the surface that did the allocation after all aliasing surfaces are closed.

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. An authenticated client is one that is identified by the server and can be unprotected, protected or untrusted. An unauthenticated client is one that is not identified by the server. It can also be unprotected, protected or untrusted. We highly recommend that unauthenticated clients only be set to untrusted mode.

Authentication is set up on the server using NEXUS_Platform_RegisterClient.

Unauthenticated client options and defaults are set on the server using NEXUS_PlatformStartServerSettings.

Nexus Interface Multi-Process Capabilities

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

Interface	Capability	New API for multi-process systems
AudioDecoder	Exclusive and statically-allocated	SimpleAudioDecoder wrapper
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 statically-allocated	NEXUS_Platform_AcquireFrontend
Graphics2D	Virtualized	Unchanged
Graphics3D	Virtualized using OpenGL-ES	Unchanged
HdmiInput	Exclusive and statically-allocated	Unchanged
HdmiOutput	Exclusive and statically-allocated	Unchanged
I2c	Exclusive and statically-allocated	NEXUS_Platform_AcquireI2c
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
StcChannel	Exclusive and dynamically-allocated	NEXUS_ANY_ID support added
Surface	Local	Unchanged
Timebase	Exclusive and dynamically-allocated	NEXUS_Timebase_Open
VideoDecoder	Exclusive and statically-allocated	SimpleVideoDecoder wrapper

Multi-process Capable Interfaces

This section defines the capability types listed in the previous section.

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.[3]

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[4].

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, even in unprotected mode apps.

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[5], Recpump and Playpump[6].

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.

---

[3] 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.

[4] PictureDecoder virtualization has been recently added. It requires a small bit of special code in the application until the underlying SID FW can be re-designed for multi-context support.

[5] 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 in unprotected clients.

[6] For some chips, we only have 4 HW playback channels available, but this is still sufficient for most systems. For newer silicon, many more playback channels are available.

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.

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. For example, a Frontend handle can be acquired by a client using NEXUS_Platform_AcquireFrontend, but the client must know which frontend it wants to acquire. Once it has acquired it, it can use it exclusively until it is released.

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.

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

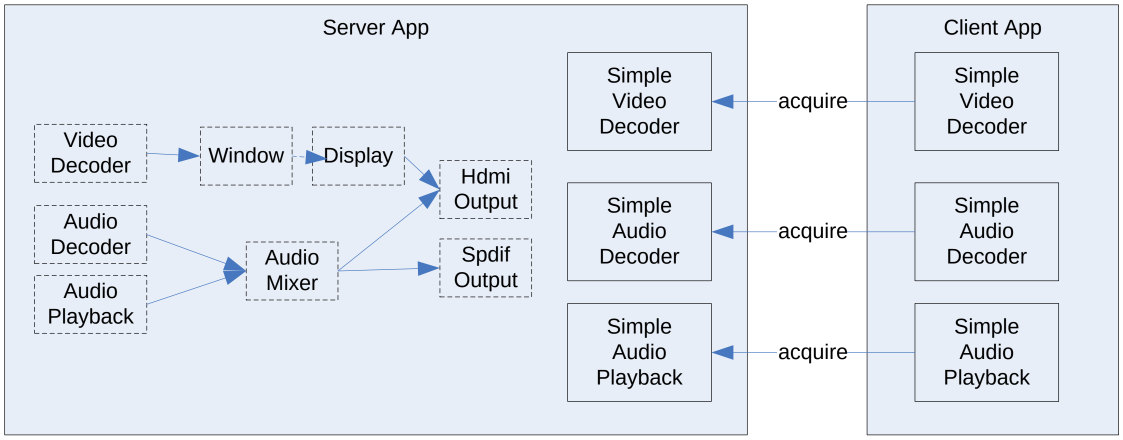
User input devices like IrInput or UhfInput must be managed by the many server application and routed to the appropriate client based on “focus”.

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.



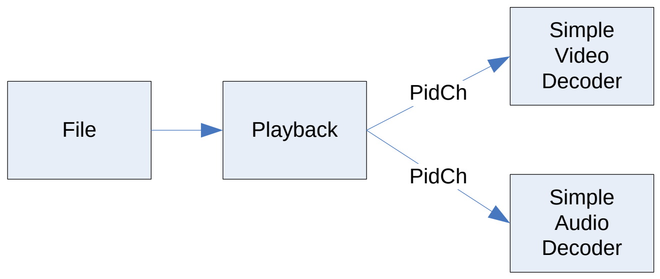
If you do not use the simple decoder, there are two other options, but both have significant complexity or risk. First, the client can open the decoder before each use and close it after each use[7]. The application will need to provide some application IPC to an unprotected server so that a connection from the VideoDecoder can be made to the VideoWindow. Second, the client can run in unprotected mode and get the decoder from a handle cache (but then the server is not protected.)

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.

[7] 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.

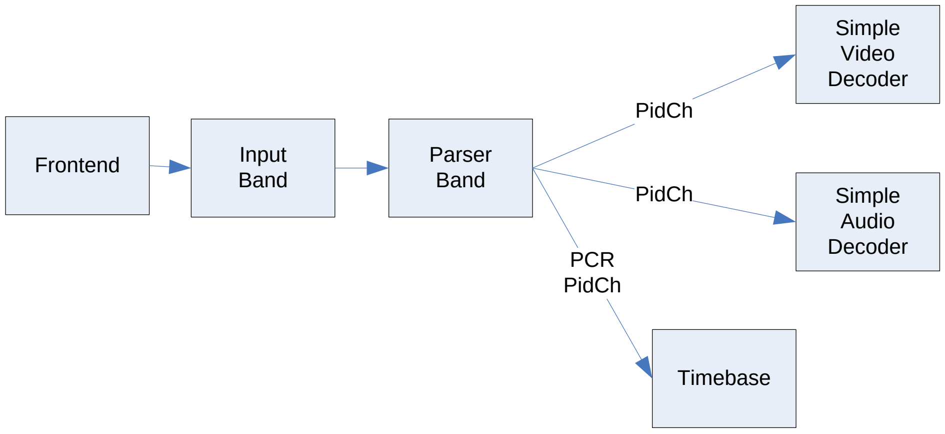


These simple decoder interfaces have the following advantages:

- 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.

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



ParserBand and Timebase have been given new Open/Close functions.[8]

Frontend also has new Acquire/Release function made available through the Platform. See NEXUS_Platform_AcquireFrontend.

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.

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

There is a separate document giving a detailed explanation of the SurfaceCompositor module.

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

[8] 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.

kernel. Calls from every client only require a user->kernel mode switch, which is very fast[9].

See nexus/examples/multiprocess/blit_server and blit_client for an example.

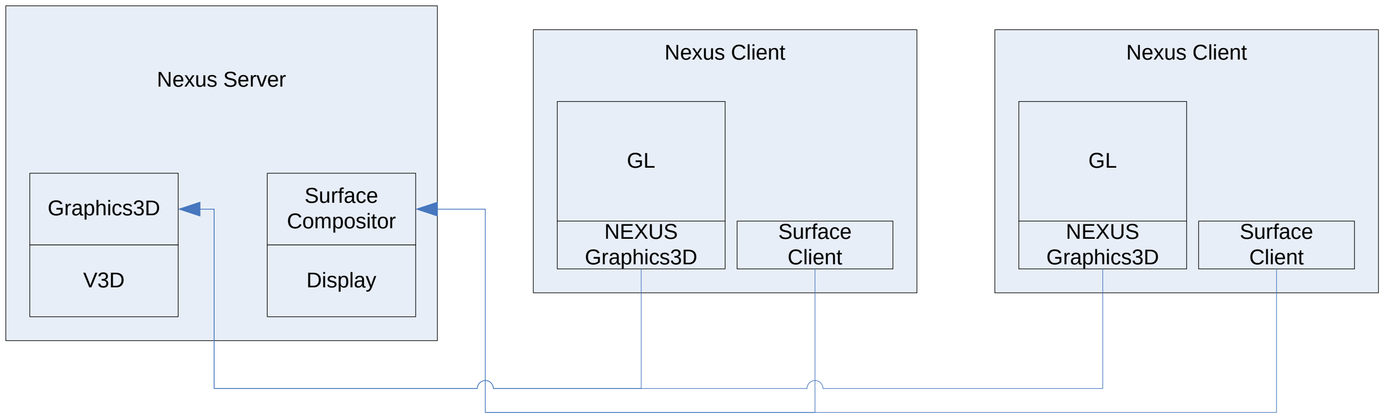
Input Router

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.c 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.

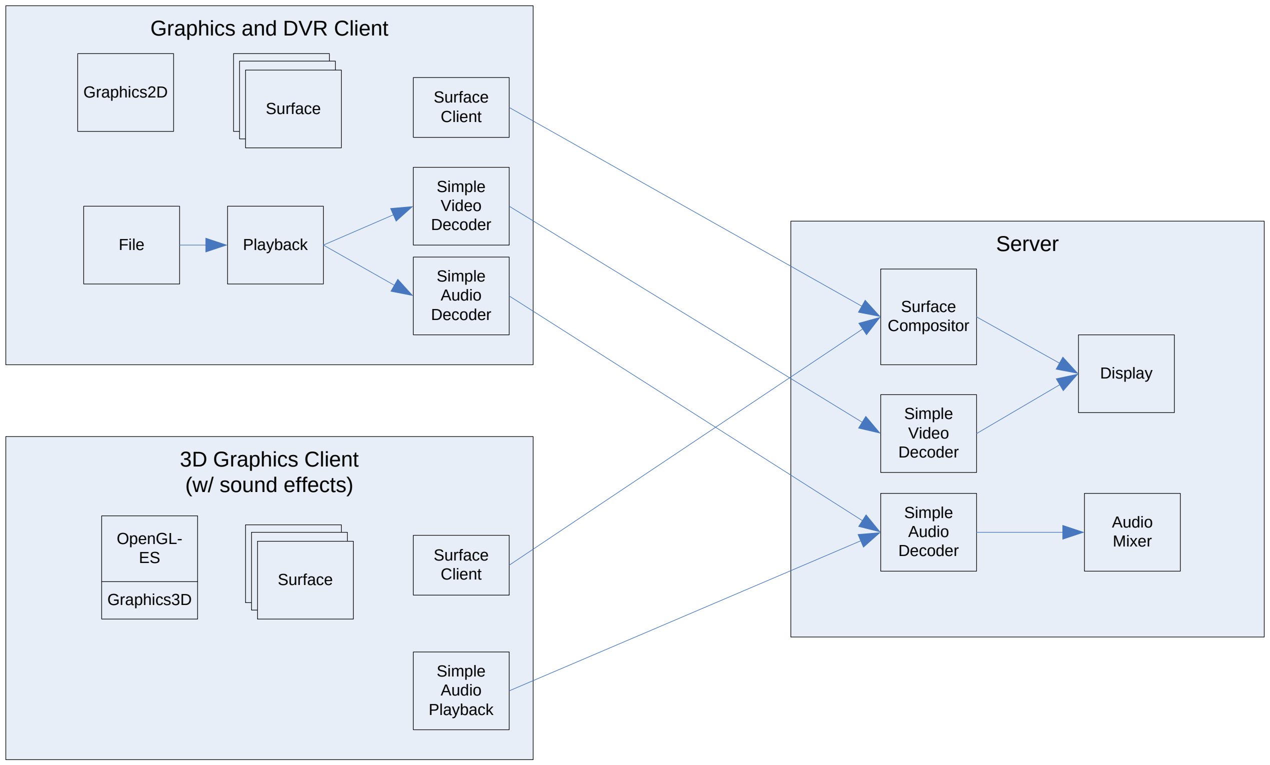


[9] If Nexus is compiled for user mode, SurfaceCompositor will run in the server's user mode thread. Performance will not be as good, but it is supported.

Typical Systems

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

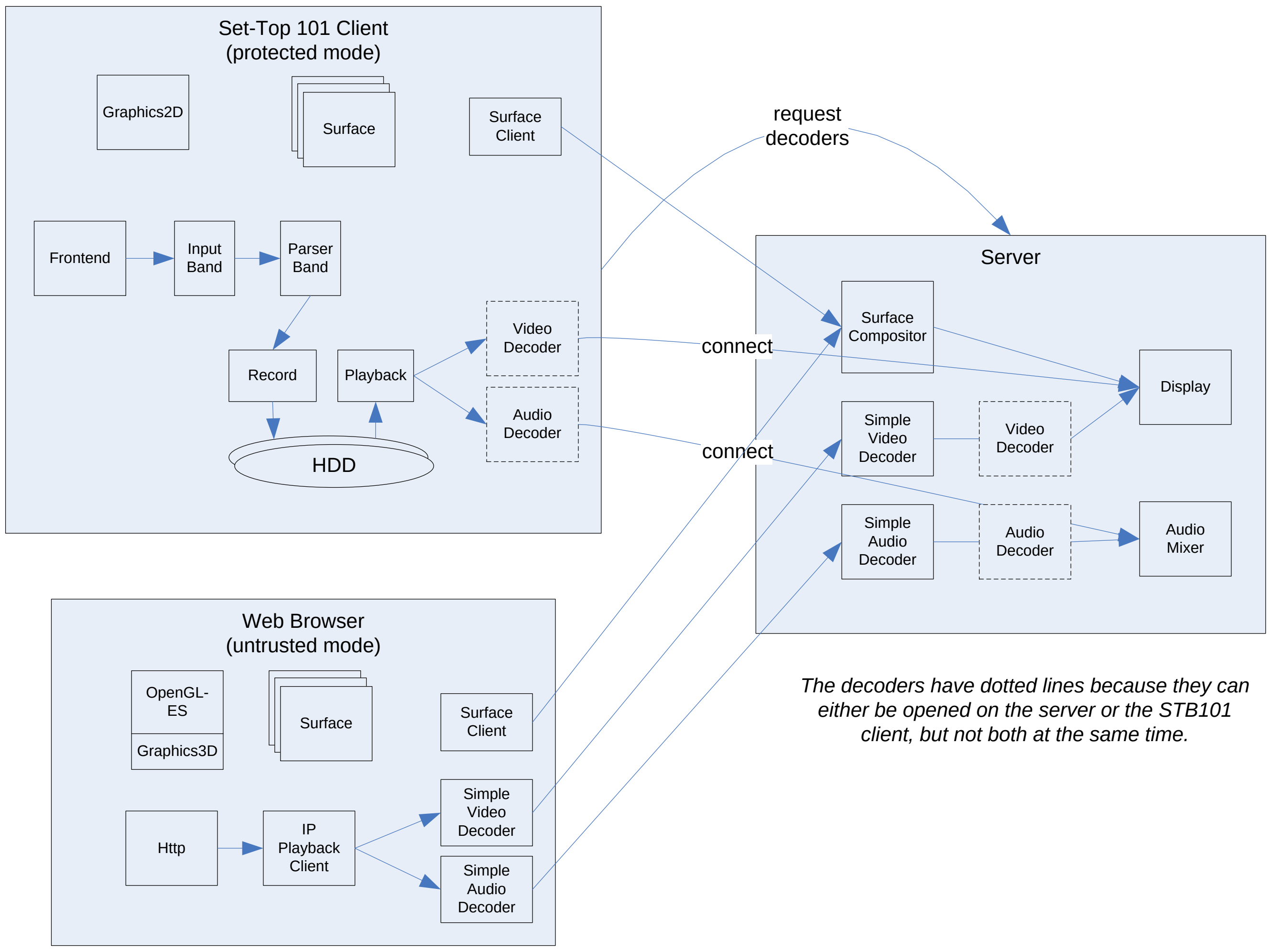
Two Graphics Clients



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.

Sharing the regular decoder between the Set-Top 101 client (STB101) and the server app requires the following steps:

- The server opens the regular video decoder and simple video decoder. It places the regular video decoder into the simple decoder as a container. This allows the client to acquire and decode.
- When STB101 wants to start a regular decoder, it sends an app IPC request to the server app asking for the regular decoder.
    - If it tried to open it early, it would simply fail.

- The server removes the regular decoder from the simple decoder and closes it. It tells the STB101 client that the decoder is available.
    - Depending on the NEXUS_SimpleDecoderDisableMode, the web browser client may know nothing about the change.
- STB101 opens the decoder.
- It calls an app IPC request to connect the decoder to a video window.
    - STB101 is unable to make the connection because it does not own the display, and therefore does not own the window handle.
- When STB101 is done, it closes the video decoder and sends an app IPC function to notify the server that it is done.
- The server can open the regular decoder and repopulate the simple decoder.

There are many other variations possible:

- The Set-Top 101 app may be inside the server (not a separate client). This would remove the need for app IPC to make the resource request and connection.
- The Set-Top 101 client may be unprotected. This would require app IPC to make the resource request, but no app IPC to make the connection.

Application IPC

While Nexus supplies an IPC mechanism and security policy for its interfaces, most multiprocess systems require their own application-specific communication. This is done above Nexus and can be accomplished by a variety of means including sockets, shared memory, message queues, or custom drivers.

Typically usage modes of application IPC are:

- Dynamically creating clients
- Allowing SurfaceCompositor clients to position themselves on the screen
- Allowing clients to change the display format or outputs
- Connecting a regular video decoder to a window

bipc

In order to facilitate application IPC, we have created a simple IPC mechanism called bipc (pronounced "bee i-p-c"). bipc reuses concepts from Nexus and some Perl header file parsing from Nexus, but does not actually depend on Nexus.

Typical usage is as follows:

- The application designer specifies a header file which follows a limited coding convention, similar to the Nexus coding convention.
- The header file is given to a parser which builds client-side and server-side IPC.
- The client opens a connection and the server receives it.
- The client makes a call and it results in a call on the server.

See nexus/lib/ipc for the implementation and coding convention. The nexus/examples/multiprocess/refsw_server application uses this application IPC.

bipc coding convention

- The API must be structured as an interface: there is an opaque handle. There is a constructor suffixed by _open or_create which returns this handle. Every other function in the interface takes that handle as its first param. There is a destructor suffixed with _close or _destroy which returns void and has the handle as its only param.
- All other params can only be primitive data types or structs, passed by value or by reference.
- Pass-by-value params and const pass-by-reference params are input. Non-const pass-by-reference params are output. There are no input/output params.
- No support for callbacks. If a client/server need full duplex communication, the client must implement its own server-side bipc mechanism.

- The interface handle is not global. The server and the client will have different handle values for the same connection. This means that bipc interface handles should not be passed between client and server.
- No variable length array support.
- We do not recommend passing Nexus handles over bipc. The reason is that there is no way for the server is know that the handle has not been closed by the client. The one exception we allow is passing NEXUS_VideoInput/NEXUS_AudioInput abstract connectors.[10]

Implementing

We recommend you study nexus/examples/multiprocess/refsw_server.c and copy its techniques and code. There is no generic implementation of the listener thread because of the synchronization complexity. Each server app must write its own thread and manage its own state.

If client and server code are compiled into a single application, the bipc mechanism disappears and the client → server call is simply a direction function call. Being able to seamlessly remove the IPC mechanism makes for simple system testing.

[10] See Nexus_MultiProcessDesign.doc for a discussion of this vulnerability.

Example Applications

Ever nexus release comes with multi-process example applications which are useful for learning nexus' multi-process capabilities. You can find them in nexus/examples/multiprocess.

Typical application pairs are:

Server App	Client Apps	Description
decode_server	decode_client live_client audio_client	Audio/video decode using DVR or live with the Simple Decoders. No graphics.
mosaic_decode_server	decode_client	Video decode using mosaic decoders instead of regular video decoders. The client application does not change because the difference is abstracted by the simple decoder wrapper.
blit_server	blit_client tunneled_client animation_client picture_decoder_client	Client renders graphics to an offscreen surface, then the server blits from that surface to the framebuffer. No decode.
refsw_server	All clients run by decode_server and blit_server, plus smooth_pig and cmd_client	Reference server application. This is a complex server app that supports decode, graphics and an example of application IPC for dynamic client creation and positioning.

Note that smooth_pig and cmd_client don't work with decode_server or blit_server because the apps require both SurfaceCompositor and SimpleDecoder.

You can build the applications as follows:

```
cd nexus/examples/multiprocess
make
```

For linux user mode, the examples Makefile will do a two-pass build. For the server, it will compile with no NEXUS_MODE. For the client, it will compile with NEXUS_MODE=client. Linux kernel mode is a one-pass build with NEXUS_MODE=proxy[11]. You can also build manually for user mode as follows:

```
unset NEXUS_MODE
make decode_server
export NEXUS_MODE=client
make decode_client
```

You can run the application pairs on a set-top box using a console and a telnet session. The execution is the same for Linux kernel and user modes.

First, start the server from the console. For instance:

```
nexus decode_server
```

Then, telnet into the set-top and start the client application:

```
nexus.client decode_client
```

You can stop and restart the client app multiple times without restarting the server. You can also kill the client with Ctrl-C and restart it. The server-side handle verification and clean-up can be reviewed by running with export msg_modules=nexus_driver_objects (or export config=msg_modules=nexus_driver_objects for kernel mode).

The nexus platform code has two special header files for multi-process support:

- nexus_platform_server.h – API's for starting the IPC server and managing client resources
- nexus_platform_client.h – API's for connecting to the IPC server and inquiring about client resources

All other multi-process API's are simply part of the regular Nexus API.

[11] All kernel mode applications link to libnexus.so, whether they are server or client apps. User mode server applications must link to libnexus.so and client applications to libnexus_client.so.

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.

Running unprotected clients

If you must run an unprotected client, you should be aware that the unprotected client cannot crash. We have found that the requirement that clients can crash is typically assumed and not explicit.

You can enable an unprotected client in two ways:

- set NEXUS_PlatformStartServerSettings.unauthenticatedClientMode=eUnprotected when calling NEXUS_Platform_StartServer
- set NEXUS_ClientSettings.configuration.mode=eUnprotected when calling NEXUS_Platform_RegisterClient

If an unprotected client terminates abnormally, you will see an error on the server's console and the server will terminate itself immediately. Your client application should be debugged.[12]

[12] You can set NEXUS_PlatformStartServerSettings.allowUnprotectedClientsToCrash to avoid the server termination, but this mode is undefined and not supported.