Nexus

Architecture

Broadcom
Corporate Headquarters: San Jose, CA

Web: www.broadcom.com

Revision History

Revision	Date	Change Description
STB_Nexus-SWUM00-R	03/28/2008	Initial version
STB_Nexus-SWUM101-R	04/25/2008	
STB_Nexus-SWUM102-R	09/8/2008	
STB_Nexus-SWUM103-R	04/24/09	
2.0	4/24/13	
2.1	9/30/14	Revise overview diagrams Extend section on Thunking
2.2	1/12/16	Update “Application Callback Synchronization”

Table of Contents

Introduction

Nexus is a high-level, modular API for Broadcom cable, satellite, and IP set-top boxes. The goal of Nexus is to allow Broadcom's customers to ship their products in less time and with less effort. This document defines the software architecture for Nexus.

Nexus provides easy-to-use and thread-safe interfaces that have been designed for the purpose of porting customer applications and Hardware Abstraction Layers (HALs) to Broadcom silicon. Nexus is modular because all code lives within well-defined modules that can only interact through well-defined APIs. Nexus uses some component-based concepts found in systems like DCOM or CORBA, but Nexus is still lightweight and simple.

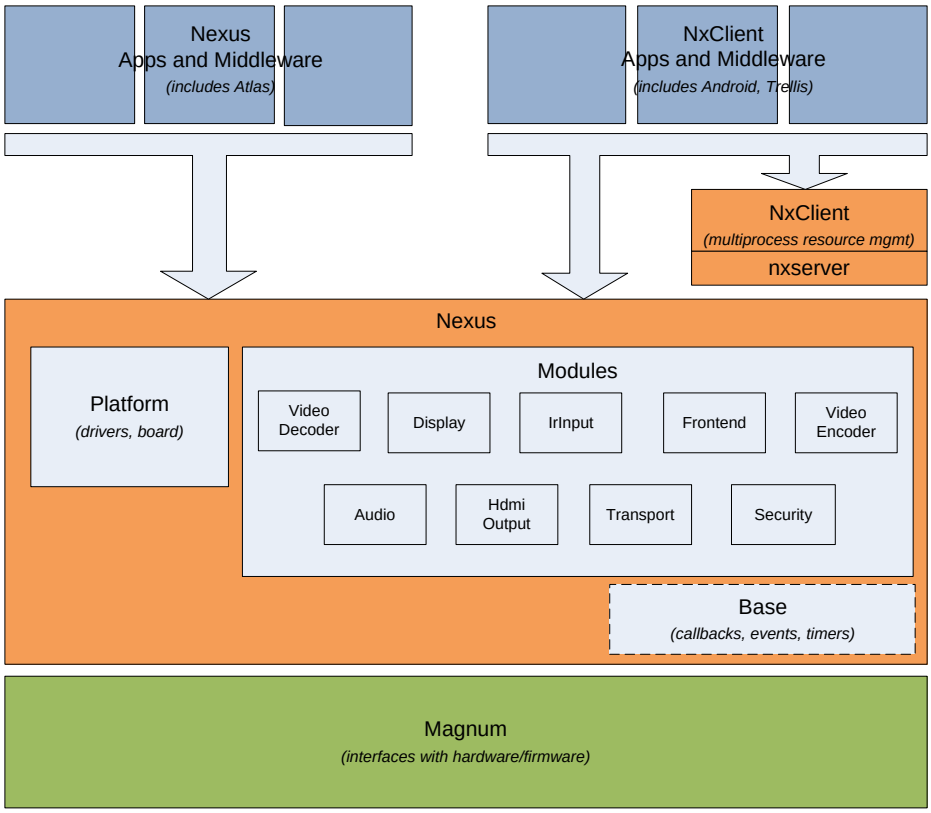
Nexus has a full-featured and robust synchronization model that makes it possible to build a hierarchy of modules, in which the modules have a network of complex calling and callback relationships, yet still avoid deadlock and provide maximum efficiency.

Nexus Overview

Set-Top Software Stack

Figure 1 shows how Nexus fits into Broadcom's overall software architecture for cable, satellite and IP set-top boxes.

Figure 1: Nexus and the Set-top Box Software Architecture



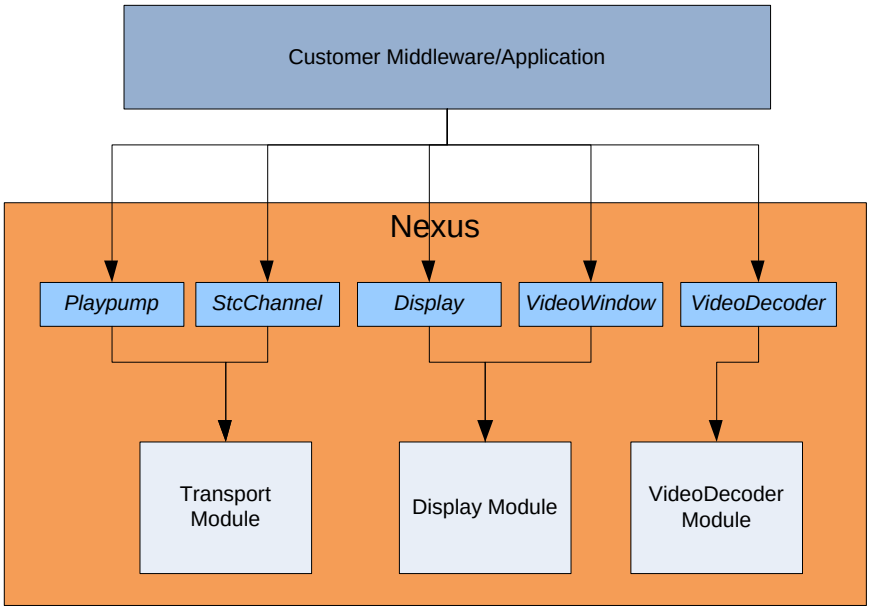
The components of the overall software architecture are:

- **Magnum**: Low-level software layer that directly interacts with hardware and firmware and provides internal interfaces to Nexus.
- **Nexus:** High-level software layer that provides easy-to-use interfaces for middleware and applications.
- **NxClient:** Multi-process library and re-usable server for resource management and system control. See nexus/nxclient/docs/NxClient.pdf for more information.
- **Reference Applications:** A variety of reference applications built on top of Nexus and NxClient which demonstrate and test system functionality.

Nexus Architecture

Nexus is made up of interfaces and modules. A module presents one or more interfaces. Each interface calls into exactly one module. Nexus interfaces provide an abstraction on the module implementation. The same system can be seen from two perspectives: an application perspective and implementation perspective.

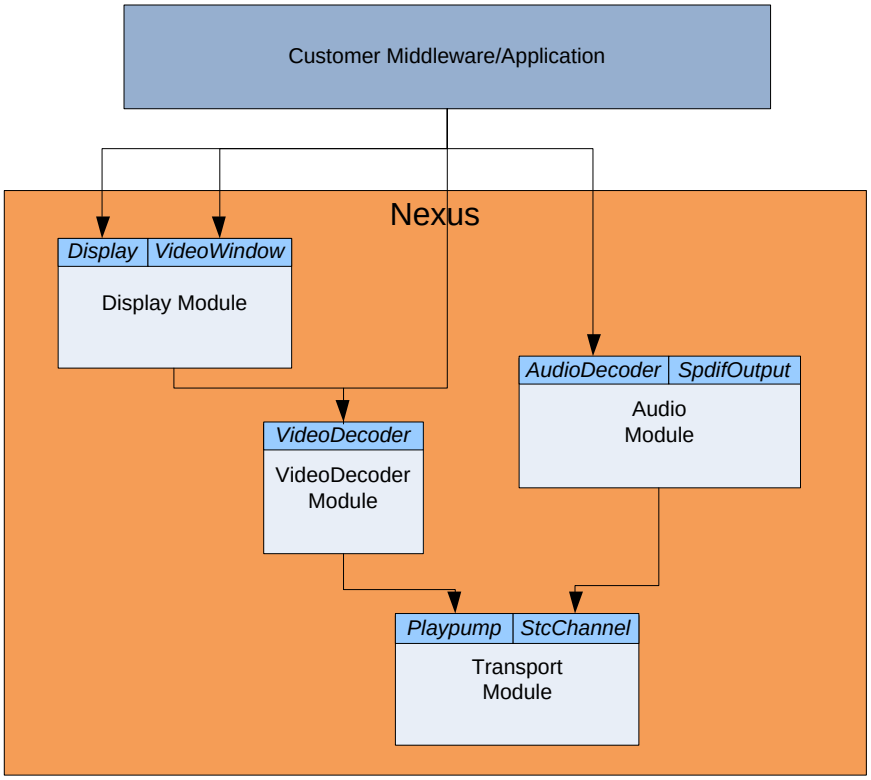
Figure 2: Nexus Interfaces from an Application Perspective



From an application perspective, Nexus is a collection of interconnected interfaces. Each interface is a set of function signatures (that is, prototypes) along with their structures and enums that present a feature. Nexus interfaces are mapped to internal modules, so how Nexus services interrupts, internal callbacks, and hardware and firmware state is abstracted from the application.

The following is the same system from an implementation perspective:

Figure 3: Nexus Interfaces from an Implementation Perspective



From this view, Nexus is a collection of interconnected modules. Each module has a common state and synchronization. Modules can interact with each other in a hierarchical arrangement. A module can be called by an application or by a higher level module.

Example Code

A good way to begin understanding Nexus is to view a simple Nexus application, as illustrated by the following code fragment (error handling omitted for clarity):

```
main() {
    // open Interfaces
    videoDecoder = NEXUS_VideoDecoder_Open(0, NULL);
    display = NEXUS_Display_Open(0, NULL);
    window = NEXUS_VideoWindow_Open(display, 0);
    component = NEXUS_ComponentOutput_Open(0, NULL);

    // make connections
    NEXUS_Display_AddOutput(display, NEXUS_ComponentOutput_GetConnector(component));
    NEXUS_VideoWindow_AddInput(window,NEXUS_VideoDecoder_GetConnector(videoDecoder));
    startSettings.pidChannel = NEXUS_PidChannel_Open(NEXUS_ParserBand_e0, 0x21, NULL);

    // start video decode
    NEXUS_VideoDecoder_Start(videoDecoder, &startSettings);
}
```

The application builds up the system by opening a variety of Nexus interfaces. Each Open call returns a handle. The application then connects the various interfaces together. After the system is connected, the application can start operation.

The following code is an example of an Interface API:

```
/* opaque handle */
typedef struct NEXUS_Display *NEXUS_DisplayHandle;

typedef struct NEXUS_DisplaySettings
{
    int param1;
    int param2;
} NEXUS_DisplaySettings;

NEXUS_DisplayHandle NEXUS_Display_Open(
    unsigned index,
    const NEXUS_DisplaySettings *pSettings
    );

void NEXUS_Display_Close(
    NEXUS_DisplayHandle handle
    ;

void NEXUS_Display_GetSettings(
    NEXUS_DisplayHandle handle,
    NEXUS_DisplaySettings *pSettings /* [out] */
    );

NEXUS_Error NEXUS_Display_SetSettings(
```

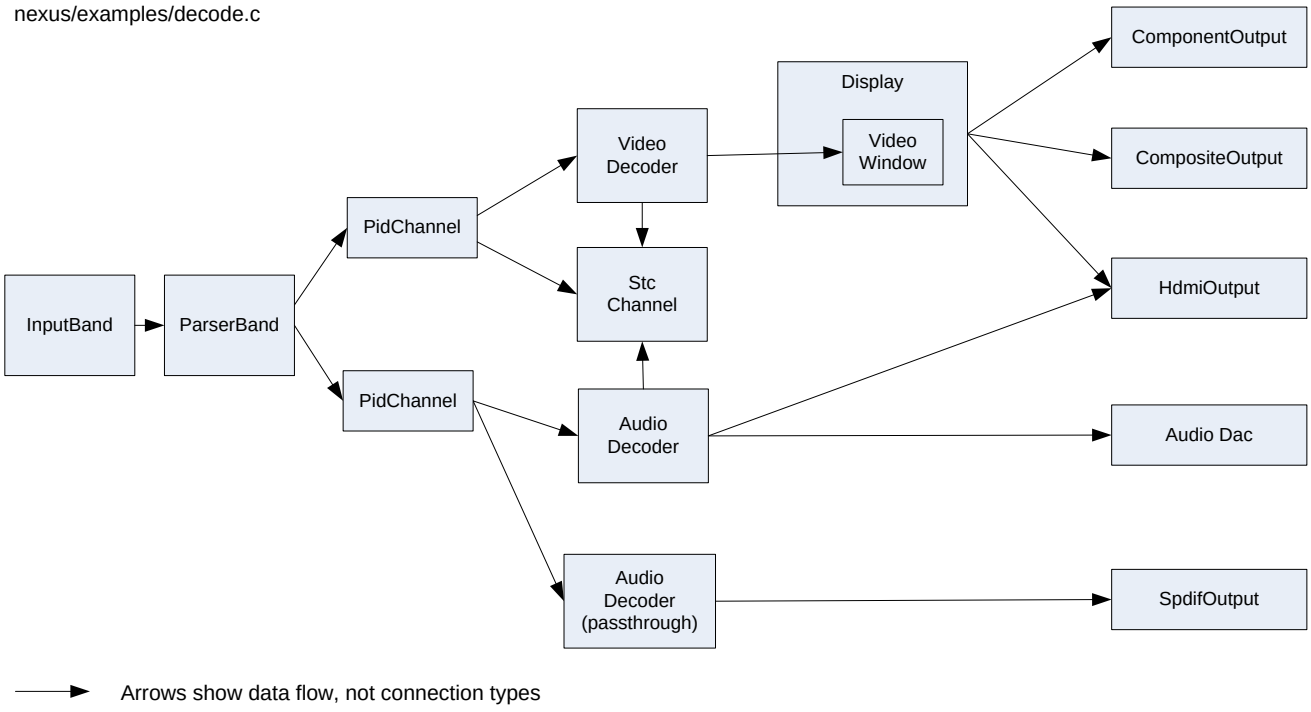
```
NEXUS_DisplayHandle handle,
const NEXUS_DisplaySettings *pSettings
);
```

For working examples, see nexus/examples. For actual interfaces, see nexus/modules/<MODULE>/include.

Example Diagrams

A simple Nexus application can usually be illustrated with a data flow or filter graph diagram like the following. Each interface corresponds to one box. The connections between the interfaces correspond to the data flow. Each interface has settings and status which can be used to customize behavior. After the filter graph has been constructed and configured, the application can start decode and all processing happens automatically and internally.

Figure 4: Audio/Video Decode and Display



For details on how specific interfaces actually work, see Nexus_Usage.pdf.

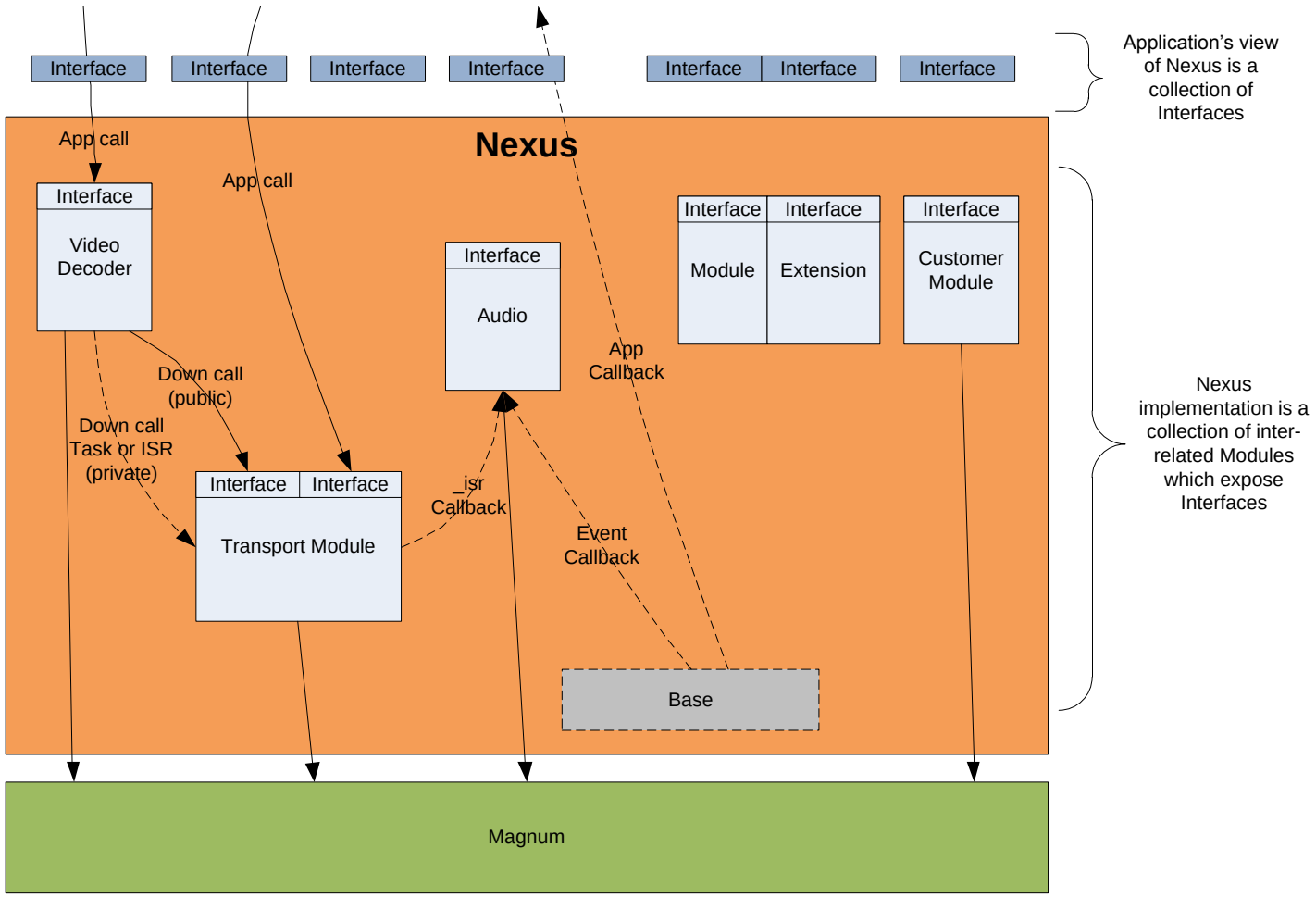
Nexus Modules

Overview

A Nexus module is a collection of code with a common state and synchronization. A Module implements one or more Nexus Interfaces.

Modules are an internal architectural construct. Nexus users do not need to know about Module architecture in order to write an application; they only need to understand Nexus Interfaces. However, if you want to write your own Modules or assemble your own Nexus Platforms, then this information is crucial.

The following illustrates how modules interact with applications, with other modules, and with lower-level software:



Interface to Module Mapping

The Interface/Module distinction allows Nexus to transparently support internal connections that Magnum requires without having to push those requirements up to every application. Nexus Interfaces are defined based on user requirements. Magnum APIs are defined based on HW/FW requirements. Nexus Modules stand between these two sets of requirements and make the overall system easier to use.

The mapping of Interfaces to Modules can change over time. This mapping is transparent to the Nexus user and could be changed later without any impact to the application. When moving to a new chip, the underlying hardware or low-level software design may necessitate a change. When adding a new feature, new inter-module communication may require refactoring modules. If the Interfaces do not change, then this module refactoring will not affect applications. It will affect the Nexus directory structure and the location of the Interface header files. See the Nexus_Usage.pdf for a high-level description of the actual Interfaces and their mapping to modules.

If a user tries to use a Nexus Interface in an application but does not include or link in a Nexus Module that provides that Interface, the user will get a compiler error (if the header file is missing) or a linker error (if the implementation is missing). Static binding ensures that these problems are caught at compile time.

Module State

Nexus Modules must follow these rules:

- A module is always a singleton. It has one state in any system.
- A module's state is always protected by one mutex. That state is protected from reentrancy and race conditions.
- A Magnum Porting Interface or SysLib handle can be owned by only one module.

Every module has a single handle which is stored as a global variable. This handle is created at system initialization time. When any Interface is called, it gets access to the handle for its module. No module can access another module's handle without it being explicitly passed to it at system init time.

Software developers are understandably wary of global variables. They can often be a symptom of poor software design. Requiring that all modules be singletons is an essential part of Nexus' synchronization architecture and maps the underlying low-level software and hardware well. When designing Nexus, Broadcom recognizes that the hardware is the ultimate singleton. The Module as a self-synchronizing singleton follows the software pattern called a "Monitor." This pattern is often used to provide general access to a hardware device, which is exactly what Nexus is intended to do. Nexus Module synchronization is provided transparently by means of a synchronization "thunk" layer.

The singleton paradigm still works for a system with multiple set-top SoCs. In those systems, two distinct copies of Magnum must be run in two execution environments (for example, separate applications or drivers). If that is the case, two copies of Nexus should be run or a single copy of Nexus should be implemented, one that is able to interact with two distinct copies of Magnum. The advantage of having modules as singletons is that this allows module implementation, driven by HW/FW requirements, to be separated from the Interface, which is best driven by user requirements. Without this separation, module interconnections would be pushed up to the top-level API, making the API less resilient to internal change.

If there are multiple HW resources to control, the Interface should provide an index (for example, opening a video decoder should always take an integer index to allow for dual-decode systems). This means Nexus Interfaces are always designed for multiple resources and never hard-coded to one (such as multiple PCR blocks, decoders, displays, etc.)

Magnum requires that all calls to each PI or SysLib be serialized by the caller. The rules for Nexus Module state provide inherent serialization of Magnum state.

Magnum state rules are not intended to prevent users from adding their own custom PI calls. Instead, it provides a framework in which these calls can be made safely. See below for a description of how Nexus Modules can be extended with custom features. This allows users customize features while still holding the tight Module State requirements.

Module APIs

The functions that a module exposes to outside callers, either applications or other modules, are its API. They fall into one of the following three categories:

Public APIs

- Public APIs are callable by applications and other modules.
- The sum of all Interfaces exposes by a module is its public API.
- The prototype of the public API is restricted by certain rules to allow for proxying. See the "Coding Convention" section in Nexus_Development.pdf.
- Synchronization is automatic and internal by means of a synchronization thunk.

Private APIs

- Private APIs are callable by other modules which were given the module handle at system init time. They are not callable by applications.
- There are no restrictions on the prototypes of the private API functions.
    - There is an exception for passing module handles. See the following points.

- The caller must already have the module's handle in order to perform explicit synchronization. No private API function can be passed a module handle. If a module handle was passed, there would be no way to guarantee a non-deadlocking system.

The following is an example of module XXX invoking module YYY's private API:

```
NEXUS_XXX_Foo()
{
    /* at this time, XXX is locked. */

    NEXUS_LockModule(xxx.yyyModuleHandle);

    /* during this time, YYY is locked and XXX can make
    private API calls */

    /* Call a private API */
    NEXUS_YYY_WorkWonders_priv(1, 2, 3);

    NEXUS_UnlockModule(xxx.yyyModuleHandle);
}
```

Module APIs

A module can provide functions that are callable from the Platform, for doing system initialization and un-initialization. They are not self-synchronizing. They should be called from a single thread when the system is coming up or shutting down.

```
NEXUS_TransportModule_Init();
NEXUS_DisplayModule_SetSettings();
```

Private Functions

A module can implement private functions, callable from only inside the same module, but they are not consider an API. The prototypes for these functions cannot be put into the modules' include directory. The Nexus build system enforces this by not providing the include path to a module's source directory. Any bypass around this restriction is not allowed. All access to a module must go through an API.

Avoid Modifying Modules

Nexus Modules have been designed so that the customer does not need to modify the module. If you find that your application needs more control or custom features in a Nexus module, please request that Broadcom add a new API to the module. By not modifying the module, you will be able to easily take software updates and future bug fixes without having to reapply your branched or modified code.

Execution Contexts

Nexus can operate across multiple execution contexts. An execution context is a distinct address space or restricted linking. Different execution can be as follows:

- User mode vs. kernel mode
- Separate processes
- Separate device drivers (that cannot link to each other)

In all cases, code cannot call from one context directly into another context. This document describes various techniques for operating across these execution contexts.

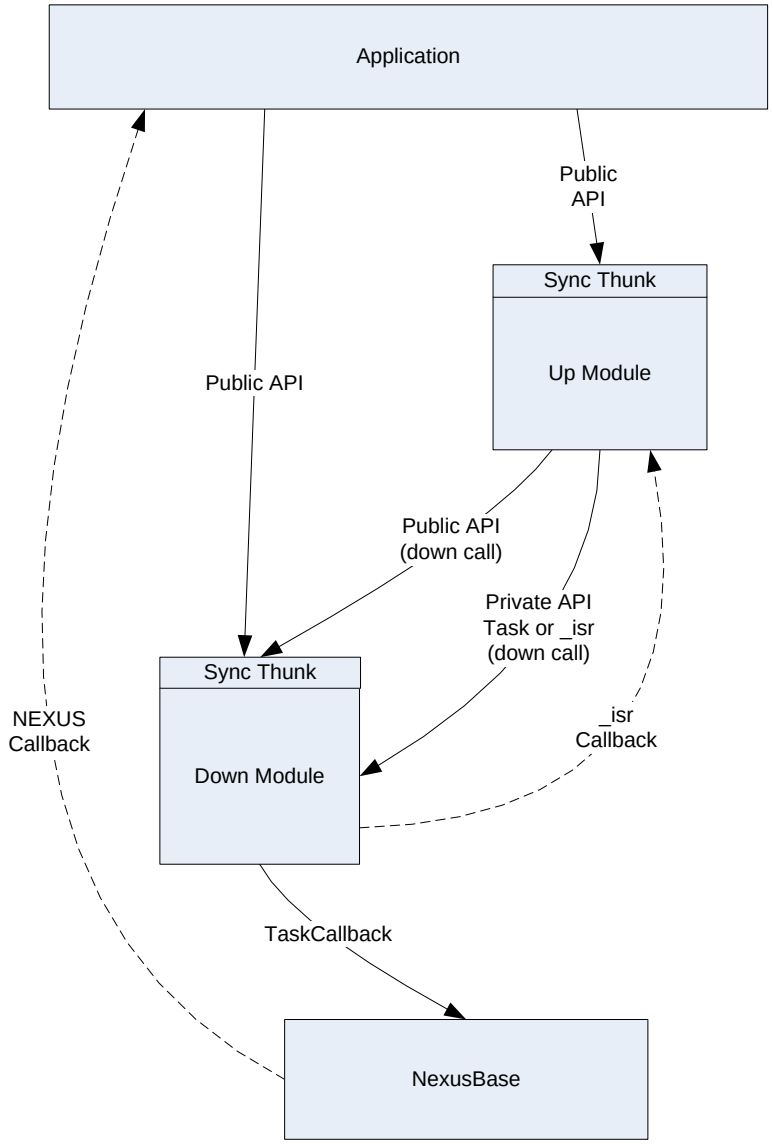
Nexus also operates in both ISR and task contexts. ISR context is a high-priority context for servicing interrupts which has certain restrictions[1]. Task context is the normal execution context. The ISR and task context distinction is inherited from Magnum. Task code can call into ISR code by first entering a critical section. ISR code can never call into task code. See the Magnum architecture documentation for more details.

[1] ISR context cannot use the scheduler, which means it cannot sleep, acquire a mutex, or wait for an event.

Module Interconnections

Nexus modules can call other modules under a set of strict rules. These rules prevent deadlock. Figure 7 shows the different types of interconnections:

Figure 5: Examples of Module Interconnections



Down Calls

A module may call another module's public or private API, but relationship between two modules must be unidirectional. One module is the “up module” and the other module is the “down module.” The up module can call the down module's public or private API, but the down module cannot call the up module’s public or private API.

A down module may communicate with an up module only by callback. That callback could be:

- ISR callback, registered via a private API
- NEXUS_Callback registered via a public or private API
- A BKNI_EventHandle, exposed via a private API, with a NEXUS_RegisterEvent call managed through the Nexus Base.

Any call into a down module is a synchronized call. The up module does not release its mutex while making the call. Deadlock is not possible because no synchronized up call is allowed.

Down Call Enforcement

At Compile Time

The build system requires that module's register the down modules they intend to call using NEXUS_<MODULE>_DEPENDENCIES in their module.inc file. Without this registration, the up module will not have the include path of the down module and so no interface can be included. The build system will check for circular dependencies.

It is important that no module work around the module.inc rules and allow #includes for files that are not in registered down modules. Dangerous deadlock conditions may result.

ISR Calls and Callbacks

Nexus modules can make Magnum ISR calls and service Magnum ISR callbacks. All Magnum synchronization rules apply. Those rules are:

- ISR functions are identified by an “_isr” suffix.
- BKNI_EnterCriticalSection() is required before a task can call an ISR function.
- No ISR function can call a non-ISR (i.e., task) function.

ISR calls are allowed between any two modules. No up/down relationship governs the ISR call. This works because there is only one ISR context in the entire system and so no deadlock can occur. Modules must observe the up/down relationship when making the task calls necessary to set up an ISR callback. For instance, an up module can make a private API task call to give an ISR callback to a

down module. Once that is done, the up and down modules can communicate via ISR down calls and ISR up callbacks freely.
ISR calls and ISR callbacks can only be a part of a private API. This means that ISR callbacks are not allowed from a module to an application. Any feature requiring ISR-level performance must be implemented inside a Nexus Module.

Callbacks

Callback Descriptor

A Nexus callback has a uniform signature:

```
void callbackFunction(void *callerContext, int callerParam)
```

This callback is only called in task context. It can be used in a public API, but may also be used in a private API. The uniform signature makes it possible to centrally manage task callbacks in Nexus Base and across proxy layers.

The NEXUS_CallbackDesc structure (short for Nexus Callback Descriptor) bundles the three required elements of a callback (that is, the function pointer, void * context, and int parameter) in order to make Interfaces simple and consistent.

This is an example of Nexus callback usage:

```
vbiSettings.dataReady.callback = callbackFunction;
vbiSettings.dataReady.context = my_context;
vbiSettings.dataReady.param = my_param;
```

Callback parameters are not references to internal Nexus data. Applications receive a callback when there is a state change, then call Nexus to get the status of what that change is.[2]
This may appear inefficient, but it is not. The vast majority of callbacks occur as the result of interrupts. Any data passed up from the system at ISR time must be copied while in ISR content, then an event must be set, and then the application receives the event in a task and can make changes to the system. Nexus saves the data from ISR callbacks and performs the ISR-to-task conversion internally. The application simply gets the event, reads status, and takes action. It is efficient and safe

Requesting a Callback

A Nexus Module can receive callback function requests from an up-module or an application. It does not know where the callback request comes from or whether it is a nexus module or an application.

[2] Unlike Magnum callbacks, there is no 3rd “void *” parameter that allows the module to pass out data.

If a Module requests a callback from another module, you must use the NEXUS_CallbackHandler infrastructure. This makes synchronization of the callback automatic, just like timers and events. The code looks like this:

```
NEXUS_InterfaceHandle NEXUS_Interface_Open(unsigned index)
{
    NEXUS_InterfaceHandle h;

    /* able to receive a callback from another module */
    NEXUS_CallbackHandler_Init(h->handler, NEXUS_Interface_P_Callback, h);

    /* request callback from another module */
    NEXUS_Other_GetSettings(otherHandle, &otherSettings);
    NEXUS_CallbackHandler_PrepareCallback(h->handler, otherSettings.dataReady);
    NEXUS_Other_SetSettings(otherHandle, &otherSettings);

    return h;
}

void NEXUS_Interface_Close(NEXUS_InterfaceHandle h)
{
    /* disconnect */
    NEXUS_CallbackHandler_Shutdown(h->handler);
    return h;
}

static void NEXUS_Interface_P_Callback(void *context)
{
    NEXUS_InterfaceHandle h = context;
    NEXUS_ASSERT_MODULE(); /* we are already locked */
    NEXUS_Interface_P_DoWork(h);
}
```

Creating and Firing a Callback

The internal callback API includes:

- NEXUS_TaskCallback_Create
- NEXUS_TaskCallback_Set
- NEXUS_TaskCallback_Fire
- NEXUS_TaskCallback_Destroy
- NEXUS_IsrCallback_Create
- NEXUS_IsrCallback_Set
- NEXUS_IsrCallback_Fire_isr
- NEXUS_IsrCallback_Destroy

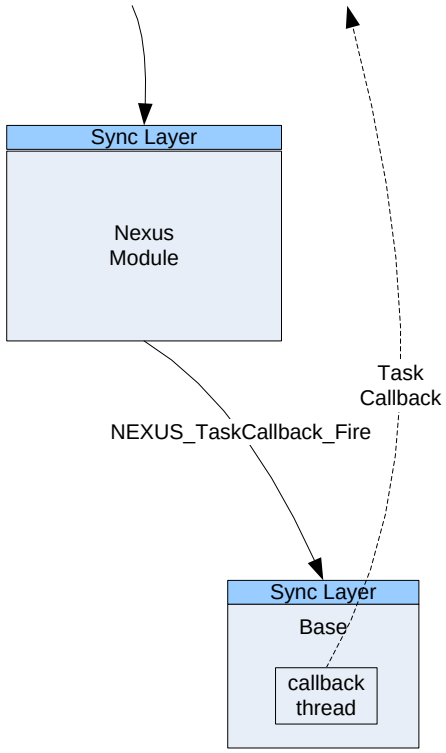
The Nexus module creates a NEXUS_TaskCallback if the callback must be fired from a task context. The module creates a NEXUS_IsrCallback if the callback must be fired from an ISR context.

When the user sets the NEXUS_CallbackDesc, the information is loaded into the pre-allocated NEXUS_TaskCallback or NEXUS_IsrCallback structure and is ready to be fired when the event occurs.

When the event occurs, the Nexus Module fires the callback by invoking NEXUS_TaskCallback_Fire or NEXUS_IsrCallback_Fire_isr. Nexus Base queues the callback for later firing from another thread. It will never execute the callback immediately while the Nexus Module is holding its mutex. This allows the caller to make a call back into Nexus from inside the Nexus callback without deadlocking. Because the Nexus Module does not release its mutex when queuing the callback with Base, there is no possible reentrancy in the module.

Figure 8 shows how Nexus processes callbacks in a Base callback thread.

Figure 6: Base Callback Thread



Callback Performance

Nexus Base uses multiple threads to queue callbacks. The callback thread is the same thread as the event and timer threads. By default, there are four threads created based on module priority: high, default (i.e. medium), low and idle (i.e. very low).

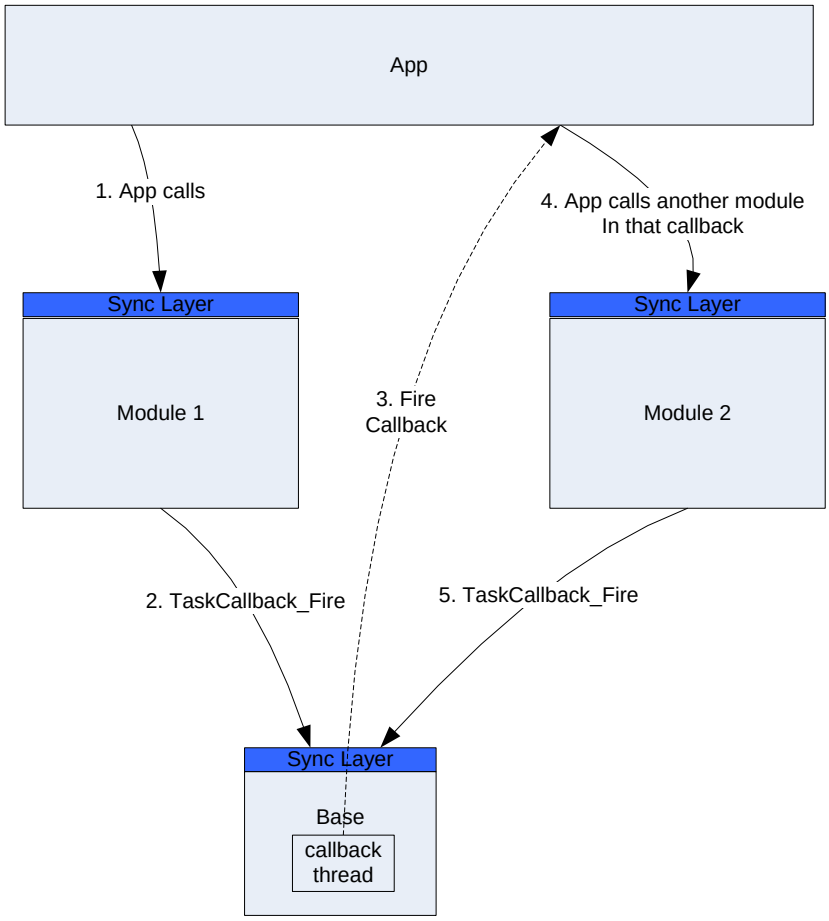
The user must be aware that any work done in the callback should be as quick as possible. Every other callback being processed by the same callback thread will wait while the application does its work. No Nexus state corruption will result, but you might have performance problems like dropped data.

No Magnum ISR callback can be sent to an application. While this may be seen as a limitation, there is actually very little work one can do at ISR time. For instance, an application may want an ISR callback when the video decoder detects a source change. However, some typical work done in response to this information may be to change the format of the display. However, this format change can only be done at task time. Therefore a context switch from ISR to task is required. The conversion from ISR to task time will happen inside Nexus, and the user is free to change the display format directly from the Nexus callback. Only one context switch is required, which is optimal.

Base Reentrancy

When an application receives a callback from Nexus, it is allowed to make immediate calls into Nexus from that callback. This introduces reentrancy. However, because callbacks are fired asynchronously from Nexus modules, the reentrancy burden is concentrated into one place – Nexus Base. Figure 9 shows how callbacks cause reentrancy in a Nexus Base:

Figure 7: Reentrancy in a Nexus Base



The Nexus Base internal callback thread is still pending in step 3 when Module 2 schedules a new callback in step 5. Nexus Base cannot hold its lock in step 3; otherwise the TaskCallback_Fire from Module 2 in step 5 would deadlock. Nexus Base must release its lock in step 3 before firing the callback, and then handle any possible reentrancy after the callback returns and it reacquires its lock. This is the only place in the Nexus architecture where reentrancy may happen. Nexus Base has been carefully written and tested to make sure it is handled correctly.

Application Callback Synchronization

While Nexus internally synchronizes its internal state for callbacks, the application must synchronize its own state when receiving Nexus callbacks. The two main dangers to be aware of are in-flight callbacks arriving after an unregister call, and deadlock when closing a handle inside of a callback.

In-Flight Callbacks

If an application unregisters a callback function, a callback to that function may already be "in flight" from Nexus and arrive afterward. This creates a danger for the following code:

```
void framebuffer_callback(void *context, int param)
{
    DoWork(AppState.resource)
}

app()
{
    settings.framebuffer.callback = NULL;
    NEXUS_Display_SetSettings(display, &settings);
    Dealloc(AppState.resource);
    /* framebuffer_callback may be called here */
}
```

The application must test its own state to be sure the resource is still available. It can do this with a mutex as follows:

```
void framebuffer_callback(void *context, int param)
{
    Lock(AppState.mutex);
    if (AppState.resource) DoWork(AppState.resource)
    Unlock(AppState.mutex);
}

app()
{
    settings.framebuffer.callback = NULL;
    NEXUS_Display_SetSettings(display, &settings);
    Lock(AppState.mutex);
    Dealloc(AppState.resource);
    AppState.resource = NULL;
    Unlock(AppState.mutex);
    /* framebuffer_callback may be called here */
}
```

Or it can use NEXUS_StopCallbacks and NEXUS_StartCallbacks:

```
void framebuffer_callback(void *context, int param)
{
    if (AppState.resource) DoWork(AppState.resource)
}

app()
{
    settings.framebuffer.callback = NULL;
    NEXUS_Display_SetSettings(display, &settings);
    NEXUS_StopCallbacks(display);
    Dealloc(AppState.resource);
    AppState.resource = NULL;
    NEXUS_StartCallbacks(display);
    /* framebuffer_callback may be called here */
}
```

NEXUS_StopCallbacks does not discard in-flight callbacks; it only suspends them. They are all fired when NEXUS_StartCallbacks is called. And NEXUS_StartCallbacks is required otherwise no callbacks will ever come for that handle again.

Deadlock when Closing Interfaces in a Callback

An exception to this handling of in-flight callbacks is a Nexus “Close” function. Because Nexus callbacks are tracked by handle, Nexus discards all in-flight callbacks on “Close” so that none arrive the close. This removes the need for application synchronization in a very common case, like the following:

```
void framebuffer_callback(void *context, int param)
{
    DoWork(AppState.resource)
}

app()
{
    NEXUS_Display_Close(display);
    Dealloc(AppState.resource);
}
```

However, this automatic purge of callbacks creates another risk for applications. In order to do this, Nexus must synchronize with the callback thread while in the Close function. If a Nexus interface is called from inside of a callback itself, a deadlock condition may occur. It is not guaranteed to occur because Nexus modules can be of different priorities and use different callback threads. But to prevent it in all cases, Nexus interfaces should never be closed from inside callbacks.

Slow Callbacks

Applications should avoid blocking or otherwise waiting on any other callback inside a callback. The callback you are waiting for may be serviced on the same thread inside Nexus Base, which is a

deadlock. If the callbacks are serviced from different threads inside Base, deadlock may not occur, but Broadcom highly recommends that applications avoid blocking inside callbacks.

NEXUS_OBJECT

Every Nexus handle that is tracked and verified must be declared as a NEXUS_OBJECT. NEXUS_OBJECT is a set of macros which automatically generate reference counting code. NEXUS_OBJECT also holds run-time type information for the interface and allows for registration with the Object Database (objdb). Every NEXUS_OBJECT must have one or more constructors and one destructor. Constructors are usually called Open or Create. Destructors are usually called Close or Destroy.

When an object is created with NEXUS_OBJECT_INIT, its reference count is set to 1. In can be further incremented and decremented with NEXUS_OBJECT_ACQUIRE and NEXUS_OBJECT_RELEASE to protect interdependencies between interfaces. When an object's destructor is called by the application, its reference count is decremented. When the reference count falls to 0, Nexus will actually destroy it by calling the "Finalizer".

The Close or Destroy function for a NEXUS_OBJECT is auto-generated using the NEXUS_OBJECT_CLASS_MAKE macro. The only purpose of the public Close or Destroy is to decrement the reference count. The NEXUS_OBJECT_CLASS_MAKE macro requires that a static function called NEXUS_<InterfaceName>_P_Finalizer be written.

As an option, NEXUS_OBJECT_CLASS_MAKE_WITH_RELEASE can be declared. This requires The _P_Finalizer function as well as a _P_Release function. The _P_Release function is called when the destructor is called. If the reference count has been incremented with NEXUS_OBJECT_ACQUIRE, the _P_Finalizer will not be called immediately after.

Object Database (objdb)

Nexus uses its handle tracking to verify handles in the kernel mode and user mode proxies. It does this with an object database. The object database is in nexus/base/src/b_objdb.c. It is build on top of NEXUS_OBJECT: NEXUS_OBJECT can work without the object database, but not vice versa.

The proxy will automatically insert or remove handles into and out of the object database. Handles can also be manually inserted or removed using NEXUS_OBJECT_REGISTER and NEXUS_OBJECT_UNREGISTER.

For protected or untrusted mode clients, their handles are cross-referenced with the object database before they are allowed to be used. See nexus/docs/Nexus_MultiProcess.pdf for a complete discussion of handle verification.

Limiting client resources

Nexus allows the server to limit the API's and resources that untrusted clients can access.

The autogenerated proxy cross-references with nexus_untrusted_api.txt in nexus/build/common/tools to learn which functions are callable by untrusted clients.

In addition, nexus modules call NEXUS_CLIENT_RESOURCES_ACQUIRE and NEXUS_CLIENT_RESOURCES_RELEASE to perform per-client bookkeeping for those interfaces. This limits the number of times an interface can be opened or which ID's can be opened or acquired.

See nexus/docs/Nexus_MultiProcess.pdf for a complete discussion of untrusted clients.

Nexus Interfaces

Overview

An Interface is a set of function signatures (that is, prototypes), along with their structures and enums, which present a feature.

An Interface consists only of functions. For comparison, Java interfaces contain both methods *and* properties; Nexus has no concept of properties.

An Interface can connect to another interface. Connections are considered *loosely* bound or *tightly* bound, depending on what variables and underlying interconnections they need to share. Recurring connection patterns have been identified when designing the API in order to make the overall system more uniform and understandable.

The granularity of an Interface is a measure of how many features are collected together in a single Interface. Broadcom has attempted to design Interfaces with the appropriate granularity (not too high, not too low) to allow for easy mapping to customer middleware and HALs.

If the granularity is too high, then the Interface will never be expressive enough to accomplish all user configurations. The API will grow and become complicated. If the granularity is too low, then the user must write a lot of integration code to get a simple feature done.

An Interface is statically bound with its implementation. If a module needs to implement a dynamic binding to various implementations, it can do so, but this is not part of the standard Interface architecture.

Nexus Interfaces are analogous to the Interfaces of 4GL languages like Java and C#, but Broadcom is using 4GL concepts while doing the implementation in the 3GL language of C. Nexus Interfaces are also analogous to DCOM and CORBA Interfaces, while using a simpler implementation model with static binding.

Nexus has no IDL (interface definition language). Instead, Nexus has a strict coding convention that allows Perl scripts to scan the public API header files and generate synchronization thunks and user/kernel mode proxy layers. This gives Nexus the high-level features it needs with minimal overhead.

Reentrancy

Nexus Interfaces are reentrant. The application can call any Nexus Interface from any thread without danger of internal Nexus corruption.

Applications can make Nexus calls from within Nexus callbacks[3]. While this is thread-safe, users should be aware that slow work in a Nexus callback may cause performance problems in other parts of the system. All work in a callback should be minimal and as fast as possible.

While Nexus is internally thread-safe, an application can still get unpredictable results if it tries to access an Interface from multiple threads. For instance, if one thread calls NEXUS_VideoWindow_GetSettings/SetSettings and another thread calls the same GetSettings/SetSettings with the same handle, although Nexus will internally synchronize the individual calls, the application could get unexpected data due to race conditions.

Allocation

All allocation and de-allocation is explicit in Nexus. There is no garbage collection. If a user opens a handle, it is responsible to close that handle. For most interfaces, Open cannot be called more than once for the same index. Some interfaces support virtualization where Open can be called more than once with the same index. In all cases, open must be paired with close. All handle management is explicit.

Interface Connections

Interfaces interconnect in several standard ways. Options include:

Shared Handle → Tight Binding

When an Interface's handle is given to another Interface, they are tight binding. They must execute in the same execution context. Sharing of handles is always unidirectional. An up module may acquire an Interface handle from a down module, but never vice-versa.
If your application does not allow for shared handles, a "Get(index)" interface can be used to retrieve the handle, thus creating a semi-loose binding Interface. The user should be aware that getting a handle, apart from Open/Close, requires the user to carefully manage the state of that handle. A handle should never be used after it is Closed.

Indexed Resource → Loose Binding

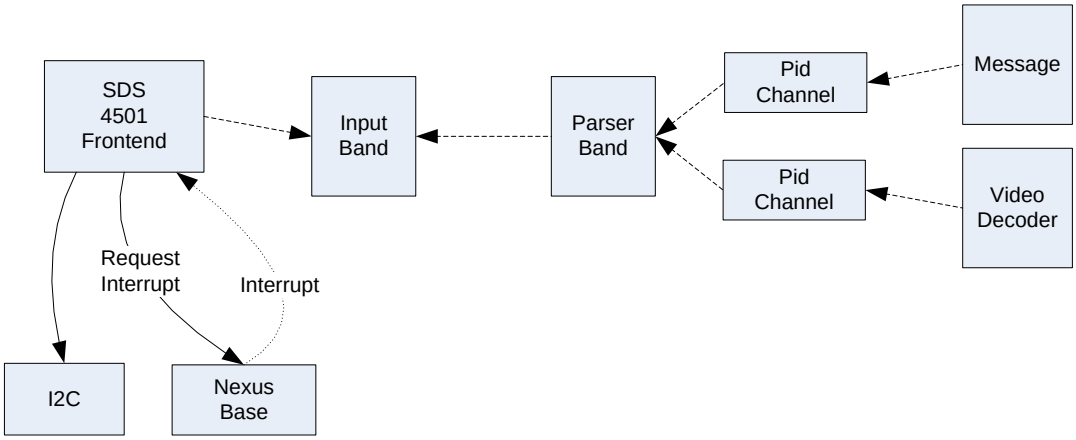
Two Interfaces use the same enum. Connections are made by the application giving an enum and integer index to another Interface. Often the enum/index refers to a hardware resource (e.g. parser

[3]There are some important exceptions. Applications should not close any interfaces inside one of its callbacks. Applications should also not wait for another callback inside of a callback. See "Callbacks " on page 29 for more details.

band) and the HW connection between two SW modules is sufficient. No additional SW connection is needed. If there is no internal software connection, they may execute in different execution contexts.

Figure 10 shows the loose binding between front-end and back-end Interfaces.

Figure 8: Front End/Back End Separation



The connection between a demodulator interface (front end) and the decoder interface (back end) is through transport. The connection through transport is through input bands and parser bands. Input bands and parser bands are managed through enumerated types; therefore, the connection between front-end and back-end code uses loose binding.

Abstract Connector → Loose Binding

In the audio and video subsystems, abstract connector tokens are used (NEXUS_VideoInput, NEXUS_VideoOutput, NEXUS_AudioInput, NEXUS_AudioOutput) to connect Interfaces. This abstraction preserves modularity and allows complex filter graphs to be constructed.
See Nexus_Development.pdf for implementation details and information about Connection Patterns when designing Nexus Interfaces.

Limitations on Nexus “Get” functions

Nexus is designed to provide a minimal set of efficient tools that are sufficient to build large-scale applications. This means that Nexus does not provide certain functions, not because they are technically difficult to provide, but because they are not essential and they lead to difficult system architecture issues.
These functions are typically “Get” functions. A "Get" function returns information from the system. This information can be either status or settings. Status is internal information that is output from a system to a user. Settings are external information provided as input to a system from a user. Nexus maintains a strict separation of status and settings in its structures and functions.

Getting status is typically not a problem. Status is usually returned as a single structure with various simple data types like integers and enumerations. Status structures typically do not include handles. Status cannot include pointers to other structures (which would require deep-copy over a proxy).

Getting settings can pose a problem. As a design rule, we do not provide a “Get” function for settings unless the user can make an incremental change with the returned settings. The following is a typical example:

```
NEXUS_VideoWindow_GetSettings(window, &windowSettings);
windowSettings.position.x = 10;
windowSettings.position.y = 10;
rc = NEXUS_VideoWindow_SetSettings(window, &windowSettings);
```

The following are examples of settings for which there is no Get function:

- There is no function to get settings which cannot be incrementally changed. For example, NEXUS_VideoDecoderStartSettings is passed into NEXUS_VideoDecoder_Start; it cannot be modified after starting decode.
- There is no function to get a handle that was opened. For example, `NEXUS_Display_Open(index, &settings)` returns a handle, but there is no `NEXUS_Display_Get(index)`.
- In general there is no way to get a handle that was connected. For example, `NEXUS_VideoWindow_AddInput(window, input)` connects a window to an input, but there is no `NEXUS_VideoWindow_GetInput(window)`.

Note that getting handles is a form of getting settings. Getting handles is not supported.
The reason for this policy regarding “Get” is that Nexus should not relieve applications and libraries from having to design their handle and settings management.

Handle Management

One of the main responsibilities of an application and its libraries is to allocate, manage, and de-allocate resources. If an application opens or creates a handle, it is responsible for that handle's use until it closes or destroys that handle. It manages the lifecycle of the handle. If a subroutine or library needs that handle, it must be given it by the handle owner. The giving of handles involves a contract between the resource owner and any resource users as to what can be done with that handle. This is a basic design requirement of good code.

One often-overlooked consideration of handle management is the fact that there is no safe way to verify dynamically allocated handles. If code uses a handle that has been closed, the handle is typically pointing to freed memory. This memory cannot be safely accessed because it may have been reallocated and reused. If Nexus tried to validate a handle and return an error code, applications would think they could incorporate that Nexus function failure into their logic only to discover very subtle bugs later on. The best action is for the Nexus function to immediately assert so that the bug can be fixed during application development. Nexus uses BDBG_OBJECT to attempt this[4]. See Nexus_Development.pdf for details on parameter validation.

Settings Management

A similar system-level synchronization problem exists for getting settings, even in cases where Nexus provides a GetSettings() function. Nexus can be called from multiple threads, but that does not mean that those two threads need not communicate. If one thread might set new settings that the other thread overwrites with a read/modify/write race condition, Nexus' internal state is fine, but it is a bug because the application does not get the desired result.

Recommended Solutions

The request for a Get function typically arises from a large system with parts A and B which both have access to a shared resource X, but don't communicate otherwise. This can be illustrated like this:

[A] → [X] ← [B]

There are two ways for the application to resolve handle and settings management problems that arise.

- **Unification**: Perhaps A and B are really working at the same thing, and so they should be synchronized in the application. They need to know about each other. That is:
  [A B] → [X]
- **Arbitration**: There is should be some application-specific arbiter X' that encapsulates and controls access to X. That is:

[4] There is no way to guarantee that BDBG_OBJECT_ASSERT will catch a freed handle. It only greatly increases the chance that it will be caught.

[A] → [X' [X]] ← [B]

These are nontrivial design issues that Nexus cannot solve and which nonessential “Get” functions would exacerbate. Nexus has been designed so that the application must conscientiously manage its handles, interconnections, and settings. Broadcom realizes this creates design burdens for code built on top of Nexus. In the short term, this can be an annoyance, but in the long term it leads to better code design, both inside and above Nexus.

Nexus Platforms

Overview

A platform is user-implemented code that collects various Nexus Modules into a cohesive whole for use on a specific board or application.
Although there are common features in the platform, there is no architecture for how Platform code must be organized. Every platform is custom.
Every Nexus release comes with sample Platform code that is designed for a Broadcom reference board. Please refer to that code to learn how to use Nexus. Feel free to use or not use that code.

Typical Features

A platform includes the following services:

Driver/OS Code

- OS driver interface
  - For instance, Linux user or kernel modes
  - Proxy layer for user->kernel calls or other required execution context changes.
  - A proxy thunk is not required. If a Platform needs to use manually generated ioctls, it is free to do this.
- L1 interrupt mapping
- Memory mapping

Board Code

- Module initialization
  - Every module must be brought up in correct order. Any dependency between module's will be expressed with module handle sharing.
  - Modules may expose init-time APIs. The Platform code can decided how to bring up the module's based on system requirements.
- Pin-mux code
- Memory layout
  - Where does heap memory start? How many heaps are there?

Build System

- The platform will contain a Makefile which can build a Nexus library and/or driver.
- The Makefile should use the modulename.inc files. All other aspects of the Makefile can be custom.
- The build system must generate the synchronization thunk and any proxy thunk.
  - Although the exact method of generating this is not required, we will have reusable libraries and recommended techniques.

Application Interface

- Makefile include (i.e. platform_app.inc) which allows an application to easily link to Nexus.

> **Note:** There will not be a required universal nexus.h that collects all public APIs for a platform. The platform could choose to make such a nexus.h, or the applications could be required to #include the public APIs needed in each source file.

- The Platform could have its own API for an application to configure it.
  Because every Platform is board/application specific, it could also choose to have no API and just internally code all requirements. This is up to the platform developer.
- Broadcom does recommend that every platform implement one common function:

```
NEXUS_Platform_Init(const XXXX_Settings *pSettings)
```

This allows a generic example application to run, assuming it passes a NULL pSettings parameter. This is not required. If you do not provide this generic init, you simply will not be able to run an unmodified example application.

Enumerating Platform Features

Modules can have one dependency on platform code. They may #include "nexus_platform_features.h" and use a standard set of #defines which enumerate the number of resources available. For instance:

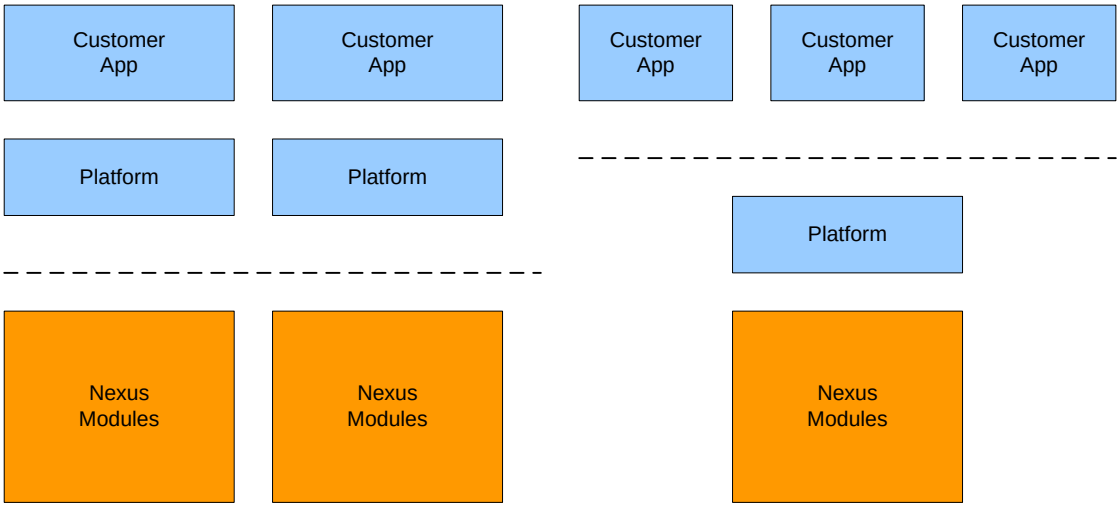
```
#include "nexus_platform_features.h"

NEXUS_Module_Func()
{
#if NEXUS_NUM_VIDEO_DECODERS
    do_work();
#else
    return BERR_TRACE(NEXUS_NOT_SUPPORTED);
#endif
}
```

These #defines usually follow the form of NEXUS_NUM_XXX. See nexus/platforms/97445/include/nexus_platform_features.h for a typical list of #defines

Example Platform Options

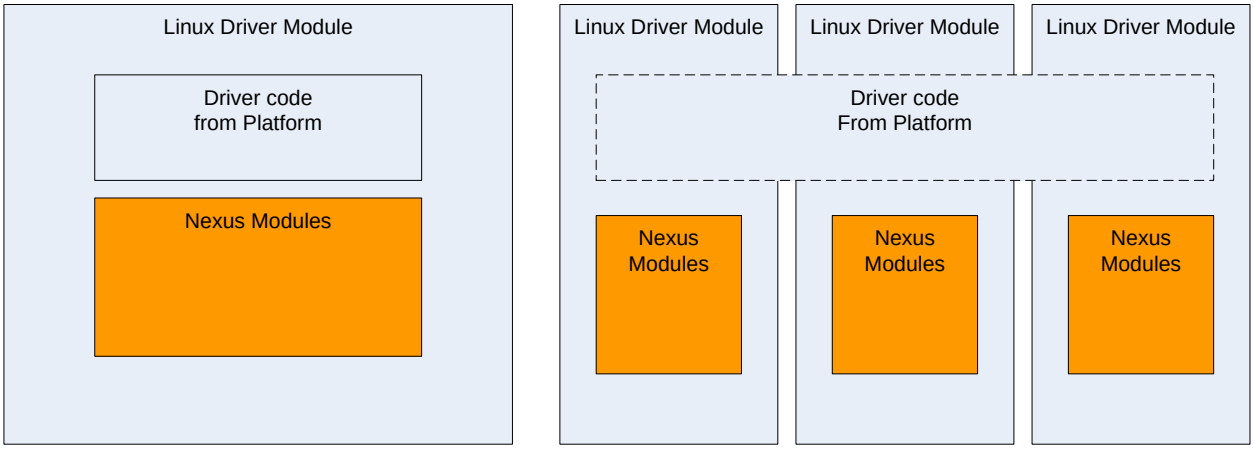
Figure 11 shows some examples of how Platforms may differ between users.
Some users may deliver a separate Platform per application and embed the application-specific information in the Platform. Other users may deliver a multi-application Platform and expose options via a Platform API.

Figure 9: How Platforms Differ Between Users



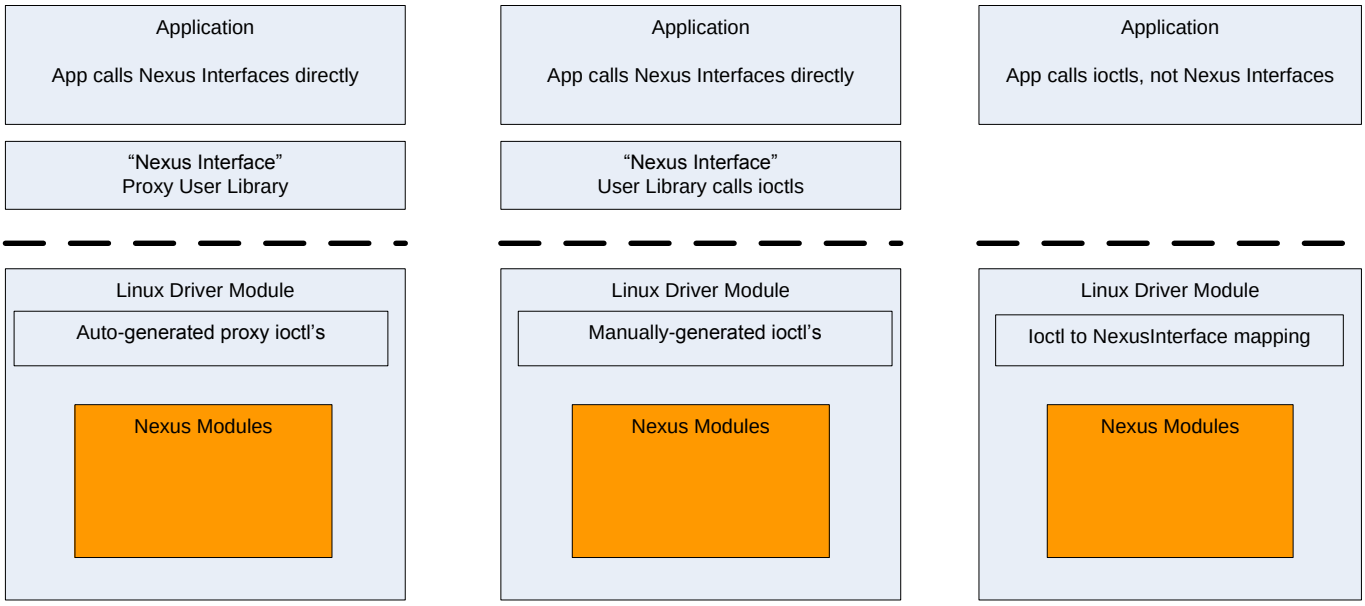
Some users may prefer putting all of Nexus into one Linux driver module. Other users may want to insmod separate modules. Of course, by breaking things up you will need to export driver module symbols to make the necessary module interconnects.

Figure 10: Use of Nexus in a Linux Driver Module



Some users may prefer to use the automatic Nexus Interface proxy to hide the user-to-kernel transition. Others may prefer to expose and support an ioctl layer.

Figure 11: Use of the Automatic Nexus Interface Proxy



Nexus Base

Overview

Nexus Base provides a set of high-level software services that enable the Nexus architecture. These services are high-level OS abstractions that extend the capabilities of Magnum's KNI base module. Nexus Modules can use both KNI and Nexus Base.

Base is not a Nexus Module and cannot be called by applications. It has no synchronization thunk layer. It is reentrant in many places. It has functions that require module handles that cannot be a part of the Nexus public API.

Base cannot be proxied across execution environments. You may run multiple instances of the Base module in a system if there are multiple execution contexts, usually one in each context (e.g. Linux kernel and user modes).

Nexus Base provides the following features:

Events

Base allows a Module to receive a task-time callback in response to a BKNI_SetEvent. Events are serviced in the same thread as timers. Base synchronizes with the module by acquiring the module's mutex before the callback is dispatched.

Timers

Base allows a Module to receive a task-time callback after a specified amount of time has elapsed. Timers are serviced in the same thread as events. The module's synchronization is acquired before the callback is dispatched.

Callbacks

The module can queue an asynchronous callback an application or another Nexus module. Base supports two flavors: TaskCallback and IsrCallback. Use TaskCallback if you fire the callback from inside a task context. Use IsrCallback if you fire the callback from inside an isr context.
Unlike Events and Timers, no module synchronization is obtained before firing the callback; this means the caller is responsible for obtaining synchronization when the callback is received.

Module Priorities and Base Threads

Nexus Base creates multiple threads for servicing callbacks, events, and timers. The purpose of having priorities is so that high-priority code can be serviced separately from low-priority code so that we don't have something fast (with low latency requirements) waiting on something slow (with high latency requirements).

The default system will create four threads mapping to four module priorities:

- High (Example module: transport, dma) – all API's should be non-blocking
- Default (Example module: display) – some API's can block for a few milliseconds
- Low (Example module: video_decoder, audio) – some API's may block for 10 milliseconds or more
- Idle (Example module: i^2c, frontend) – some API's may block for 100's of milliseconds or more

When modules are created, they specify their own priority based on their own characteristics and requirements.

Module priorities do not necessarily map to OS thread priorities. The Base API, callable from a customized NEXUS_Platform_Init, can be used to set OS-specific priorities. The appropriate OS-specific code for creating threads can be added to nexus/base/src/$(B_REFSW_OS).

Because Nexus code is generally not CPU-bound, the OS's scheduler priority for Nexus threads is usually not significant. However, if your application has CPU-bound code, those CPU-bound threads should be set at a lower priority than Nexus threads so that Nexus code is not starved. Remember also that calls into the Nexus public API will execute in the caller's thread. You should limit the amount of work done in Nexus callbacks and not call slow Nexus APIs (such as Frontend) from fast callbacks (such as Transport).

Comparison with Magnum KNI

Nexus Base is analogous to Magnum's KNI (kernel interface) base module. KNI provides a low-level OS abstraction. Nexus Base provides a higher-level set of OS abstractions. The following is a comparison:

Table 1: OS Abstractions in Magnum and Nexus

Type	Magnum	Nexus
Timers	Magnum can request a timer from the TMR PI, but it will only come in at ISR time.	Nexus can request a synchronized task-time timer from Base
Events	Magnum can set an event. It generally should not wait on an event, otherwise the interface is slow.	Nexus can request a synchronized task-time callback from Base when an event fires.
Tasks	Magnum cannot spawn tasks, consume tasks or synchronize between tasks.	Nexus can spawn tasks, consume tasks and synchronize between tasks.
Callbacks	Magnum should only fire ISR callbacks. Task-time callbacks are not allowed.	Nexus can fire task callbacks which have no synchronization dangers. See Callbacks for dangers of an application doing slow work in a callback.
ISR handling	Magnum can receive an ISR, but cannot schedule task-time work in response to that ISR. It must request the application callback at task time and advance its state machine.	Nexus can receive ISR's, processes them at ISR time, and/or scheduled synchronized task-time work using events.

Priority Inversion

Nexus Base will create and manage multiple threads in order to service events, timers and callbacks. These threads will be created with priority set by the module.

This creates potential priority inversion problems, especially for OS's which don't automatically resolve priority inversion live locks (i.e. a high priority task waiting on a resource which is held by a low priority task and the scheduler doesn't allow the low priority task to run). Most modern OS's will resolve this situation automatically and the programmer can safely use Base in its default configuration.

There is no simple solution for OS's which don't resolve priority inversion live locks. However, Nexus Base will have an Init interface where the application can set the # of threads (including setting it to one) and priority per thread. If it is suspected that the OS may have a live-lock problem, use this interface. Be aware that the final solution will involve these trade-offs:

- If multiple threads are used, the application will be responsible for being selective about what work is done in a callback to prevent live lock.
- If a single thread is used, overall system performance may suffer.

Viewing Module Priorities

Every module sets its own priority inside its NEXUS_XxxModule_Init function. The module is a good judge of its own priority because it knows if the type of work done in its callbacks must be slow or fast. A module can provide user-programmable priority by exposing a parameter in its NEXUS_XxxModuleSettings structure.

If you want to see the module priorities in your system, you can use the debug interface as follows:

```
# export msg_modules=nexus_base
# nexus decode

… Other output …

*** 00:00:00.037 nexus_platform_linuxuser: Using realtime priority for ISR
task
--- 00:00:00.165 nexus_base: Creating module core, priority 2
--- 00:00:00.169 nexus_base: Creating module i2c, priority 0
--- 00:00:00.177 nexus_base: Creating module security, priority 1
--- 00:00:00.188 nexus_base: Creating module dma, priority 3
--- 00:00:00.302 nexus_base: Creating module ir_input, priority 2
--- 00:00:00.302 nexus_base: Creating module gpio, priority 2
--- 00:00:00.303 nexus_base: Creating module spi, priority 2
--- 00:00:00.410 nexus_base: Creating module led, priority 2
--- 00:00:00.410 nexus_base: Creating module keypad, priority 2
```

```
--- 00:00:00.411 nexus_base: Creating module ir_blaster, priority 2
--- 00:00:00.411 nexus_base: Creating module uart, priority 2
--- 00:00:00.411 nexus_base: Creating module frontend, priority 0
--- 00:00:00.412 nexus_base: Creating module transport, priority 3
--- 00:00:00.568 nexus_base: Creating module surface, priority 3
--- 00:00:00.569 nexus_base: Creating module graphics2d, priority 1
--- 00:00:00.569 nexus_base: Creating module hdmi_input, priority 1
--- 00:00:00.570 nexus_base: Creating module audio, priority 1
--- 00:00:01.200 nexus_base: Creating module videodecoder, priority 1
--- 00:00:01.685 nexus_base: Creating module display, priority 2
--- 00:00:02.105 nexus_base: Creating module pwm, priority 2
--- 00:00:02.105 nexus_base: Creating module picture_decoder, priority 1
--- 00:00:02.105 nexus_base: Creating module sync_channel, priority 1
--- 00:00:02.105 nexus_base: Creating module astm, priority 1
--- 00:00:02.106 nexus_base: Creating module file, priority 2
--- 00:00:02.110 nexus_base: Creating module playback, priority 2
--- 00:00:02.111 nexus_base: Creating module record, priority 2
```

The application should be aware of module priority when handling Nexus callbacks. As a general rule, it's always best to be as fast as possible inside a callback. If the module priority is high (e.g. 3), then you must be very fast. If the module priority is low (e.g. 0), then it probably doesn't matter much how fast or slow you are.

Nexus Core Module

The Nexus Core module owns all the Magnum Base Module handles. These include:

- Register Interface (REG)
- Chip Interface (CHP)
- Managed Memory Allocator (MMA)
- Interrupt Interface (INT)
- Box mode interface (BOX)

These Magnum handles are not accessible to applications; this means the APIs to retrieve them are private APIs.

The Core module has some general purpose functions that are callable from an application. The purpose of these public functions is usually data type conversion or the exposure of some Nexus Base features to applications.

Thunks

A “thunk” is a transparent wrapper around an API that adds a service. Nexus uses thunks to add process synchronization and to allow proxying between execution environments. Nexus provides three: one for synchronization, one for kernel mode proxy and one for user-mode proxy. This follows the “Monitor” pattern, where implicit synchronization is transparently added to an API with some compiler/preprocessor tool.

The thunks are automatically generated during the build process by Perl scripts stored in nexus/build/tools. The autogenerated code is stored in obj.$(NEXUS_PLATFORM)/nexus/core/syncthunk. Every module’s public API is passed through the thunking process. A module’s private API is not thunked. The caller is responsible to acquire the module's lock, make private API calls, and then release the lock.

Synchronization Thunk

Every Nexus Interface has a synchronization thunk which acquires the module’s mutex. The thunk layer automatically acquires the module's mutex, calls into the implementation, then releases the mutex before returning to the caller. The synchronization thunk is used in Linux usermode on the server and in kernel mode for code not using the proxy.

The synchronization thunk looks like this:

- **nexus_MODULE_thunks.c**–The implementation of each thunk.

```
NEXUS_Error NEXUS_Module_Func(int parm1, int parm2)
{
    NEXUS _Error rc;
    NEXUS_LockModule();
    rc = NEXUS_Module_Func_impl(parm1, parm2);
    NEXUS_UnlockModule();
    return rc;
}
```

- **nexus_MODULE_thunks.h**–The redefinition of the public to the _impl function.

```
#define NEXUS_Module_Func NEXUS_Module_Func_impl
```

The _impl functions created by the thunk are never callable by applications or other modules. There is no exception. It is an internal, implementation detail.

Kernel Proxy Thunk

The kernel proxy thunk converts Nexus API calls into Linux ioctl system calls. The following code is a simplified version of the actual kernel proxy:

- **nexus_MODULE_proxy.c** – the user-mode conversion to an ioctl

```
NEXUS_Error NEXUS_Message_GetStatus(NEXUS_MessageHandle handle, NEXUS_MessageStatus
*pStatus)
{
    int rc;
    TRANSPORT_NEXUS_Message_GetStatus_data st;
    st.handle = handle;
    st.pStatus = pStatus;
    rc = ioctl(nexus_ state.fd, IOCTL_TRANSPORT_NEXUS_Message_GetStatus, &st);
    if (rc!=0) return BERR_TRACE(NEXUS_OS_ERROR);
    return st.__retval;
}
```

- **nexus_MODULE_driver.c** – the kernel-mode processing of that ioctl

```
case NEXUS_IOCTL_NUM(IOCTL_TRANSPORT_NEXUS_Message_GetStatus):
    copy_from_user_small ( &ioctl , (void*)arg , sizeof(ioctl) );
    if(!nexus_p_api_call_verify(&client->client, NEXUS_MODULE_SELF, ...)) {
        goto err_fault;
    }

    ioctl.__retval = NEXUS_Message_GetStatus(ioctl.handle, &pStatus);

    nexus_p_api_call_completed(&client->client, NEXUS_MODULE_SELF, ...);
    if (ioctl.__retval == NEXUS_SUCCESS) {
        copy_to_user_small(ioctl.pStatus , &pStatus , sizeof(pStatus));
    }
    copy_to_user_small ( &arg->__retval , &ioctl.__retval , sizeof(ioctl.__retval) );
    break;
```

User-mode Proxy Thunk

The user-mode proxy thunk converts Nexus API calls into socket calls from a client application to the server application. The following code is a simplified version of the actual user-mode proxy:

- **nexus_MODULE_ipc_client.c** – the client conversion from API call to a socket message

```
NEXUS_Error NEXUS_Message_GetStatus( NEXUS_MessageHandle handle, NEXUS_MessageStatus
*pStatus )
{
    __rc = NEXUS_P_Client_LockModule(nexus_client_module_state, &temp, &__data_size);
    if (__rc) {__rc=BERR_TRACE(__rc);goto err_lock;}
    out_data = in_data = temp;

    in_data->data.handle = handle;
    __rc = NEXUS_P_Client_CallServer(state, in_data, __n, out_data, sz, &__n);
    if (!__rc && pStatus) {
      *pStatus = out_data->data.pStatus;
    }
    return out_data->data.__retval;
}
```

- **nexus_MODULE_ipc_server.c** – the server conversion from socket message to API call

```
case B_IPC_TRANSPORT_NEXUS_Message_GetStatus:
   rc = nexus_p_api_call_verify(&client->client, NEXUS_MODULE_SELF, ...);
   if(rc!=NEXUS_SUCCESS) {goto err_fault;}

   out_data->data.__retval = NEXUS_Message_GetStatus ( in_data->data.handle ,
       &out_data->data.pStatus ) ;

   nexus_p_api_call_completed(&client->client,  NEXUS_MODULE_SELF, ...);
   break;
```

Attributes

Nexus supports a variety of “attributes” which provide hints to the various thunks. Attributes are located within comments. You can only have one attribute per comment. Each attribute is a set of one or more name/value pairs. If you have more than one, the delimiter is a ‘;’. They follow this syntax:

```
/* attr{name=value[;name=value][;name=value]} */
```

Attributes may be attached to functions, function parameters or structure members. The syntax for each is as follows:

Function attribute

```
void NEXUS_Interface_Foo( /* attr{name=value} */
    int param
    );
```

Function parameter attribute

```
void NEXUS_Interface_Foo(
    int param1, /* attr{name=value} */
    int param2
    );
```

Structure member attribute

```
typedef struct NEXUS_Settings
{
    int member;  /* attr{name=value} */
} NEXUS_Settings;
```

The following attributes are supported:

Attribute Name	Type	Meaning
destructor	function	Defines a class used for handle verification. This function is a constructor, paired with a specified destructor. More than only constructor can be paired with the same destructor.
shutdown	function	Registers a function that must be called before handle is closed. Requires an implementation in the platform.
release	function	Defines acquire/release pair for client. Requires that the handle also have a destructor.
local	function	Proxy should implement a local implementation. There is no valid remote implementation. For example, NEXUS_Surface_Flush.
nelem	function param	Input or output parameter which is a variable list of elements or (if void*) bytes. For input parameters, it determines the allocation and memcpy of the array. For output parameters, it determines the allocation of the array. If nelem_out is also specified, nelem is not used for memcpy of the array. If nelem_out is not specified, nelem is used for memcpy as well.
nelem_out	function param	Output parameter which determines the number of elements to memcpy. Requires nelem to specify the maximum size that can be output. This is optional. If not specified, nelem will be used for memcpy. Doing a memcpy of nelem will always be valid, but may be more than is required.
nelem_convert	function param	Invoke a driver-side macro to convert an nelem parameter
memory=cached	function param, struct member	BMEM heap pointer which must be converted to local memory map
null_allowed=y	function param	Pointer which may be NULL. If null_allowed is not set, pointer may not be NULL.
reserved	function param	Used in conjunction with nelem. Pre-allocated a fixed length number of elements/bytes for faster proxy. Only used in kernel mode.

Thunking Exceptions

The following file names have special processing by the thunking scripts:

include/priv/* - private API which is not thunked

include/*_init.h – module initialization API which is not thunked

include/*_private.h – thunked API, but for Broadcom-internal use. API's in these header files are subject to change without notice. They are used for internal communication between the user space proxy and kernel mode driver.

Local Source

A module can declare some of its source to be local. It must declare the function to be local using attr{local=yes} attribute, then implement that function in a .c file which is added to the <MODULE>.inc's NEXUS_<MODULE>_LOCAL_SOURCES.

Local functions will run in the context of the caller. For kernel mode, they will run in the user mode proxy. For user mode multi-process, they will run in the client's IPC proxy. Local functions are also compiled into the server or driver, so they can be called from single process or non-proxied kernel mode.

A common purpose of local functions is to process memory pointers in the caller's address space when a pointer to the same memory may not exist in the driver's space.

Module Extensions

A requirement of the Nexus architecture is that Nexus be extensible. Nexus users need to be able to get a new feature (often a custom feature) into the system without having to modify the core features and creating a long-term support burden.

Extension Options

Extensibility can be achieved with multiple techniques:

1. **New Module**: Creating a new module allows you to implement features without modifying underlying modules If the new module only uses PIs that it owns, no private connection is needed with other modules. It builds on top of the existing public API.

   If the new module calls a lower module's public API, it must be careful to document what it is doing so that it doesn't conflict with an application's use of the lower module's API. If the new module will be widely used, it might be best for the lower module to expose a private API to guarantee no conflict (for example, VideoDecoder has a private API dedicated to the SyncChannel and Astm modules).

   The new module may also use a lower module's Magnum Porting Interface handles or receive ISR callbacks, if the lower module exposes them through its private API. The two module authors must work in close communication to avoid overlapping calls to the PI and avoiding bad timing conditions.

2. **Replacement Module:** Replacing an existing module with a new implementation of a module. If the public and private APIs match (which can be enforced with source control symlinks), it should slip in.

3. **Module Extension:** Adding features to an existing module using a build system hook and optional run-time hooks. The extension's code runs as if it were part of the existing module with full access to all of the module's resources. The extension can write a public API which reuses the Interface handles and is integrated into the existing module's synchronization thunk. The new extension functions run as if it were part of the existing module.

   Because of this close connection, there are coding conventions that are described in the following section. In addition, a robust integration process is highly recommended.

Module Extension Convention

A module extension is a section of code that is added into a core module.

Hooks

The existing module must provide two types of hooks to enable an extension:

1. **Build system hook**: Inside the module's .inc file, it must provide an option to include an extension .inc. This option should be the path of the extension's .inc file. The extension's .inc will then extend the module's Makefile variables.
2. **nexus_<MODULE>_module.h hook**: The thunk layer includes this file and it is required that this include file include the entire API. In order to keep changes to this main header file at a minimum, we recommend creating nexus_<MODULE>_extensions.h and adding the hooks in there.
3. **Optional runtime hooks**: Somewhere inside the module, a `#if` is usually required for the module to call into the extension code, either to initialize its state or to provide a callback.

These hooks should be minimal (that is, one line of code). Any one running without the extension does not need to be burdened by a heavy footprint of an unknown hook.
The hooks should be anonymous. Other projects will be seeing your hooks and you may not want them to know what you are doing.

The location of the extension code is not specified. Broadcom recommends maintaining a similar nexus directory structure (separate src and include dirs, include/priv subdirectory for private API), but that is not required.

Naming Convention for Module Extension Hooks

The build system hook is a path to the extension's .inc file. Here's an example of how display.inc allows itself to be extended:

```
ifneq ($(NEXUS_DISPLAY_EXTENSION_INC),)
include $(NEXUS_DISPLAY_EXTENSION_INC)
endif
```

The extension's .inc can append to NEXUS_DISPLAY_PRIVATE_INCLUDES, NEXUS_DISPLAY_PUBLIC_INCLUDES, NEXUS_DISPLAY_SOURCES, NEXUS_DISPLAY_DEFINES, and whatever else is needed.

If you want to integrate more than one extension, you must provide a single .inc that then includes all the multiple extensions.

Run-time hooks must be wrapped with a `#if` that uses a NEXUS_<MODULE>_EXTENSION_ prefix. The required hook is in nexus_<MODULE>_extensions.h. Here is an example in nexus_display_extensions.h:

```
#if NEXUS_DISPLAY_EXTENSION_PQ_CUSTOM_DYNAMIC_CONTRAST
#include "nexus_pq_custom_dynamic_contrast.h"
#endif
```

Here is an example for extending nexus_display_module.c. The following might go at the bottom of NEXUS_DisplayModule_Init:

```
#if NEXUS_DISPLAY_EXTENSION_PQ_CUSTOM_DYNAMIC_CONTRAST
    NEXUS_Display_P_PqCustomDynamicContrast();
#endif
```

From there, the extension can create and maintain state.

The `#define` is set by the extension's .inc.

Broadcom's Directory Structure

Nexus extensions can be located in any location. However, extensions that are written or managed by Broadcom are located in a central location for ease of use. That is:

```
nexus/extensions/CUSTOMER/MODULE/CHIP/extension.inc
nexus/extensions/CUSTOMER/MODULE/CHIP/include/
nexus/extensions/CUSTOMER/MODULE/CHIP/src/
```

Directory Structure

Directory	*Notes*
nexus/	Top level directory for Nexus layer
nexus/base/	Nexus Base
nexus/docs/	Nexus documentation
nexus/examples/	Simple example applications for learning the API.
nexus/modules/	Collection of modules
nexus/modules/video_decoder/	An example of one module
nexus/platforms/	Collection of common platforms
nexus/platforms/common	Common platform code
nexus/platforms/97435/	Board-specific platform code
nexus/platforms/97435/include	Board-specific platform API
nexus/platforms/97435/build	Board-specific platform build, including Makefile
nexus/platforms/97435/src	Board-specific platform implementation
nexus/utils	Command-line utilities for demo and debug
nexus/build	Makefile include files
obj.$(NEXUS_PLATFORM)	Default location for binaries

Module Directories

This is the standard directory structure for modules:

Directory	Description
modulename/	Top level directory for the module
modulename/modulename.inc	Makefile include for building the module
modulename/include/	Public API header files for module, callable by apps and other modules
modulename/ include/priv/	Private API header files for module, only callable by other modules
modulename/src/	Implementation of the module

The API and implementation are separated by directory. There is a clear distinction as to which functions are callbacks from applications and modules and which ones are private functions. Extensions, both the API and implementation, are stored in separate subdirectories. This allows the extensions to be selected or removed easily. nexus/extensions is one location used in reference software releases, but you can use your own location outside of the nexus directory structure.