



Nexus Architecture Guide

REVISION HISTORY

<i>Revision</i>	<i>Date</i>	<i>Change Description</i>
STB_Nexus-SWUM104-D2	08/27/09	Added: <ul style="list-style-type: none">• Two paragraphs to the section “Module Priorities and Base Threads ” on page 30.• The sections “Limitations on Nexus “Get” functions” on page 22, “Handle Management” on page 23, and “Settings Management” on page 24
STB_Nexus-SWUM103-R	03/16/09	Updated: <ul style="list-style-type: none">• Figure 1 on page 2• Figure 2 on page 3• Figure 3 on page 5• Figure 4 on page 6• Figure 5 on page 7• Figure 6 on page 8• Figure 7 on page 12• Figure 8 on page 16• Figure 9 on page 17• Figure 10 on page 22• Figure 11 on page 25• Figure 12 on page 25• Figure 13 on page 26• “Module Priorities and Base Threads ” on page 28
STB_Nexus-SWUM102-R	09/08/08	Added: <ul style="list-style-type: none">• “Viewing Module Priorities” on page 29 to Section 6: “Nexus Base” Updated: <ul style="list-style-type: none">• The entire Section 9: “Module Extensions” on page 33
STB_Nexus-SWUM101-R	04/25/08	Added: <ul style="list-style-type: none">• The sections “Example Code” on page 4 and “Example Diagrams” on page 5, some of which include material that was relocated for smoother logical flow.• A new section, “Execution Contexts” on page 11• A new section, Section 4: “Nexus Interfaces” on page 20, some of which includes material that was relocated for smoother logical flow Updated: <ul style="list-style-type: none">• Some text about Nexus modules in “Overview” on page 7 and in “Private Functions” on page 11
STB_Nexus-SWUM100-R	03/28/08	Initial release; based on v1.1.

Broadcom Corporation
5300 California Avenue
Irvine, CA 92617

© 2009 by Broadcom Corporation
All rights reserved
Printed in the U.S.A.

TABLE OF CONTENTS

Section 1: Introduction	1
Related Documents	1
Section 2: Nexus Overview	2
Overall Software Stack	2
Nexus Architecture	3
Example Code	4
Example Diagrams	5
Section 3: Nexus Modules	7
Overview	7
Interface to Module Mapping	8
Module State	9
Module APIs	10
Public APIs	10
Private APIs	10
Module APIs	10
Private Functions	11
Avoid Modifying Modules	11
Execution Contexts	11
Module Interconnections	12
Down Calls	13
Down Call Enforcement	13
At Compile Time	13
ISR Calls and Callbacks	13
Task Callbacks	14
Callback Descriptor	14
Requesting a Callback	14
Creating and Firing a Callback	15
Base Reentrancy	17
Application Callback Synchronization	18
Other Callback Dangers	19

Section 4: Nexus Interfaces	20
Overview	20
Reentrancy	20
Allocation	21
Interface Connections	21
Limitations on Nexus “Get” functions	22
Handle Management	23
Settings Management	24
Recommended Solutions	24
Section 5: Nexus Platforms	25
Overview	25
Typical Features	25
Driver/OS Code	25
Board Code	25
Build System	25
Application Interface	26
Enumerating Platform Features	26
Example Platform Options	27
Section 6: Nexus Base	29
Overview	29
Features	29
Events	29
Timers	29
Callbacks	29
Module Priorities and Base Threads	30
Comparison with Magnum KNI	31
Priority Inversion	31
Viewing Module Priorities	32

Section 7: Nexus Core Module	33
Section 8: Synchronization Thunks	34
Section 9: Module Extensions	35
Extension Options	35
Module Extension Convention	36
Hooks	36
Naming Convention for Module Extension Hooks.....	36
Broadcom's Directory Structure	37
Section 10: Integrating Nexus with Non-Nexus Code	38
Section 11: Directory Structure	39
Overall Structure	39
Module Directories	40

LIST OF FIGURES

Figure 1:	Nexus and the Set-top Box and DTV Software Architecture	2
Figure 2:	Components of Nexus	3
Figure 3:	Interconnected Interface Example: Low-Level PVR	5
Figure 4:	Audio/Video Separation in Decode and Display	6
Figure 5:	Two Perspectives (Internal vs. External Applications) of Nexus	7
Figure 6:	How Nexus Interfaces Map to Nexus Modules	8
Figure 7:	Examples of Module Interconnections	12
Figure 8:	Base Callback Thread	16
Figure 9:	Reentrancy in a Nexus Base	17
Figure 10:	Front End/Back End Separation	22
Figure 11:	How Platforms Differ Between Users	27
Figure 12:	Use of Nexus in a Linux Driver Module	27
Figure 13:	Use of the Automatic Nexus Interface Proxy	28

LIST OF TABLES

Table 1: OS Abstractions in Magnum and Nexus..... 31

Table 2: Nexus Directory Structure..... 39

Table 3: Nexus Module Directories..... 40

Section 1: Introduction

Nexus is a high-level, modular API for Broadcom DTV and set-top boxes. The goal of Nexus is to reduce the effort required by customers to build their system, thus shortening their time to market. 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 are still avoiding deadlock and providing maximum efficiency.

RELATED DOCUMENTS

- *Nexus Usage Guide*—available in a Nexus release bundle
- *Nexus Development Guide*—available in a Nexus release bundle
- Magnum Architecture DocJet Help—available as man pages in a reference software release bundle

Section 2: Nexus Overview

OVERALL SOFTWARE STACK

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

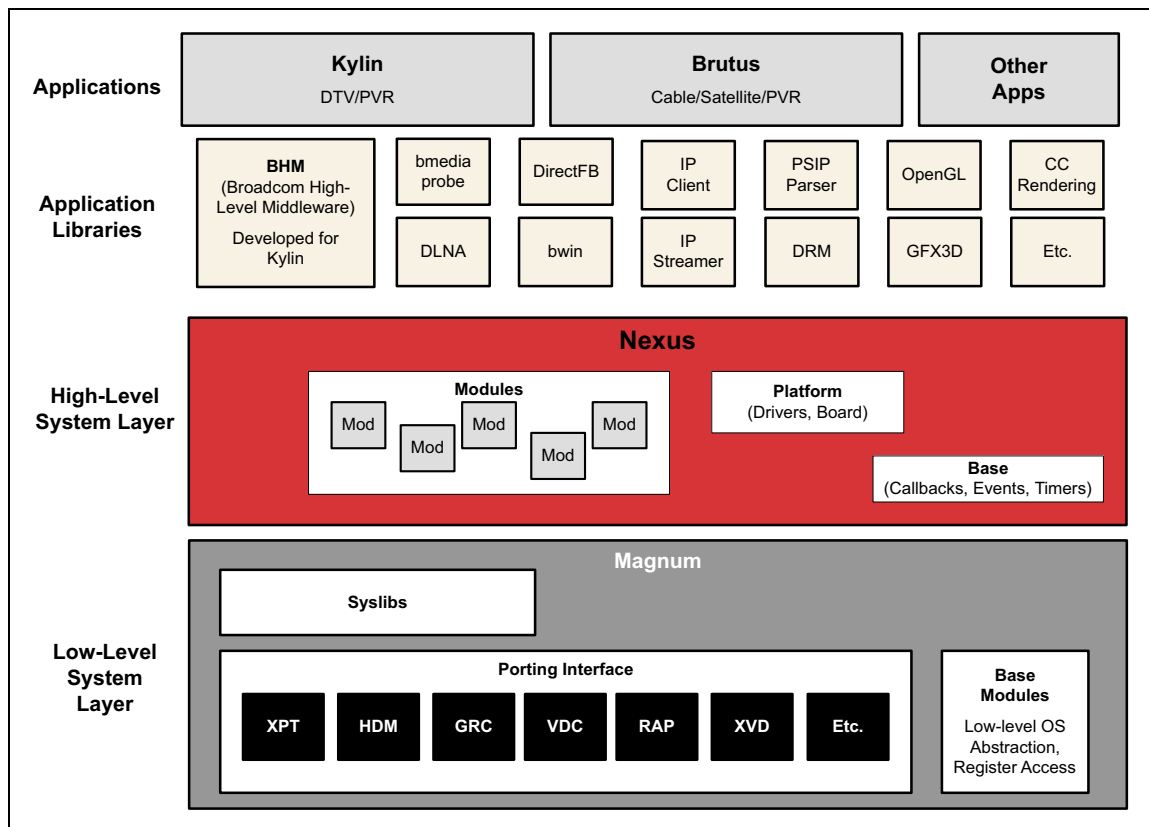


Figure 1: Nexus and the Set-top Box and DTV Software Architecture

The components of the overall software architecture are:

- **Magnum:** Low-level software modules that expose features of the hardware. Within Magnum, there are the following components:
 - **Base Modules:** Provides OS abstraction required for Magnum architecture rules, including mutexes, events, ISR critical sections, malloc, printf and related functions.
 - **Porting Interface:** A collection of independent modules which expose each HW core. This is the heart of Magnum.
 - **SysLibs:** A collection of modules that implement higher level algorithms and make some connections between modules. SysLibs must still operate within the Magnum architecture and only have access to the services provided by the Magnum Base Modules.
- **Nexus:** High-level software modules that integrate Magnum functionality and provide easy-to-use interfaces for end-user middleware and applications.

- **AppLibs:** High-level software modules that can be used in a variety of environments. Any software dependencies are minimized and explicitly documented. Many do not require Nexus. None are allowed to use Magnum directly.

NEXUS ARCHITECTURE

The components of Nexus are shown in [Figure 2](#).

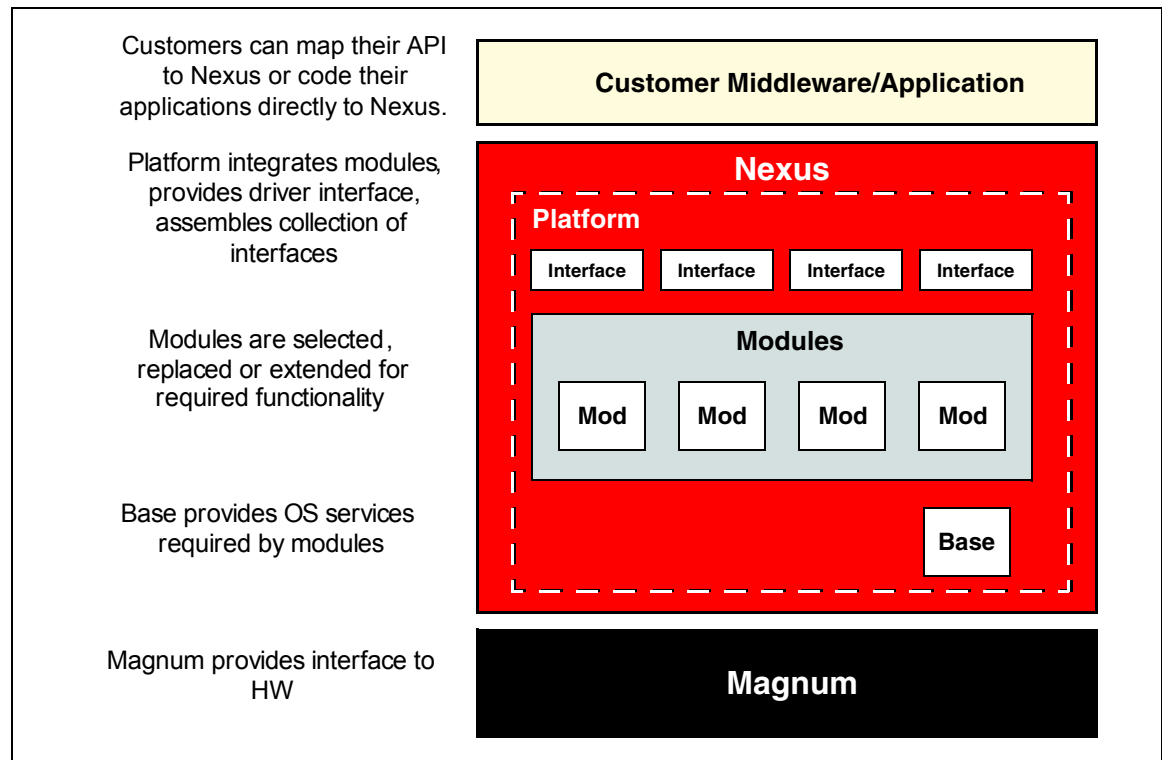


Figure 2: Components of Nexus

The components of the Nexus software architecture are:

- **Interfaces:** From the application's perspective, Nexus is a high-level API for DTVs and set-top boxes. This is represented as a collection of interconnected Nexus Interfaces. An Interface is a set of function signatures (that is, prototypes), along with their structures and enums, that present a feature.
- **Modules:** A collection of code with a common state and synchronization. It provides the implementation for one or more Nexus Interfaces.
- **Base:** Provides OS abstraction required for Nexus architecture rules, including creating tasks, requesting task callbacks for timers and events, and queuing callbacks to applications. A Nexus Base is analogous to Magnum's Base Modules, except at a higher level.
- **Platforms:** Collects all the Nexus Interfaces available on a system into a cohesive whole. It also provides OS driver code, board configuration, along with a build system. A Platform can be customized to meet a user's configuration and integration needs.

EXAMPLE CODE

A good way to begin understanding Nexus is to view a simple Nexus application, as illustrated by the following code fragment:

```
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 connection, 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 *settings
);

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 and actual interfaces, see the Nexus source tree.

EXAMPLE DIAGRAMS

The diagrams in this section are intended to provide a generic example of how interfaces are connected, rather than to depict what the functionality of each interface.

Figure 3 shows a set of interconnected Nexus Interfaces.

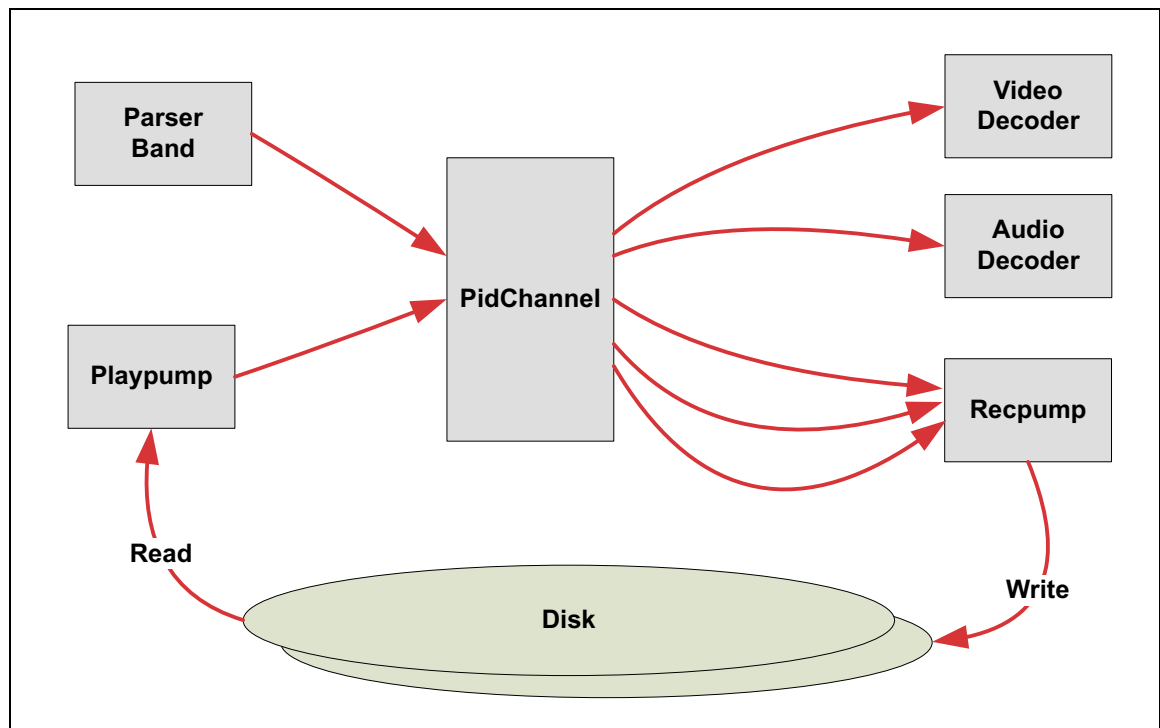


Figure 3: Interconnected Interface Example: Low-Level PVR

Each box represents a separate Nexus Interface. The direction of the arrows represents data flow.

Figure 4 shows how audio and video are handled independently from the same data source.

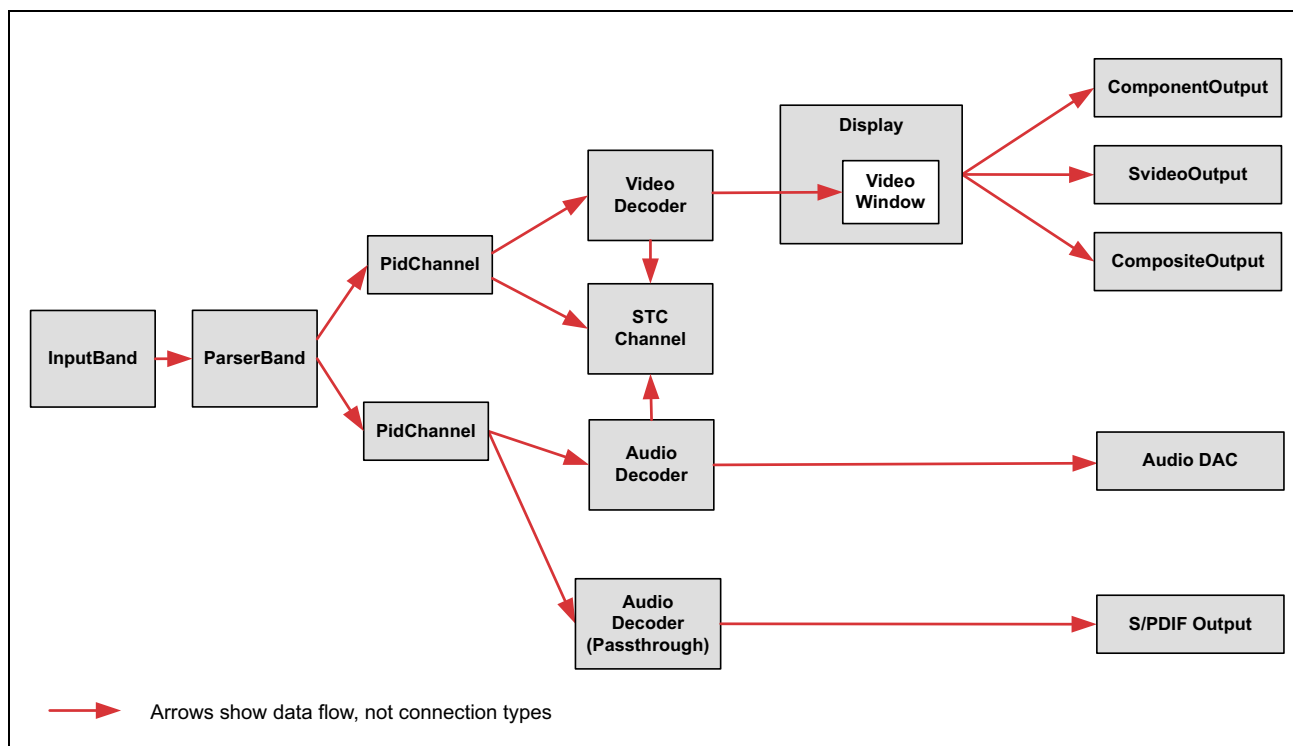


Figure 4: Audio/Video Separation in Decode and Display

For details on how specific interfaces actually work, see the *Nexus Usage Guide*.

Section 3: 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.¹

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.

Figure 5 illustrates the difference between internal Nexus Module architecture and the Interface abstraction seen by applications:

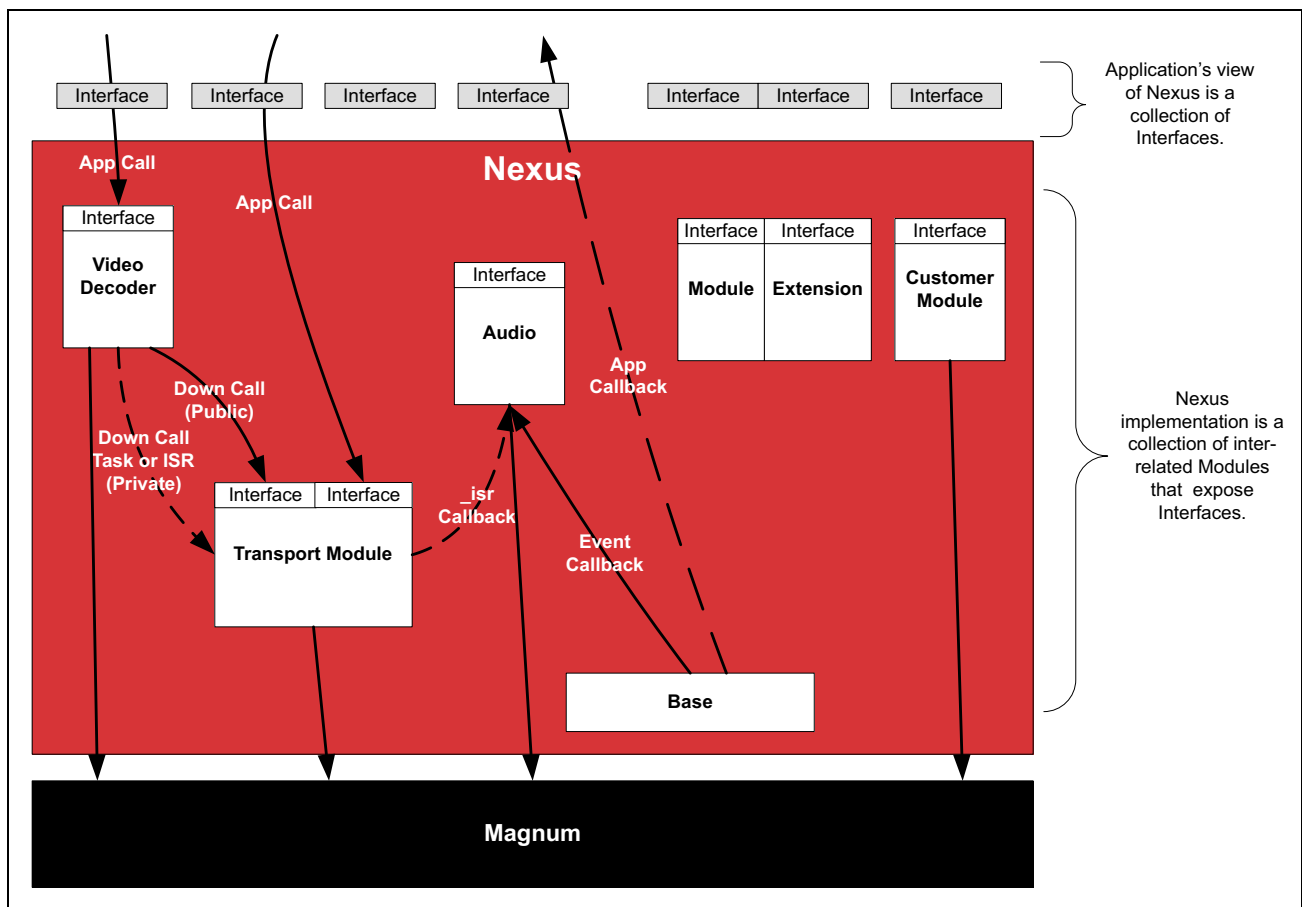


Figure 5: Two Perspectives (Internal vs. External Applications) of Nexus

1. The Core module has no public API, and therefore no Interface. It has a private API callable by other modules.

INTERFACE TO MODULE MAPPING

Figure 6 shows how Nexus Interfaces map into Nexus Modules.

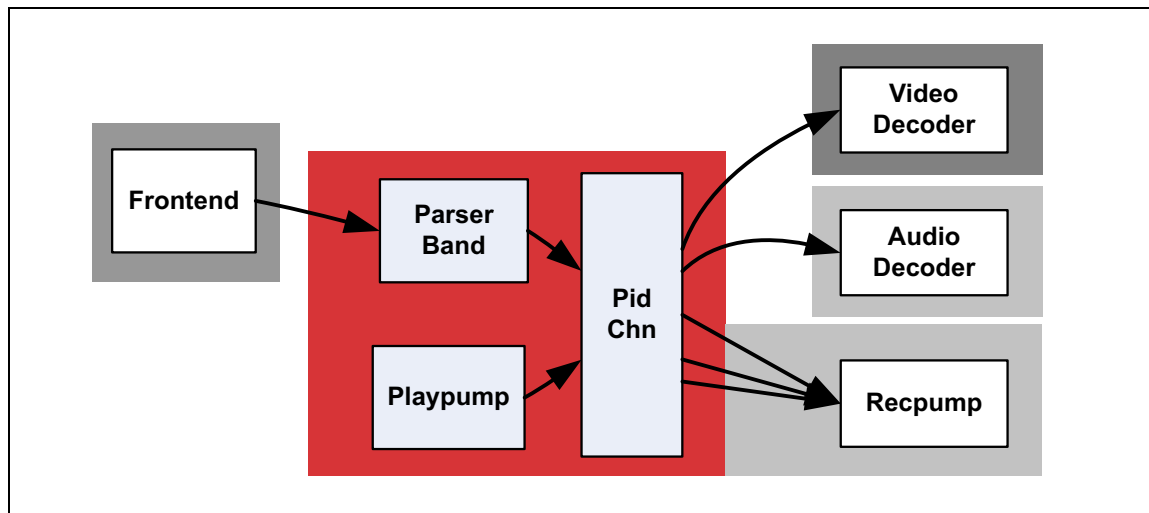


Figure 6: How Nexus Interfaces Map to Nexus Modules

The green area shows the Transport module. In this example, the Transport module provides three Nexus Interfaces². The other colored regions have only one Nexus Interface per module.

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 Guide* 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, they 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.

2. There are actually nine right now and that number may change.

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. See [Section 8: "Synchronization Thunks" on page 34](#) for more details.

The singleton paradigm still works for a system with multiple DTV or 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 Connections” section in the *Nexus Development Guide*.
- 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 considered 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 restrictions³. Task context is the normal execution context. It inherits this distinction 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.

3. 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:

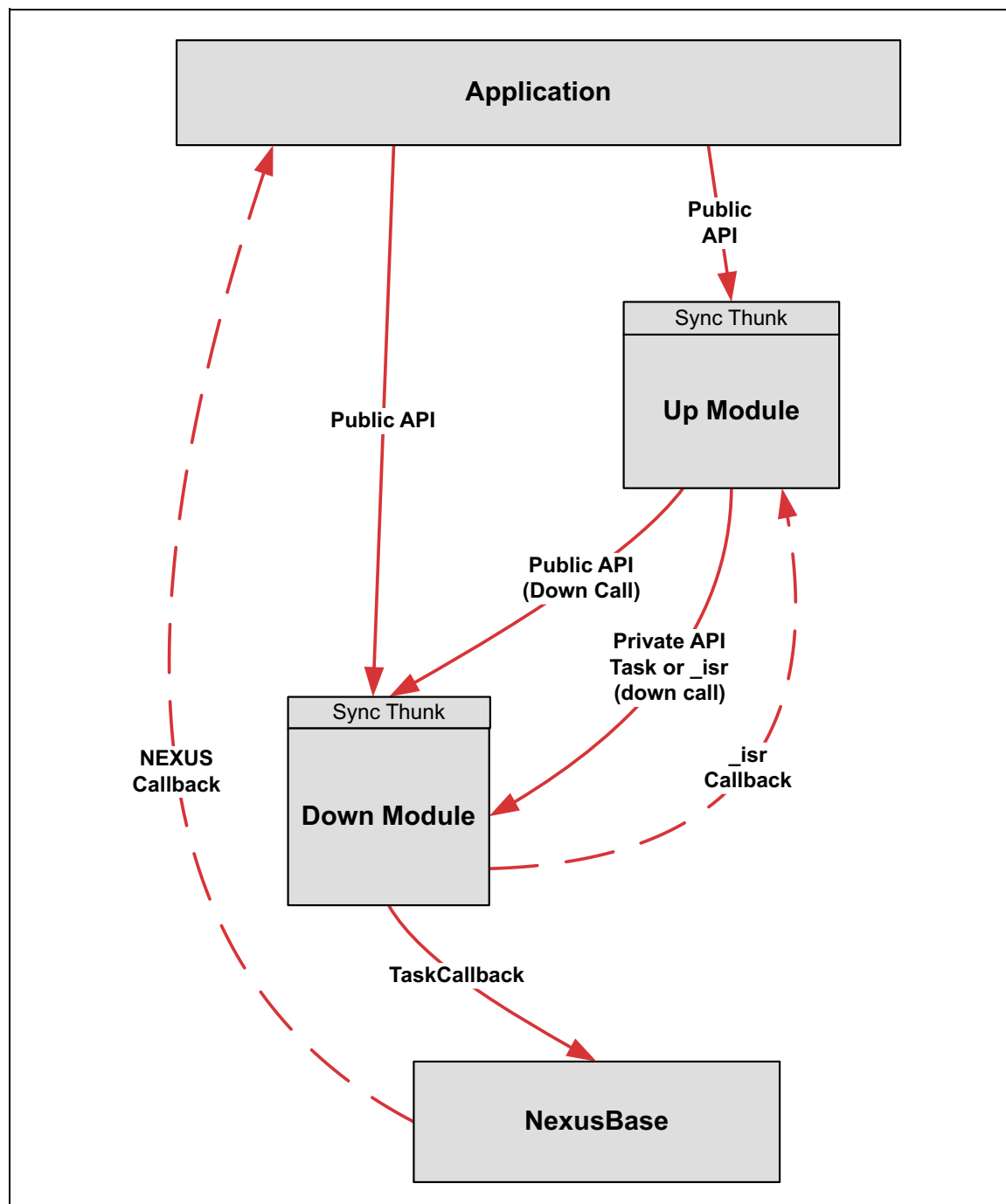


Figure 7: Examples of Module Interconnections

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

A Nexus modules can service Magnum ISR calls. All Magnum synchronization rules apply. Those rules are summarized as:

- ISR functions are identified by a “_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 module with no up/down relationship governing the ISR call. This works because there is only one ISR context in the 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.

TASK 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.⁴ 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.

If a Module requests a callback from another module, the requesting module must acquire its own module lock inside of its own callback function. Synchronization in the callback is never automatic, which means a module must never use one of its own public API functions as a callback function.⁵

-
4. Unlike Magnum callbacks, there is no 3rd "void **" parameter that allows the module to pass out data.
 5. It is not obvious why this is the case. When a module sets up a callback, if it provides a public API, the synchronization thunk will convert that into the "_impl" variety. There is no way a module can access the non-impl version of its own public API.

To make these functions more obvious, we recommend that their function names be suffixed with "_sync". They will look like this:

```
void NEXUS_Interface_SetUpCallback (NEXUS_InterfaceHandle handle)
{
    NEXUS_OtherInterface_GetSettings(handle->downmodule, &settings);
    settings.majorEvent.callback = NEXUS_Interface_P_Callback_sync;
    settings.majorEvent.context = handle;
    NEXUS_OtherInterface_SetSettings(handle->downmodule, &settings);
}
void NEXUS_Interface_P_Callback_sync(void *context, int param)
{
    NEXUS_InterfaceHandle handle = context;
    NEXUS_LockModule()
    NEXUS_Interface_P_DoWork(handle);
    NEXUS_UnlockModule()
}
```

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.

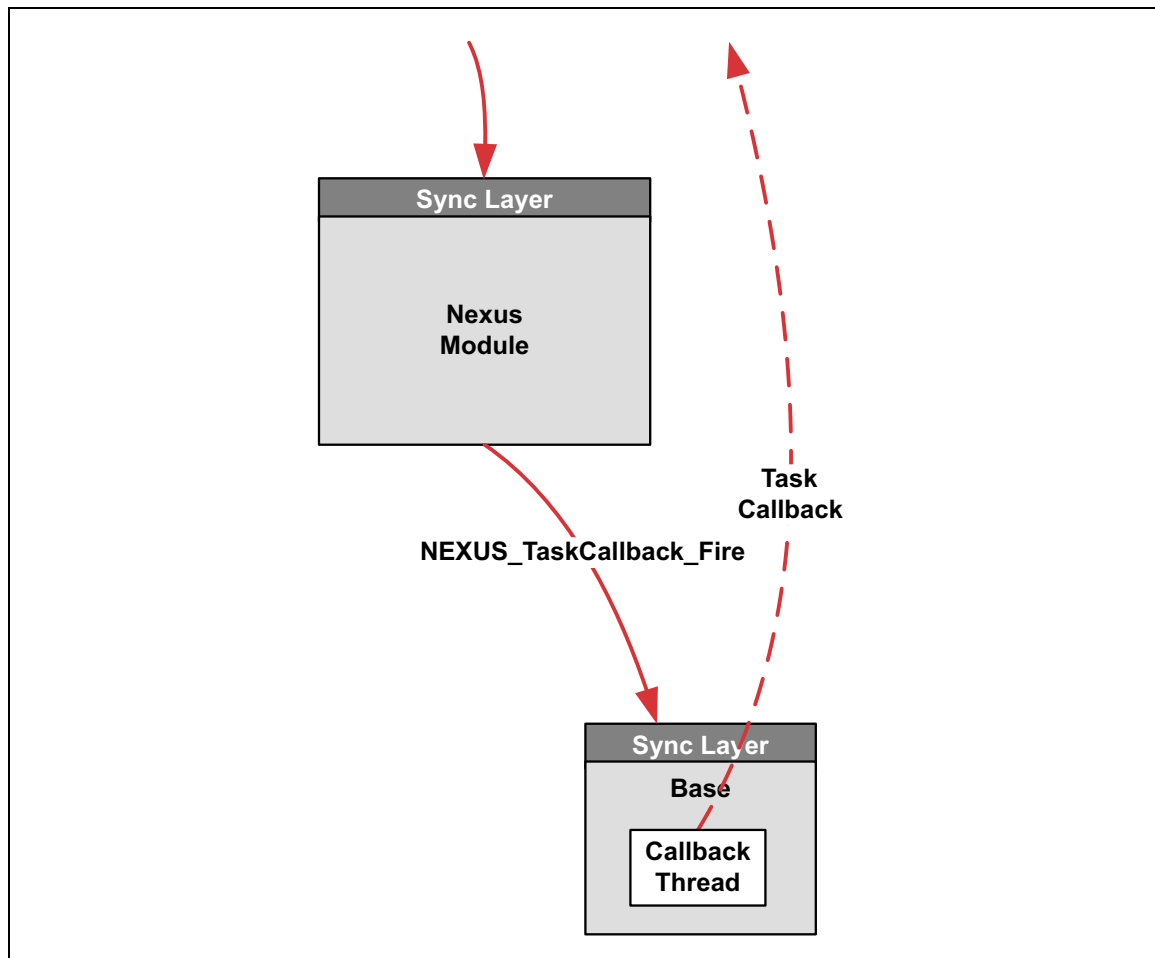


Figure 8: Base Callback Thread

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 three threads created based on module priority: high, medium and 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 it would do in response to this information is to change may be to change the format of the display. This format change can only be done at task time. 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. No loss of performance occurs.

Base Reentrancy

Figure 9 shows how callbacks cause reentrancy in a Nexus Base:

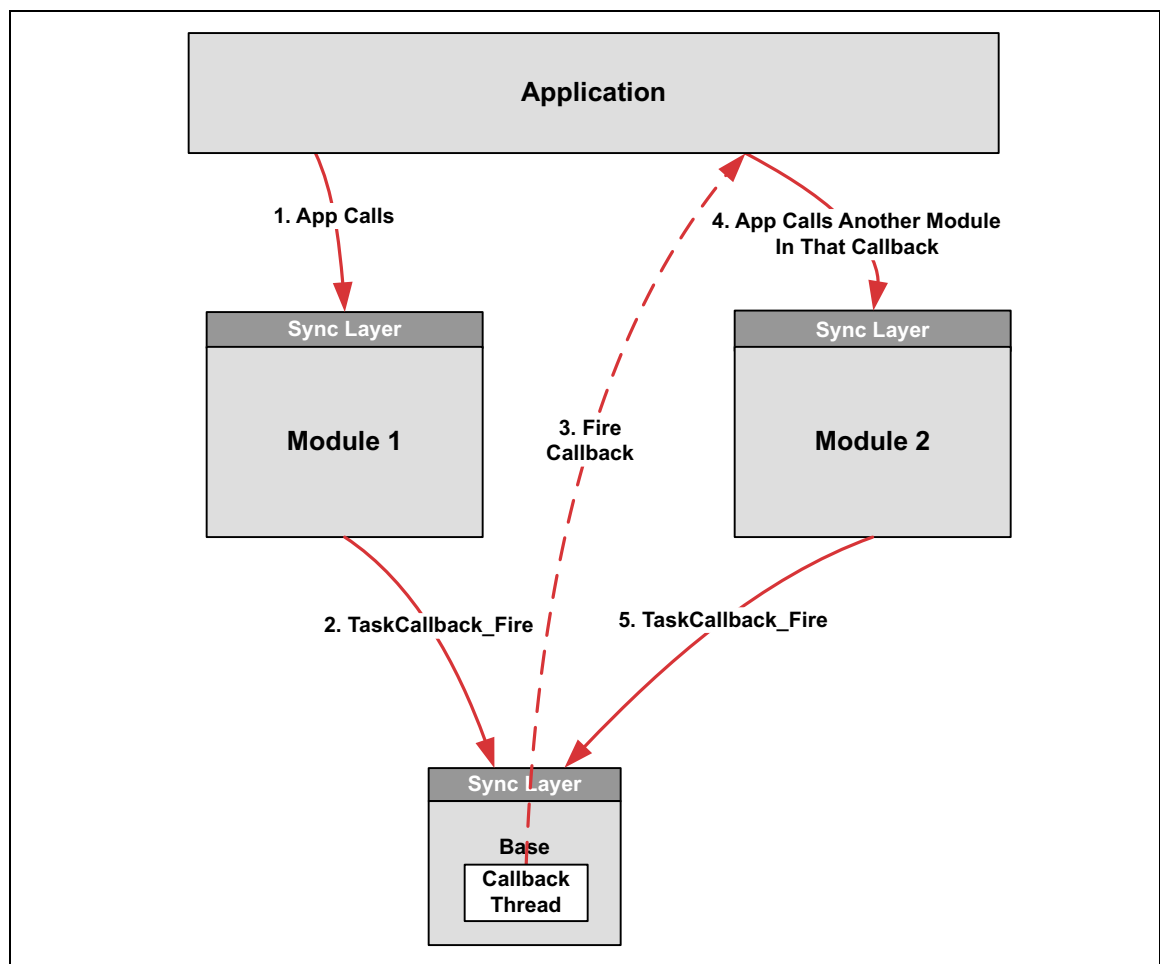


Figure 9: Reentrancy in a Nexus Base

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; otherwise the queue callback from Module 2 would deadlock. This means that Nexus Base must release its lock 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 this may happen and it is easily handled.

Application Callback Synchronization

If an application accesses Nexus handles from various threads, including callback threads, the user is responsible for ensuring that only valid Nexus handles are used. If a Nexus interface is accessed with a handle that has been closed, the system will fail⁶.

Because Nexus Base fires callbacks from internally created threads, Nexus provides an automatic way of synchronizing with those callbacks before an Interface is closed. This is handled automatically, with no user interaction required. This section documents how this is done so that you are aware of the issue.

The Nexus Core Module provides the `NEXUS_StopCallbacks(void *interfaceHandle)` function that stops the firing of all callbacks for an Interface handle and waits for any currently executing callbacks to complete. This function has been integrated with the synchronization thunk, but it could be manually used as follows:

```
void callback(void *msg, int param)
{
    NEXUS_Message_GetBuffer(msg);
    /* do work */
    NEXUS_Message_ReadComplete(msg);
}

main()
{
    msg = NEXUS_Message_Open(0, NULL);
    NEXUS_Message_Start(msg);
    /* do work */
    NEXUS_Message_Stop(msg);

    /* a Message callback may still be in progress here */

    NEXUS_StopCallbacks(msg);

    /* Nexus guarantees that there are no callbacks for the "msg"
    Handle in progress at this point. It is safe to close. */

    NEXUS_Message_Close(msg);
}
```

6. In debug mode (compiled with `DEBUG=y`), Nexus will ASSERT on any bad handle. In release mode (compiled with `DEBUG=n`), Nexus will not ASSERT but memory corruption, segfaults, or other undesired behavior will likely result.

The above call to NEXUS_StopCallbacks is not actually needed because the synchronization thunk automatically calls NEXUS_StopCallbacks before executing the Close impl function. The auto-generated synchronization thunk code looks like this:

```
void NEXUS_Message_Close(NEXUS_MessageHandle message)
{
    NEXUS_StopCallbacks(message);
    NEXUS_LockModule();
    NEXUS_Message_Close_impl(message);
    NEXUS_UnlockModule();
}
```

This technique addresses a common synchronization problem, but it cannot address all possible synchronization problems that can be introduced by user-created threads. Nexus users must be aware of their threading model and ensure that other race conditions do not result in closed handles being used.

OTHER CALLBACK DANGERS

Because Close functions automatically synchronize with their callbacks using NEXUS_StopCallbacks, Close cannot be called from inside a callback. If this is done, Nexus will deadlock. It is possible to close another interface from inside a callback, but it is not advised. Be careful to never use a handle which has been closed. If you attempt to close interfaces different threads or contexts, this rule is very hard to ensure. We recommend that all Opens and Closes be done from a main application thread.

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.

Section 4: Nexus Interfaces

OVERVIEW

An Interface is a set of function signatures (that is, prototypes), along with their structures and enums, that 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. See the [Section 8: "Synchronization Thunks" on page 34](#) for how internal synchronization is performed.

Applications can make Nexus calls from within Nexus callbacks⁷. 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.

7. 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.

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. Open cannot be called more than once for the same index.

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 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.

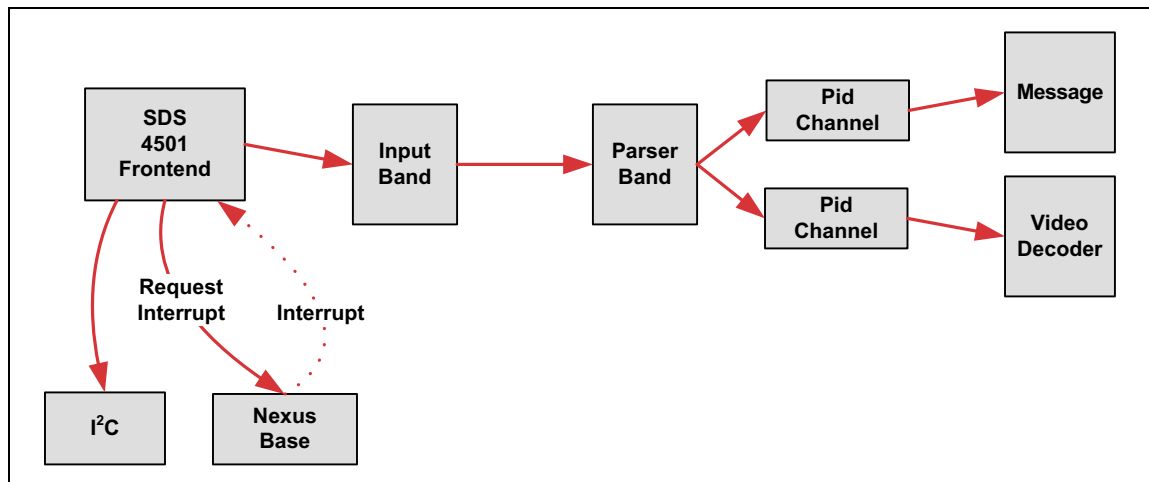


Figure 10: Front End/Back End Separation

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 the *Nexus Development Guide* 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 deallocate 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 this⁸. See the *Nexus_Development Guide* for details on parameter validation.

8. 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.

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:

$$[A] \rightarrow [X] \leftarrow [B]$$

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:

$$[AB] \rightarrow [X]$$

- **Arbitration:** There should be some application-specific arbiter X' that encapsulates and controls access to X. That is:

$$[A] \rightarrow [X' [X]] \leftarrow [B]$$

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.

Section 5: 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 later 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 decide 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. nexus.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_PlatformInit(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/97405/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.

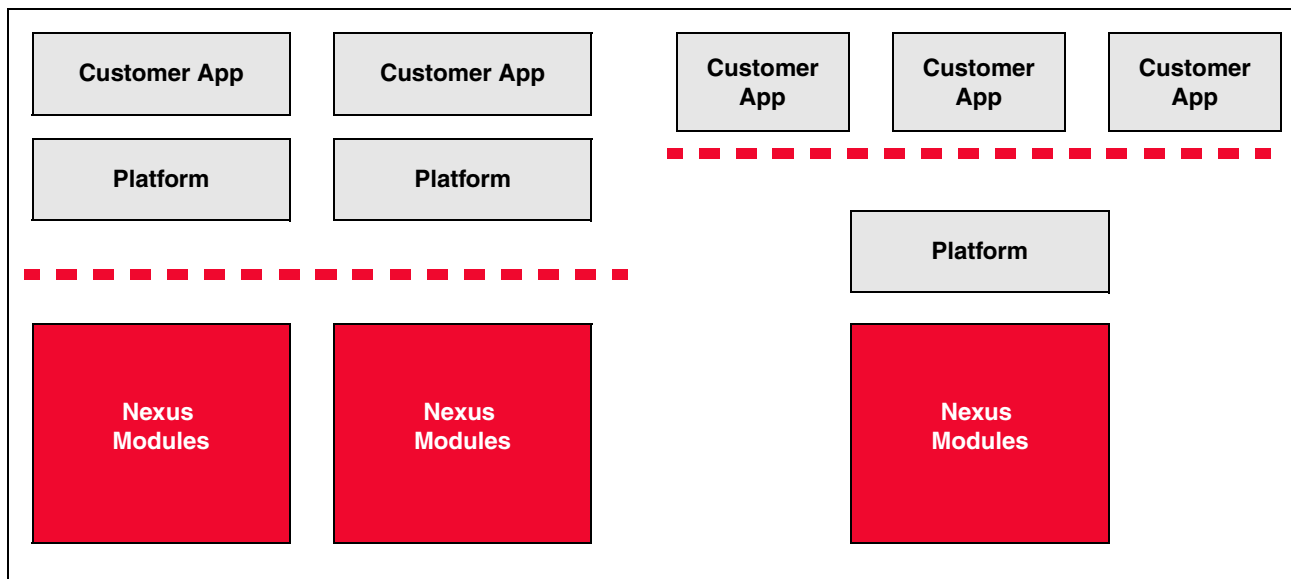


Figure 11: How Platforms Differ Between Users

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.

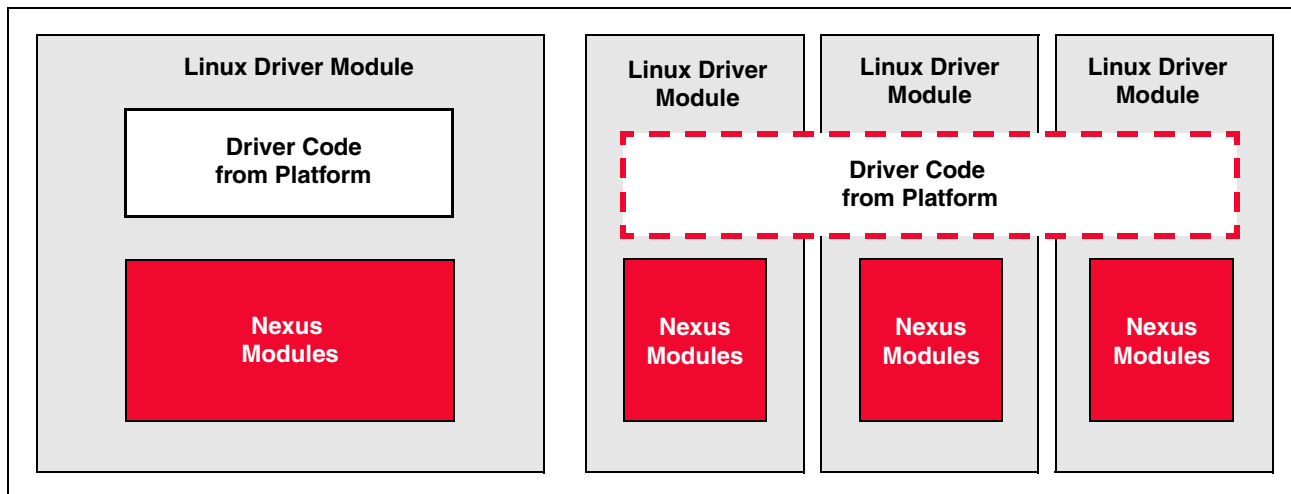


Figure 12: Use of Nexus in a Linux Driver Module

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.

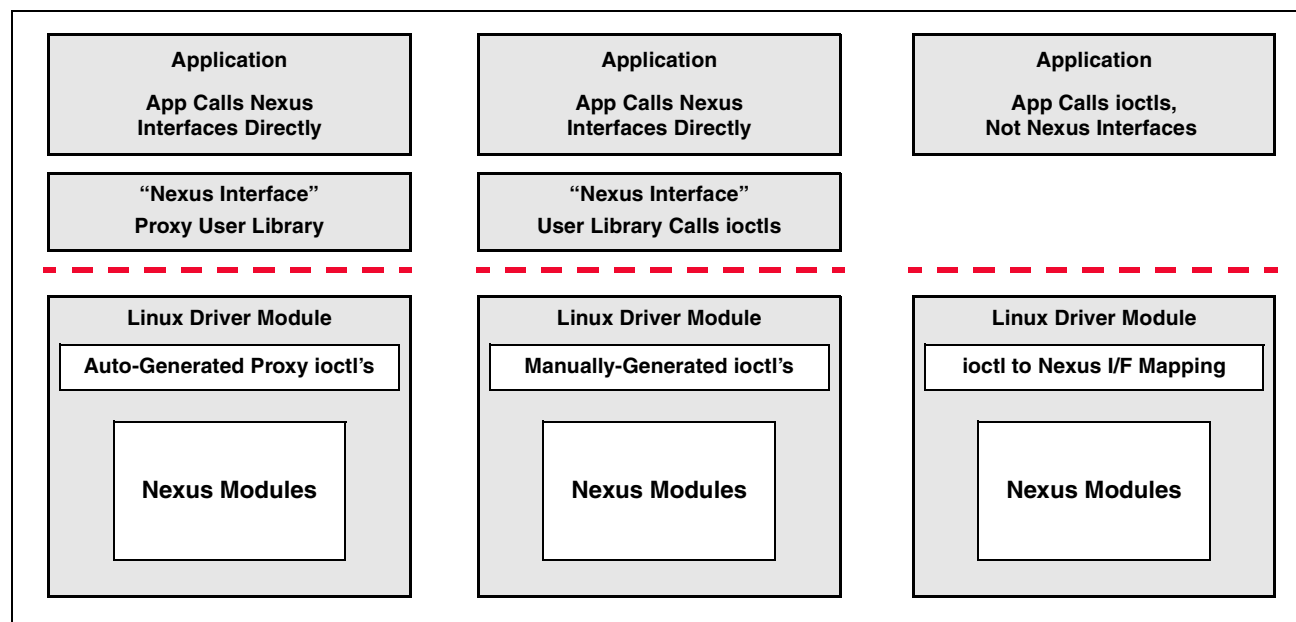


Figure 13: Use of the Automatic Nexus Interface Proxy

Section 6: 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 re-entrant in many places. It has functions that require module handles that cannot be apart 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).

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.

See [“Task Callbacks” on page 14](#) for more information.

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)
- Medium (Example module: display)
- Low (Example module: video_decoder, audio)
- Idle (Example module: i²c, frontend)

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/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" on page 29 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.

Section 7: Nexus Core Module

- The Nexus Core module owns all the Magnum Base Module handles. These include:
- Register Interface (REG)
- Chip Interface (CHP)
- Memory heaps (MEM)
- Interrupt Interface (INT)

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.

Section 8: Synchronization 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.

Every Nexus Interface has a synchronization thunk which acquires the module’s mutex. The thunk layer automatically acquire the module's mutex, calls into the implementation, then release mutex before returning to the caller.

The synchronization thunk looks like this:

- **module/CHIP/src/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;
}
```

- **module/CHIP/src/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.

This follows the “Monitor” pattern, where implicit synchronization is transparently added to an API with some compiler/preprocessor tool.

The Nexus Platform Makefile should auto-generate the synchronization thunk layer for every Nexus Module's public API. This removes potential developer oversight or error for synchronization. It also makes the internal code easier to read. We will recommend that Perl be used in the build system for this autogen. Reference Perl thunk generation code will be provided.

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.

Section 9: 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. For example, Broadcom implemented one version of the VideoDecoder for MVD systems (such as the BCM3563) and another version for XVD systems (BCM740x, BCM3548, etc.).
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/
```

For instance:

```
nexus/extensions/LarrysTvs/display/3548/display_ext.inc
nexus/extensions/LarrysTvs/display/3548/include/
nexus/extensions/LarrysTvs/display/3548/src/
```

Section 10: Integrating Nexus with Non-Nexus Code

Nexus is not an all-or-nothing solution. Users may have mature drivers that they prefer to continue to use for existing features, while taking Nexus to use new features. For example:

- User keeps front-end and transport, but adds Nexus decode and display in order to pick up mosaic mode feature.
- User keeps display and audio, but adds Nexus transport and decoder in order to pick up the PVR feature.

The rules for doing this are:

- The application may not call any Magnum PI that is owned by Nexus. There must be strict PI ownership.
- The application must provide a Nexus Base and Core Module which Nexus Modules use. The Core module does not need to own the Magnum Base Modules because they are re-entrant. The non-Nexus code can create the Base Modules and give them to the Core module. Both Nexus and non-Nexus code can call the same Base Modules.
- The application must provide all public and private APIs which the Nexus Modules require. This can be easily discovered by compiling the code and seeing which symbols are unresolved.

Actually implementation may not be trivial. There may be multiple APIs to implement.

Section 11: Directory Structure

OVERALL STRUCTURE

The overall Nexus directory structure is given in [Table 2](#).

Table 2: Nexus Directory Structure

Directory	Notes
nexus/	Top level directory for Nexus layer
nexus/base/	Nexus Base
nexus/base/include/	Nexus Base API (callable by modules)
nexus/base/src/	Nexus Base implementation
nexus/docs/	Nexus documentation
nexus/examples/	Simple example applications for learning the API.
nexus/modules/	Collection of core modules
nexus/modules/video_decoder/	A module; see Section 3: "Nexus Modules" on page 7 for internal module directory structure.
nexus/modules/audio/	Another module.
nexus/platforms/	Collection of common platforms
nexus/platforms/97401/	Reference set-top platform.
nexus/platforms/93563/	Reference DTV platform.

MODULE DIRECTORIES

This is the standard directory structure for modules:

Table 3: Nexus Module Directories

Directory	Description
modulename/	Top level directory for the module
modulename/CHIP/	Subdirectory for the module which allows long-term versioning for chips. This could be something other than chip, but is usually chip. Clearcase directory symlinks allowed shared code.
modulename/CHIP/modulename.inc	Makefile include for building the module
modulename/CHIP/include/	Public API header files for core module
modulename/CHIP/include/priv/	Private API header files for core module
modulename/CHIP/src/	Implementation of the module
modulename/CHIP/xxx_ext/	Extension “xxx”
modulename/CHIP/xxx_ext/xxx_ext.inc	Makefile include for building the extension
modulename/CHIP/xxx_ext/include/	Extension “xxx” public API header files
modulename/CHIP/xxx_ext/include/priv/	Extension “xxx” private API header files
modulename/CHIP/xxx_ext/src/	Extension “xxx” implementation

Like Magnum, a CHIP subdirectory is used to allow for major changes in implementation, and possibly API, when moving to a new chip. Broadcom’s source control system allows the sharing of code between the CHIP subdirectories when possible.

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.

Broadcom Corporation

5300 California Avenue
Irvine, CA 92617
Phone: 949-926-5000
Fax: 949-926-5203

Broadcom Corporation reserves the right to make changes without further notice to any products or data herein to improve reliability, function, or design.
Information furnished by Broadcom Corporation is believed to be accurate and reliable. However, Broadcom Corporation does not assume any liability arising out of the application or use of this information, nor the application or use of any product or circuit described herein, neither does it convey any license under its patent rights nor the rights of others.