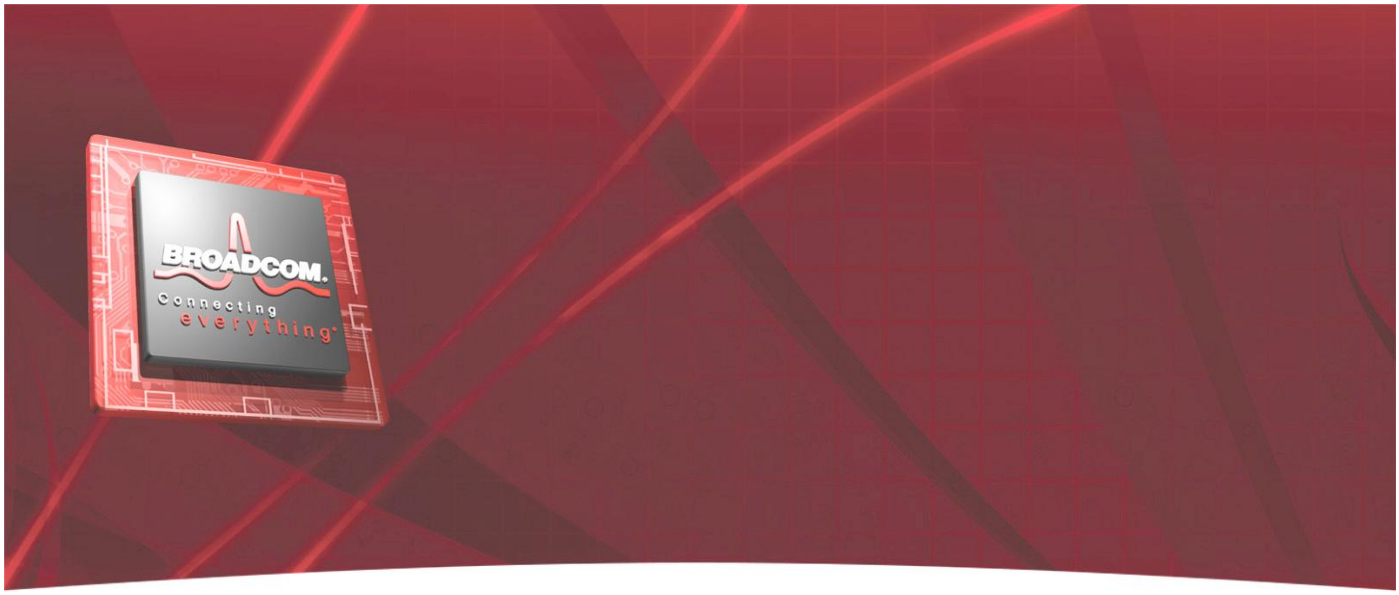


Nexus Development Guide

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

Web: www.broadcom.com

Revision History

Revision	Date	Change Description
1.0	5/30/2008	Initial version (STB_Nexus-SWUM300-R)
1.1	9/22/2008	Structure example updated (STB_Nexus-SWUM301-R)
1.2	7/9/2009	Update (STB_Nexus-SWUM302-R)
1.3	8/10/2011	Rework

Table of Contents

Introduction

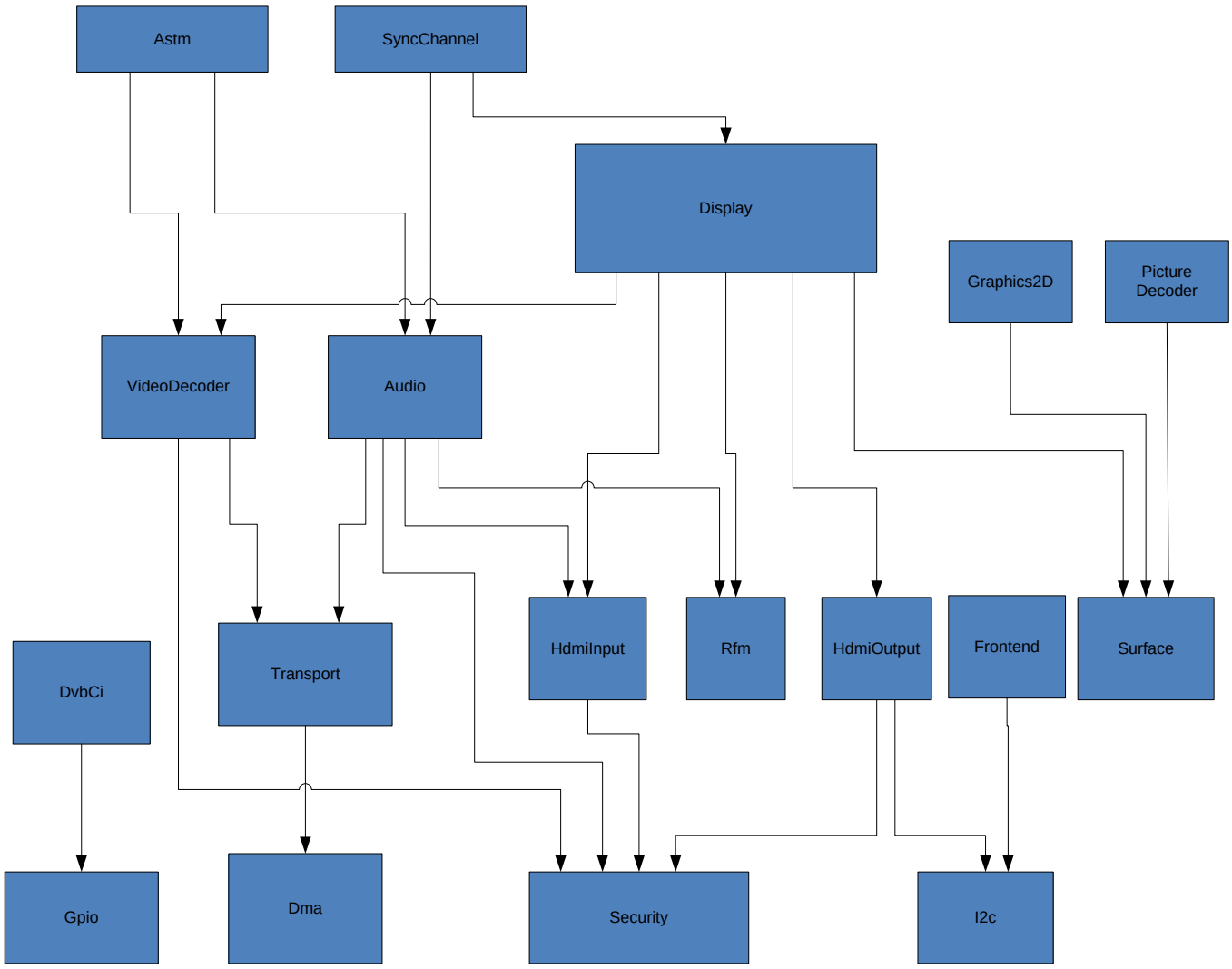
Nexus is a high-level, modular API for Broadcom digital TV and set-top boxes. This document describes how to develop new Nexus Modules and Platforms.

Before reading this document, see nexus/docs/Nexus_Architecture.pdf to learn foundational concepts.

This document does not provide function-by-function API documentation, nor does it contain information on how to write Nexus applications. See nexus/docs/Nexus_Usage.pdf for that information.

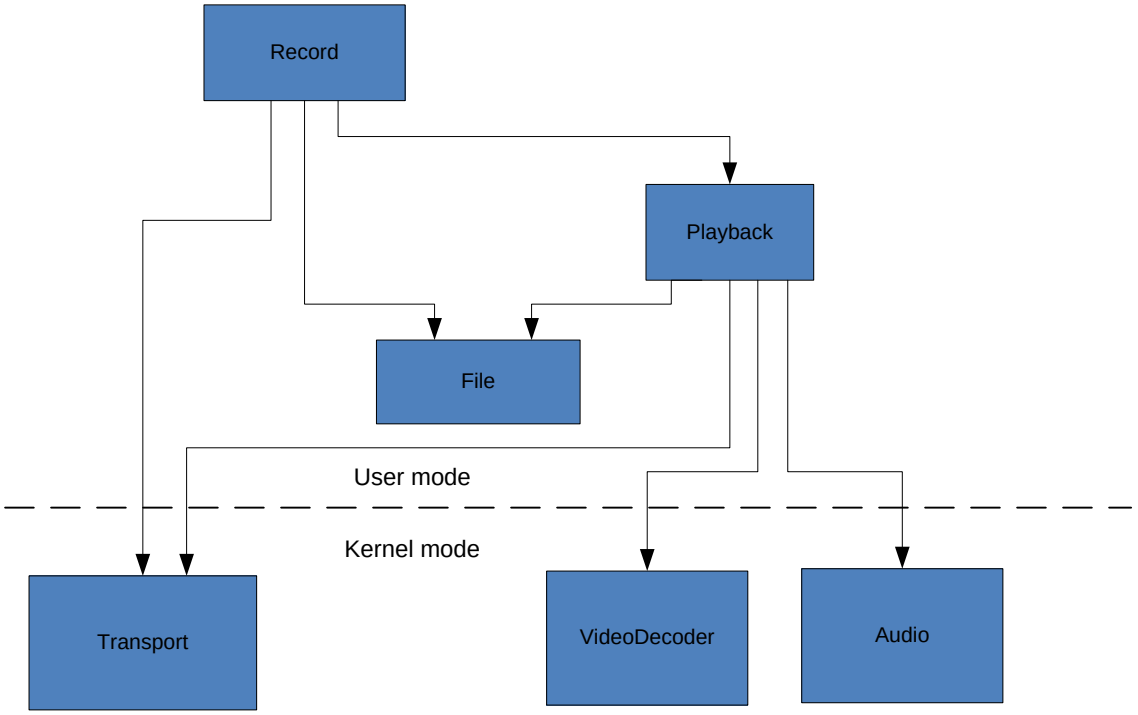
Nexus Module Tree

The following is a hierarchical relationship of typical set-top box Nexus modules that would run in the kernel.



The Nexus Core module is not shown here. The Core module owns all the Magnum Base Module handles, including the Register Interface (REG), the Chip Interface (CHP), the Memory heaps (MEM), and the Interrupt Interface (INT).

The following is a hierarchical relationship of typical set-top box Nexus modules that would run in user space, even if the rest of nexus was running in the kernel.



Code Quality Standards

Broadcom uses the following checks to ensure Nexus source code stays in a consistent and correct state:

- Compile and run in both debug mode (`export B_REFSW_DEBUG=y`) and release mode (`export B_REFSW_DEBUG=n`)
- Compile and run in both Linux user and kernel modes
  - Linux kernel mode is an easy way to find any improper libc calls in Magnum or Nexus.
- Compile with no warnings in all modes
- Pass Coverity static analysis (see www.coverity.com)
- Pass the nexus_codecheck.pl tool for all public APIs
- Pass the nexus_commentcheck.pl tool for all public APIs
- No KNI memory leaks or MEM leaks after a thorough runtime test
  - Both checks are enabled by default in debug mode. NEXUS_Platform_Uninit must be called.
- Pass the BSEAV/tools/build/check_global_symbols test
  - This ensures there are no global symbols with improve namespace prefixing.
- Pass the BSEAV/tools/build/check_global_data test
  - This ensures there are no global variables in magnum.
- Run a periodic MEM heap validation.
  - `export debug_mem=y`

Checking Coding Conventions

Broadcom provides a Perl script that checks the API header files (public and private) for consistency with the Nexus coding conventions, which are described in Coding Convetions. Please run this tool on the API and make adjustments until no ERRORs are reported.

Example:

```
cd nexus/build/tools/common
./nexus_codecheck.pl ../../../modules/video_decoder/7400/include/*
./nexus_codecheck.pl
../../../modules/video_decoder/7400/include/priv/*

./nexus_codecheck.pl `find ../../../modules/*/*/include/*.h`
```

Note: This tool is still in development and may detect false errors. Please notify the Nexus team of any doubtful warnings.

Validating Handles

All Nexus handles should be validated with the BDBG_OBJECT tag. See magnum/basemodules/dbg/bdbg.h for the API. This technique verifies the following conditions:

- The handle is not NULL or some other bad value
- The handle is the correct type (not some other handle)
- The handle has not already been closed

After allocating the instance, `BDBG_OBJECT_SET` or `BDBG_OBJECT_INIT` should be used to mark the object. If you use `BDBG_OBJECT_SET`, you should previously `BKNI_Memset` the instance to zero to ensure deterministic behavior of the interface. For example:

```
NEXUS_InterfaceHandle NEXUS_Interface_Open()
{
    NEXUS_InterfaceHandle handle = BKNI_Malloc(sizeof(*handle));
    BKNI_Memset(handle, 0, sizeof(*handle));
    BDBG_OBJECT_SET(handle, NEXUS_Interface);
    return handle;
}
```

If you use `BDBG_OBJECT_INIT`, all memory will be set to a non-zero pattern that is also deterministic. For example:

```
NEXUS_InterfaceHandle NEXUS_Interface_Open()
{
    NEXUS_InterfaceHandle handle = BKNI_Malloc(sizeof(*handle));
    BDBG_OBJECT_INIT(handle, NEXUS_Interface);
    return handle;
}
```

Similarly, there must be either an UNSET or DESTROY in the Close. DESTROY is preferred if the memory is to be deallocated:

```
NEXUS_Interface_Close(NEXUS_InterfaceHandle handle)
{
    BDBG_OBJECT_DESTROY(handle, NEXUS_Interface);
    BKNI_Free(handle);
}

NEXUS_Interface_Close(NEXUS_InterfaceHandle handle)
{
    BDBG_OBJECT_UNSET(handle, NEXUS_Interface);
    handle->open = false; /* static allocation */
```

```
}
```

Inside every function, BDBG_OBJECT_ASSERT should be used to verify the handle before use:

```
NEXUS_Error NEXUS_Interface_Foo(NEXUS_InterfaceHandle handle)
{
    BDBG_OBJECT_ASSERT(handle, NEXUS_Interface);
    do_work(handle->x);
    return 0;
}
```

If the handle is NULL, incorrect, or already closed, the application will immediately segfault. Use gdb to get a stack trace from the core dump, then find and correct the problem.

Validating Enums

All enums should be validated using range checks or switch statements with default failure clauses.

```
NEXUS_Error NEXUS_Interface_Foo(handle, NEXUS_InterfaceEnum enum)
{
    switch (enum) {
    case NEXUS_InterfaceEnum_e0: do_work(); break;
    case NEXUS_InterfaceEnum_e1: do_work(); break;
    case NEXUS_InterfaceEnum_e2: do_work(); break;
    default: return BERR_TRACE(NEXUS_INVALID_PARAMETER);
    }
    return 0;
}

NEXUS_Error NEXUS_Interface_Foo(handle, NEXUS_InterfaceEnum enum)
{
    if (enum >= NEXUS_InterfaceEnum_eMax) {
        return BERR_TRACE(NEXUS_INVALID_PARAMETER);
    }
    return do_work(handle->data[enum]);
}
```

Validating Non-Handle Pointers

Broadcom recommends but does not require that non-handle pointers be validated. Asserting that pointers are not NULL provides some value, only by spotting the error early. If a pointer is NULL, the application will likely segfault immediately after the check anyway. For instance:

```
void NEXUS_Interface_GetSettings(NEXUS_InterfaceHandle handle,
NEXUS_InterfaceSettings *pSettings)
{
    BDBG_OBJECT_ASSERT(handle, NEXUS_Interface);
    BDBG_ASSERT(pSettings); /* This would segfault on the next line
        anyway. */
    pSettings->x = handle->x;
}
```

The goal of parameter validation is to ensure immediate and deterministic failure for bad data during development. Dereferencing zero is already immediate and deterministic.

Improper use of BDBG_ASSERT

BDBG_ASSERT is a powerful way for developers to validate the internal consistency of their code. It's also helpful to fail early and consistently when eventual failure is certain (for example, a corrupt pointer).

BDBG_ASSERT should not be used to validate external API usage. This includes the following:

- Enum checking (for example, a user passes an integer instead of a valid enum.)
- Range checking on integers
- Any type of external data validation (for example, scanning memory buffers for key values)
- Call ordering (for example, a user must call "Start" because calling "Pause")

In general, no user should be able to cause a BDBG_ASSERT inside Nexus or Magnum unless they are passing invalid handles and would have crashed anyway.

Validating Memory Allocation

Nexus code must check whether BKNI_Malloc and BMEM_Alloc return NULL. Running out of memory is normal behavior in some systems. Nexus should return an error code in this case. It should not assert or crash because it runs out of memory.

Validating Return Codes

Nexus code must capture all return codes and use BERR_TRACE to print an error message (with file and line number) on failure. If a failure return code does not result in a different code path, a comment should acknowledge that this is intended.

```
NEXUS_Error NEXUS_Interface_Foo (handle)
{
    BERR_Code rc;

    rc = call_func1();
    if (rc) return BERR_TRACE(rc);

    rc = call_func2();
    if (rc) BERR_TRACE(rc); /* fall through */

    return 0;
}
```

Validating Module Synchronization

All _priv functions should assert that the caller has properly acquired the module lock. For example:

```
void NEXUS_Interface_Foo_priv(handle)
{
    NEXUS_ASSERT_MODULE();
    do_work();
}
```

Interface Patterns

When developing the most common Interfaces, the Nexus architecture team considered the use of standard design patterns. Patterns help make the system is more understandable and usable. There was not a one-size-fits-all approach to interface design, but we did attempt to solve similar problems with similar APIs.

General API pattern

Most interfaces have a core set of functions for managing life cycle, core settings and connectors. In most cases these functions are as follows:

- NEXUS_Interface_Open and NEXUS_Interface_Close
- NEXUS_Interface_Create and NEXUS_Interface_Destroy
- NEXUS_Interface_GetSettings and NEXUS_Interface_SetSettings
- NEXUS_Interface_GetBuffer and NEXUS_Interface_ReadComplete
- NEXUS_Interface_AddInput and NEXUS_Interface_RemoveInput
- NEXUS_Interface_GetConnector
- NEXUS_Interface_Shutdown

See the nexus/docs/Nexus_Usage.pdf for a discussion of using these common APIs.

Enumerated resources

Some resources have no software allocation and simply expose hardware resources. There is often a Get/SetSettings but no Open or Create function. You can just use the resource. There may be an enable boolean in the settings structure, but this enable usually refers to hardware activation, which affects memory bandwidth, not memory allocation.

An example is:

```
typedef enum NEXUS_Timebase
{
    NEXUS_Timebase_e0,
    NEXUS_Timebase_e1,
    NEXUS_Timebase_e2,
    NEXUS_Timebase_e3,
    NEXUS_Timebase_eMax
} NEXUS_Timebase;

void NEXUS_Timebase_GetSettings(
     NEXUS_Timebase timebase,
```

```
      NEXUS_TimebaseSettings *pSettings /* [out] */
      );

NEXUS_Error NEXUS_Timebase_SetSettings(
      NEXUS_Timebase timebase,
      const NEXUS_TimebaseSettings *pSettings
      );
```

The number of enumerated resources is usually determined by #define's in nexus_platform_features.h. Some common #defines are NEXUS_NUM_INPUT_BANDS, NEXUS_NUM_PLAYPUMPS, NEXUS_NUM_VIDEO_WINDOWS, etc.

See the Memory Management section of nexus/docs/Nexus_Usage.pdf for a discussion of how overall system memory should be managed by customizing nexus_platform_features.h.

Connection Patterns

Chained Interface

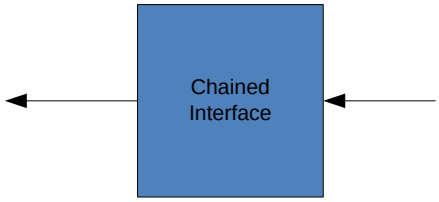


Figure 2: Chained Interface

A chained Interface, shown in Chained Interface, allows a series of one-way connections from destination to source. Destination Interfaces know about the source it's connected to, but know nothing of who is connected to them.

Changes in a destination can be applied immediately to its source. Changes in a source must be sent via callback to any destination. Callback options could be ISR/task if there's a private connection, or must be queued through based if there's only a public connection.

Examples:

- Connect a VideoDecoder to a PidChannel by setting NEXUS_VideoDecoderStartSettings.pidChannel
- Connect a VideoWindow to a VideoInput by calling NEXUS_VideoWindow_AddInput
- Connect a ParserBand to an InputBand by setting NEXUS_ParserBandSettings.sourceTypeSettings.inputBand

Connection Interface

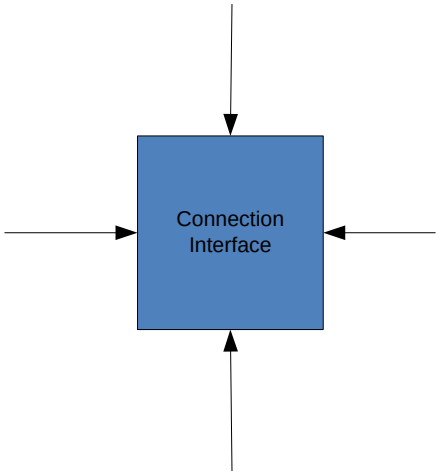


Figure 3: Connection Interface

A connection Interface, shown in Connection Interface, knows nothing about who is connecting to it. It provided a basic service and is used by other Interfaces.

Any change in this Interface must be propagated by means of callbacks.

Examples of Connection Interfaces are:

- PidChannel
  - Used by Playpump, Recpump, VideoDecoder, AudioDecoder, etc.
- Surface
  - Used by Graphics2D, Display
- VideoInput, AudioInput
  - This is an abstract Interface which allows specific inputs to be connected in various ways to aggregators.
- HdmiOutput
  - Although HDMI is not a passive output (like an analog output), it does not directly effect change in the system.
  - Hotplug is detected, then the app is notified if EDID capabilities or preferences need to effect change.
  - Display format is set in the VEC and HDMI must simply output what it is given. An HDMI call never changes the VEC.
- Rfm
  - Similar to HdmiOutput

Aggregator Interface

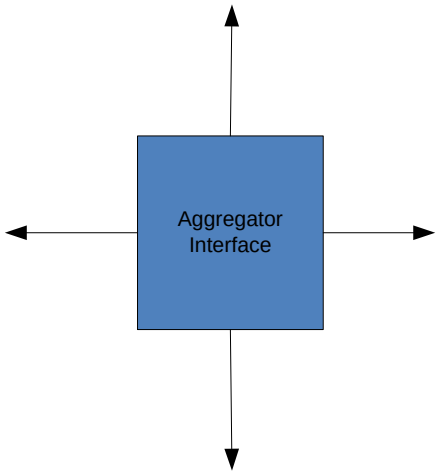


Figure 4: Aggregator Interface

An aggregator Interface, shown in Aggregator Interface, knows about a collection of Interfaces which it manages. It has the ability to set part of their state. This makes configuration management easy for an application. The app can set a mode in the aggregator, and the aggregator can validate the state in light of the various connections, then immediately make the changes to all connected Interfaces.

The downside is that the aggregator must have specific knowledge of its connections. If a new connection type is added, the aggregator must be modified to support it. This is often done with a small amount of code added in a typical location, possibly with #ifdefs.

Examples:

- Display
  - Aggregates video windows, graphics surfaces, and video outputs.
  - Controls muting, z-order, display format validation on video outputs.
- AudioMixer
  - Aggregates audio inputs and audio outputs.
  - Controls volume, muting, audio format validation on audio outputs.

Audio/Video Connections

Overview

When designing the audio and video subsystems of Nexus, Broadcom saw the need for an abstract connector to facilitate building complex filter graphs between interfaces from different modules.

The requirements were:

- A Producer should not know about a consumer. A consumer may know about a connected producer.
  - Strict up/down module relationships require that we cannot have cross-dependency.
- Producer and consumer can be in the same or different Nexus Modules.
  - Also, the modularity of some interfaces may change over time. For instance, in today's silicon the AnalogVideoDecoder is in the same module as Display. That may change for future silicon.
- The application should not be required to create and manage additional handles for each connection.
  - This creates a lot of burdensome complexity.
- The connection should be capable of holding state information which is preserved between re-connections.
- It's common to adjust a filter graph by disconnecting from one point and reconnecting to another point. State information should not be lost in that transition.

This abstraction requires polymorphism. Nexus is implemented in C, which means we can't make use of object-oriented language constructs like base classes, virtual functions and run-time type checking. We have avoided typecasting (such as passing objects by void pointers) that can lead to very difficult runtime errors.

The solution was a collection of four abstract connectors:

- NEXUS_VideoInput
- NEXUS_VideoOutput
- NEXUS_AudioInput
- NEXUS_AudioOutput

Each connector has three essential elements:

- Enumerated type (set by the producer when created)
- Source void* pointer (set by the producer when created)
- Destination void* pointer (set by the consumer when connected)

Both Video and Audio connectors have some additional variables because of implementation details. But they are not essential to the basic functionality of an abstract connector.

The abstract connector is obtained from the producer by means of a GetConnector function. The producer sets the enumerated type and the source pointer. Then, the application passes the connector to the consumer where it sets the destination pointer.

The destination may chose to cache information related to the connection. Holding state information requires allocating memory, therefore we've had to make a series of rules to make sure memory is properly managed.

The Connection Rules

The following things happen during the life cycle of a connection (using NEXUS_VideoInput as an example):

- When the producer (for example, NEXUS_VideoDecoder) is opened or upon the first GetConnector call:
  - The producer creates the NEXUS_VideoInput token.
  - VideoDecoder sets the NEXUS_VideoInput.source pointer to itself. This allows the consumer to call back into the producer if needed.
  - VideoDecoder sets the NEXUS_VideoInput.type enum to NEXUS_VideoInputType_eDecoder. This allows the consumer to know how to use the source pointer. The consumer must recognize this enum. All connector enums are registered per chip in nexus/modules/core/CHIP/include/priv.
- When connected to a consumer (for example, NEXUS_VideoWindow):
  - The consumer sets the NEXUS_VideoInput.destination pointer to its own connection data (in this case, NEXUS_VideoInput_P_Link).
  - The consumer may cache the connection data so that a subsequent connection using the same token does not reset to defaults.
- When disconnected from the consumer:
  - The consumer clears the NEXUS_VideoInput.destination pointer.
  - The consumer is not required to free any cached connection data.
- When the connector Shutdown function is called:
  - It there is still a connection, it is disconnected.
  - The consumer frees any cached connection data.
- When the producer is closed:
  - The thunk layer will automatically call Shutdown. If there is no connection, it will have no effect.
  - It verifies that there are no active connections. This can only happen if there's a malfunction in the thunk layer.

- When the consumer is closed:
  - It verifies that there are no active connections or cached connection data. If there are, it automatically shuts them down.

Synchronization Thunk

The synchronization "thunk" is a required element of the Nexus architecture. It is used to add process synchronization and to allow proxying between execution environments. Although it is possible to disable a module's thunk with various modifications, this is not a supported mode.

Source Files

Every module's thunk exists as two files:

- nexus_MODULENAME_thunks.h - This file redefines the public API into _impl varieties.
- nexus_MODULENAME_thunks.c - This file implements the public API and calls the internal _impl varieties.

Note: MODULENAME is the lowercase name of the module (for example, transport, videodecoder, etc.)

Module Hooks

Every module must include two hooks and follow some rules to engage the thunk layer. The hooks are:

- The main nexus_MODULENAME_module.h header file must include nexus_MODULENAME_thunks.h.
- The module's *.inc file must include nexus_MODULENAME_thunks.c in its sources.

Some rules are:

- nexus_MODULENAME_thunks.h must be included before any other file in nexus_MODULENAME_module.h.
- Every module source file must include nexus_MODULENAME_module.h first.
  This is needed to redefine the public API internally.
- nexus_MODULENAME_module.h must include every public API header file.
  This is needed so that nexus_MODULENAME_thunks.c can have all of the Interface data types.

Platform Generation

The sync thunk is auto-generated from the public API header files using Perl. Perl scripts have been included in nexus/build/tools. It is not required that platforms use these tools, but it is highly recommend.

By convention, the auto-generated thunk *.h and *.c files are placed in nexus/platforms/$(NEXUS_PLATFORM)/bin/syncthunk. These files are recreated by the build system, which means they should not be checked into a source control system. The module's *.inc should use `$(NEXUS_SYNCTHUNK_DIR)` to get to this directory.

Please see *Build System* for information about how the synchronization thunk is built.

Tracing with the Thunk layer

You can trace Nexus public API calls with a small modification of the autogenerated thunk layer. This is not compiled in by default because of code size concerns. To enable this, edit the thunk file as follows:

```
vi nexus/platforms/97405/bin/syncthunk/nexus_video_decoder_thunks.c
```

Then change the #define BDBG_MSG_TRACE(x) line to the following:

```
#define BDBG_MSG_TRACE(x) BDBG_MSG(x)
```

Then activate as follows:

```
export msg_modules=nexus_video_decoder_thunks
nexus decode
```

Build System

Like the Nexus code, the Nexus build system is modular. This modularity adds complexity which can be hard to unravel. The following is intended to help you navigate the build system.

The main pieces are:

- Platform Makefile and Makefile *.inc files
- nexus/build/nexus.inc
- nexus/build/os/$(B_REFSW_OS)/ files
- Module Makefile *.inc files

Platform Makefiles

The platform contains the main Makefile for building Nexus. The platform code can be customized to build Nexus into a variety of configurations including a shared library, a driver (e.g. Linux loadable module), or a archive of object files.

See nexus/platforms/97400/build/Makefile for an example. This Makefile can be traced for linuxuser mode as follows:

- The top-level rule is “all” located in nexus/platforms/$(NEXUS_PLATFORM)/build/Makefile.
- “all” invokes “nexus_install,” which is located in nexus/build/os/$(B_REFSW_OS)/os_rules.inc.
- “nexus_install” invokes “nexus_all.”
- “nexus_all” builds NEXUS_SHARED_LIB.
- NEXUS_SHARED_LIB is defined as a library of NEXUS_OBJECTS and MAGNUM_OBJECTS.
- NEXUS_OBJECTS is defined in nexus.inc by iterating through the list of NEXUS_MODULES.
- NEXUS_OBJECTS is defined in nexus.inc by iterating through the list of MAGNUM_MODULES.
- NEXUS_MODULES is incrementally built by each module’s *.inc file.
- nexus/platforms/$(NEXUS_PLATFORM)/platform_modules.inc includes every module’s *.inc file that builds NEXUS_MODULES.

NEXUS.INC

This Makefile include interprets each module’s *.inc file.

nexus.inc depends on nexus/build/os/$(B_REFSW_OS) files to drive the structure of the Makefile.

The main job of nexus.inc is to dynamically build Makefile variables using module lists. $(foreach) statements are typically used.

nexus/build/os/$(B_REFSW_OS) files

Every OS must provide a set of include files and perl files for building. Please refer to the linuxuser implementation for an example.

Unless you have a custom autogeneration thunk, you should symlink to the linuxuser/module_rules.pl.

Module Makefile *.inc files

Every module is required to provide a Makefile *.inc file. This include file uses a simple convention to define the following:

- Module name
  - NEXUS_MODULES += <module>
- Public Include paths required
  - NEXUS_<module>_PUBLIC_INCLUDES
- Source files to build
  - NEXUS_<module>_SOURCES
  - This includes the synchronization thunk source
- Build flags and other dependencies
  - NEXUS_<module>_PRIVATE_INCLUDES
  - NEXUS_<module>_MAGNUM_MODULES
- Down modules
  - NEXUS_<module>_DEPENDECIES

The nexus/build/nexus.inc file knows how to process the module *.inc files.

See the reference Platform Makefile for an example of how to use nexus.inc and module *.inc files.

Autogenerated Code

The Nexus build system uses Perl to autogenerate the following code:

- Synchronization thunk (for all OS's)
- Linux kernel mode proxy (Linux kernel mode only)
- GCC precompiled headers

The build rules for this autogenerated code are found in nexus/build/os/$(B_REFSW_OS)/module_rules.inc, which is included by nexus.inc. This *.inc file is built by module_rules.pl. The autogenerated *.c and *.h code is placed into nexus/platforms/$(NEXUS_PLATFORM)/bin/syncthunk.

nexus.inc also includes nexus/build/os/$(B_REFSW_OS)/module_rules.pl to build module_vars.inc. This is used to collect of the variables defined in every module's *.inc file.

Performance

Broadcom includes several performance measuring techniques into the standard Nexus build. These include the following:

Callbacks

Broadcom provides stats on how many callbacks are fired and how long the callback took to execute.

Broadcom includes a threshold so that a WRN will be printed on a long-running callback.

Module Lock Time

Broadcom provides stats on how long a module's lock is held. This includes public APIs, private acquire/release, and events and timers.

BProfile

Nexus has a built-in profiler. Compile with:

```
export BPROFILE_SUPPORT=y
make clean
make
```

Edit your program by adding these lines:

```
NEXUS_Profile_Start();

/* application code you want to profile */

NEXUS_Profile_Stop("mycode");
```

Run the application. You will see something similar to the following:

```
total 1372 msec
   % cumulative     self     D-Cache miss
   time     msec msectotal/functioncallsname
 29.5    411.4 405.4 102791/34.8  2947 NEXUS_FlushCache
    6.3   89.1  87.4 54186/ 0.8 61172 NEXUS_P_ThreadInfo_Get
    5.8  120.8  80.0  57488/ 2.3 24420 NEXUS_Module_Lock_Tagged
    4.6  108.8   64.4 68480/ 2.8 24419 NEXUS_Module_Unlock_Tagged
    3.9   57.8   54.5   6305/ 1.3  4644 BMEM_P_CheckGuard+0x90
    3.7  349.8  51.1 101932/34.6  2945 BGRClib_Blit+0x90
    2.5  573.5   35.5  49050/ 16.6  2945 NEXUS_Graphics2D_Blit_impl
    1.8   25.1  24.8  12781/  3.9  3221 BGRC_P_List_WritePacket+0x90
    1.5   38.7  21.8  25550/ 7.9  3222 BGRC_Source_SetSurface+0x90
```

```
    1.5   67.4   21.0  14852/  3.1  4725 BMRC_P_Monitor_Update+0x90
```

Nexus built-in BProfile options

Nexus has some built-in bprofile options. These are simply BProfile_Start/Stop pairs which can be activated with environment variables. They are:

- `export profile_init=y`
  Profile NEXUS_Platform_Init
- `export profile_playpump=y`
  Profile NEXUS_Playpump from a Start to a Stop

Configuring BProfile Output

The formatting of profile data is controlled by the following optional run-time environment variables:

- `profile_show=#`
  Limits the number of entries captured in the report.
- `profile_perthread=y`
  Prints a profile report for each active thread.
- `profile_preempt=y`
  Prints a column with preemption times, such as the time when the CPU was switched out from a function.
- `profile_calls=y`
  Prints a column with the total number of function calls from a given function.
- `profile_accurate=y`
  Accounts for the overhead of instrumentation and adjusts reported data accordingly.

Other BProfile Notes

You must run on a non-SMP Linux configuration in order to get consistent results. For 2.6.31, boot with “nosmp” boot parameter.

If your application creates its own threads, you must call NEXUS_Profile_MarkThread(“name”) from each thread in order for BProfile to interpret that thread’s stack.

INT Stats

Nexus has a timer that prints out L1/L2 interrupt statistics every five seconds. To see this output, `export msg_modules=int`.

Example output:

```
--- 00:05:05.385 int: ------[7405 dump]--------
--- 00:05:05.385 int:  0x1e:AVD_0
--- 00:05:05.385 int:    0x900000:16 BXVD_P_AVD_PicDataRdy_isr[0x2ad70900]:(0x585584,0) 301
--- 00:05:05.385 int:  0xc:BVNF_0
--- 00:05:05.385 int:    0x103000:0 BVDC_P_CompositorDisplay_isr[0x2ad11b2c]:(0x58ef98,0) 150
--- 00:05:05.386 int:    0x103000:1 BVDC_P_CompositorDisplay_isr[0x2ad11b2c]:(0x58ef98,0x1) 151
--- 00:05:05.386 int:    0x103000:2 BVDC_P_CompositorDisplay_isr[0x2ad11b2c]:(0x599310,0) 150
--- 00:05:05.386 int:    0x103000:3 BVDC_P_CompositorDisplay_isr[0x2ad11b2c]:(0x599310,0x1) 150
--- 00:05:05.387 int:  0xa:VEC
--- 00:05:05.387 int:    0x182100:9 NEXUS_Display_P_DequeVbi_isr[0x2ab66c34]:(0x5bcbe8,0x1) 150
--- 00:05:05.387 int:    0x182100:10 NEXUS_Display_P_DequeVbi_isr[0x2ab66c34]:(0x5bcbe8,0) 150
--- 00:05:05.387 int:    0x182100:7 NEXUS_Display_P_DequeVbi_isr[0x2ab66c34]:(0x5bf100,0x1) 150
--- 00:05:05.387 int:    0x182100:8 NEXUS_Display_P_DequeVbi_isr[0x2ab66c34]:(0x5bf100,0) 150
--- 00:05:05.388 int:  0x3a:XPT_DPCR0
--- 00:05:05.388 int:    0x20a008:3 NEXUS_Timebase_P_Monitor_isr[0x2ab770dc]:(0,0) 242
```

See magnum/basemodules/int/bint.c for the meaning of these fields.

Nexus Unit Tests

Every module has a set of unit tests. They are stored for Broadcom-internal use in rockford/unittests/nexus/MODULENAME.

Unit tests must meet the following requirements:

- They must run without user interaction; they cannot block waiting for user input.
- They must run without command-line parameters. (This can be an option, but is not required.)
- They must have a maximum five-minute run time. This allows for unit test automation.

You should document in the module unit test directory whether any test requires a specific stream or data file.

Unit tests should verify all defined behavior. Some examples include the following:

- Opening and closing handles with various options
- All parameters and structure options
- Allocating too much memory
- All enum values, including invalid or out-of-range values
- Calling functions in various contexts (for example, calling GetStatus before Start or after Stop)
- Verifying callbacks
- Verifying the format of data streams being passed out

Unit tests should not and cannot verify undefined or ASSERT-able behavior. Some examples include the following:

- Using a closed or otherwise invalid handle
- Calling Nexus with NULL pointers

Nexus unittests are automatically run every night by our Electric Commander automation server.

Porting a Module to a New Chip

Porting a module means getting an existing module working on a new chip.

- Create a directory symlink for your chip under nexus/modules/MODULENAME.
- Add `#if BCHP_CHIP == XXXX` in the implementation for any chip specific code.

Replacing a Module

Replacing a module means creating a new module that supports an old module's public and private APIs. From the caller's perspective, it provides the same Interfaces.

- Create a new directory for your chip under nexus/modules/MODULENAME.
- Create a directory symlink for "include" to the old chip's include directory. This will ensure that you are using the same API.
- Create a new directory for `src`. Copy files from the old chip for a head start, but then write your own implementation.

Adding a New Module

The following is a checklist for creating a new module:

- Draft your API (public and private)
  - Pattern your module API off a similar existing module to avoid mistakes.
  - Consult the section on *Interface Patterns*
- Run the Coding Convention Check tool

  See **Error! Reference source not found.**
- Have your API reviewed by the Nexus core development team
- Write your module

  Be sure to include symlinks for all chips that will support it.
- Create a module.inc file
  - To begin, copy from a similar existing Nexus module. Then make whatever changes are necessary.
- Create an example and some unit tests
  - Examples go in nexus/examples. Keep them simple and self-explanatory. They will be distributed to all customers.
  - Unit tests go in rockford/unittests/nexus. See the README.txt for rules. These are for internal use only.
- Edit Platform (based on nexus/platforms/97400)
  - Add an "include" of your module.inc file to platform_modules.inc. It must be added in sorted order, from least to greatest dependencies.

    If it must run in the kernel (that is, if it must call Magnum), then be sure to add your module in the top part of the file, before NEXUS_EXCLUDED_MODULES is defined.
  - Add your module name to NEXUS_PLATFORM_DEPENDENCIES in platform.inc.
  - Edit src/nexus_platform.c and call your module's Init and Uninit functions

    Remember to wrap with `#if NEXUS_HAS_<<MODULE>>` to protect others you don't compile in your module.
- Build and run your module and unit test.
- Check into source control (Broadcom internal).
  - Be sure to check in all files.
  - Update Clearcase configuration specifications in /public/nexus/configspecs for all platforms that need the new module. Read the README.txt there.
  - The only sure way to test is to rebuild a clean snapshot view.
  - Make sure you have not broken other platforms. The nightly "cruise control" builds will tell everyone, but you should be pre-emptive.

- Integrate your new module with your application.

Extending a Module

- Read the "Module Extensions section" of nexus/docs/Nexus_Architecture.pdf
- Study nexus/examples/extension.c and nexus/extensions/sample/display.
- Create a new extension directory.
  - It can be inside nexus/extensions for generally used extensions.
  - It should be outside of the nexus tree for customer-specific extensions.
  - The directory should be modeled on the sample extension including chip subdirs, *.inc file, and src and include directories.
- Verify that the module you want to extend has support
  - Verify that <module>.inc has the following hook. Not all do at this time.

```
ifneq ($(NEXUS_<MODULE>_EXTENSION_INC),)
include $(NEXUS_<MODULE>_EXTENSION_INC)
endif
```

- Your *.inc should define the #if needed to hook from the module to the extension.
- Edit nexus_<module>_module.h and add a #include for your public API.
  - This could be a generic name like "nexus_transport_extension.h" or may be specific.
  - The #include should be wrapped with the #if for your extension.
- Add any other hooks necessary.
- Build and run your app with the extension.
  - `export NEXUS_<MODULE>_EXTENSION_INC=xxx`
  - `make clean; make`
- Verify that you can run other code in the module without your extension.
  - unset NEXUS_<MODULE>_EXTENSION_INC
  - make clean;make
- Review the extension with the Nexus architecture team.

Porting Nexus to a New Operating System

All operating system-specific code has been isolated to specific directories and files within Magnum and Nexus. Each OS has a unique name to isolate its code. Current names in use are:

- linuxuser (i.e. Linux user mode)
- linuxkernel (i.e. Linux kernel mode)
- vxworks
- ucos (i.e. uC-OS)
- nucleus
- wince
- no-os (a no-operating system mode used in Broadcom's diags application)

Using linuxuser (Linux user mode) as a template, check the following locations:

- magnum/basemodules/kni/linuxuser
- magnum/basemodules/dbg/linuxuser
- nexus/base/include/linuxuser
- nexus/base/src/linuxuser
- nexus/build/os/linuxuser
- nexus/modules/file/include/linuxuser
- nexus/modules/file/src/linuxuser
- nexus/platforms/$(NEXUS_PLATFORM)/src/linuxuser

To add a new OS, you must choose a standard directory name, and then pattern your code after one of the existing implementations.

When you compile, define the B_REFSW_OS environment variable to your standard directory. The default value of B_REFSW_OS is linuxuser.

Reference Platforms

Nexus software releases usually contain Nexus Platforms for Broadcom Reference Board hardware. These Platforms provide an easy way to build and run standard applications such as Brutus. They also serve as a recommended template for custom Platforms.

Every Platform is custom, so there is no requirement that any part of the Reference Platforms be used in a custom Platform.

The best way for customers to implement their systems is to begin with a reference platform, then add in their application and board-specific modifications. See the next section for creating your own platform.

Source Files

The BCM97400 Platform source code is stored in the directory nexus/platforms/97400. It serves as the basis for many other Reference Platforms (for example, the BCM97401).

Table 3: Platform Directory Structure

Directory	*Purpose*
nexus/platforms/97400/include	Public API for platform
nexus/platforms/97400/src	The source code for the Platform implementation
nexus/platforms/97400/build	Makefiles
nexus/platforms/97400/bin	Location to store binaries that are generated by build; this directory is dynamically created. Do not check into source control.

The nexus/platforms/97400 directory assumes the subdirectories noted in the following subsections.

Include Subdirectory

Every platform has a set of header files in its include directory. They can be customized, but it is recommended that you leave the standard platform header files unmodified.

Table 4: Platform Include Files

File	*Purpose*
nexus_platform.h	Defines the main API for the platform. Includes: • NEXUS_Platform_GetDefaultSettings • NEXUS_Platform_Init • NEXUS_Platform_Uninit • NEXUS_Platform_GetConfiguration
nexus_platform_server.h	Multi-process API's callable from the server. Includes: • NEXUS_Platform_StartServer • NEXUS_Platform_StopServer • NEXUS_Platform_RegisterClient • NEXUS_Platform_UnregisterClient
nexus_platform_client.h	Platform API's callable from the client. Includes: • NEXUS_Platform_AuthenticatedJoin • NEXUS_Platform_GetClientConfiguration • NEXUS_Platform_Uninit
nexus_platform_features.h	Defines number of features for this platform. Can be customized to limit features to reduce memory usage.
nexus_platform_standby.h	Power management API
nexus_platform_version.h	Defines version number for the platform software

Source subdirectory

Every platform has an implementation in its src directory. They can be customized, but it is recommended that you only modify certain files as noted below.

Table 5: Platform Source Files

File	*Purpose*
nexus_platform.c	Implements NEXUS_Platform_Init which initializes all Nexus and all Nexus modules.
nexus_platform_settings.c	Implements NEXUS_Platform_GetDefaultSettings
nexus_platform_fpga.c	Configure the FPGA on the reference board. ***Should be customized.***
nexus_platform_core.c	Map memory and init Core module
nexus_platform_$(PLATFORM).c	Board-specific logic. ***Should be customized.***
nexus_platform_config.c	Configure modules with board-specific settings. Create handles for board-specific resources like DAC's, ADC's, etc. ***Should be customized.***
nexus_platform_interrupt.c	Dispatch L1 interrupts to BINT_Isr
nexus_platform_pinmux.c	Configure board-specific pinmuxing ***Should be customized.***
nexus_platform_vcxo.c	Configure board-specific VCXO ***Should be customized.***
linuxuser/nexus_platform_os.c	OS-specific code (this for linux user mode)
linuxkernel/nexus_platform_os.c	OS-specific code (this for linux kernel mode)
linuxuser.proxy/*	User-mode proxy portion of kernel mode
linuxuser.client/*	Client portion of user-mode multiprocess

Build Subdirectory

Every Platform has a main Makefile. The Reference platforms Makefile is a sample of how to build Nexus.

Table 6: Platform Makefiles

File	*Purpose*
Makefile	The main Makefile for the Platform. It will build the platform code and all modules which the platform includes.
platform.inc	Similar to a module *.inc file, but for the platform's code.
platform_modules.inc	List of modules to be included for this platform. ***Should be customized.***
platform_app.inc	This defines the variables needed by an app to build with the platform. This is the dynamic version. A static version is also available with "make nexus_headers".

The reference platform build system uses files in nexus/build for some of its processing. Some of those files are given in **Error! Reference source not found.**.

Table 7: Build Files

File	*Purpose*
nexus.inc	Processing standard nexus module *.inc files. Only used internally when building nexus.
nexus_defs.inc	Makefile include which defines standard variables for nexus.inc and platform_app.inc
module_vars.pl	Perl script which processes module *.inc files
module_vars.inc	Makefile auto-generated by module_vars.pl. Do not check this into source control.
Makefile	A high-level Makefile which dispatches to the Platform Makefile based on the NEXUS_PLATFORM environment variable.

Autogenerated Synchronization Thunk and Kernel Mode Proxy

The Platform Makefile builds the synchronization thunk and kernel mode proxy for the public API of every module. The results are a set of *.h and *.c files which are placed into the nexus/platforms/$(NEXUS_PLATFORM)/bin/syncthunk subdirectory.

See

Build System for more details.

Linux Drivers

The BCM97400 Platform supports Linux drivers in two flavors:

- User mode driver – a small driver which manages MIPS L1 interrupts. Nexus uses a few ioctls to receive and process these interrupts. Nexus runs in user mode. See BSEAV/linux/driver/usermode for the code.
- Kernel mode driver – a large driver which runs a complete Nexus implementation.

Creating a New Platform

Overview

Nexus Platform is the location to add your board configuration code and custom settings. The code is meant to be modified. However, it's likely that you won't want to start from scratch. The following describes a process to make required changes while ensuring that you are leveraging the Broadcom reference platform code.

You must keep track of any Platform changes or customizations so that when you receive a new release, you can review and reapply your changes in the new code base. The internal Platform APIs are simple, but they are not guaranteed to be unchanging.

Process

The following diagram shows the process of getting a Nexus release running on your board.

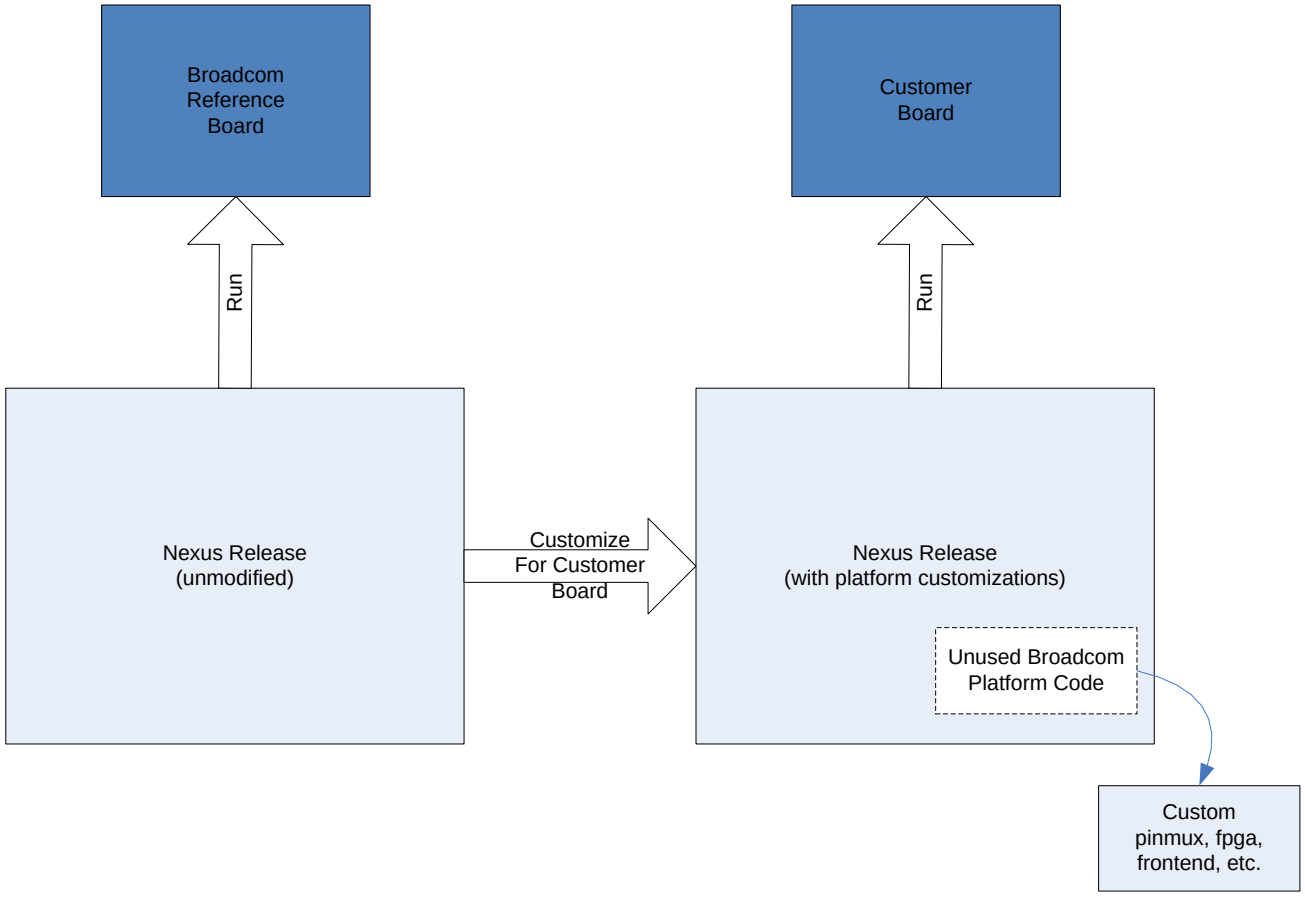


Figure 5: Process for Getting a Nexus Release Running

- When you receive a new release of Nexus, always run the unmodified example applications (nexus/examples) from an unmodified Nexus tree on a reference platform. This gives us and you a baseline of functionality.
- In most nexus releases, there will only be one platform directory. This should serve as the basis for your customized platform.
  - In some cases, there may be multiple platform directories. If so, choose the platform which most closely resembles your platform to serve as a baseline for your platform.
  - You can modify code in that directory or create your own directory under nexus/platforms.
- Determine the differences between your HW platform and reference HW platform. Look for such differences as PinMux configuration, Frontend configuration, etc. Based on differences, determine which files you will need to modify. You will likely need to modify these files:
  - nexus_platform_pinmux.c – set according to your board layout and features
  - nexus_platform_fpga.c – necessary for your board if an FPGA is used
  - nexus_platform_frontend.c – necessary to bring up your tuners and demods
  - nexus_platform_config.c – necessary to set ADC and DAC settings on inputs and outputs
  - nexus_platform_settings.c – only if you want NEXUS_Platform_GetDefaultSettings to return your preferred defaults
- Broadcom does not recommend that you modify the following unless absolutely necessary:
  - nexus_platform.c – initializes the modules. This file has #if's around each module init. If you remove a module from platform_modules.inc, it will not be called in this file.
  - nexus_platform_core.c – this brings up the lower-level elements of Nexus
  - nexus_platform_interrupt.c
  - linuxkernel, linuxuser or linuxuser.proxy subdirectories – you can create your own OS driver design, but if you are using Linux, we recommend that you use the drivers we've designed.
- There are two ways of customizing platform code:
  - Making changes to files directly in the platform directory. This could mean making your own copy of a reference platform directory or just modifying files in place. To keep track of your changes, you can simply diff your directory with the original.
  - Setting the PLATFORM_INC environment variable to point to a custom platform.inc file. You can customize this platform.inc to include a mix of files inside and outside the Nexus tree. This allows your custom code to be kept in another place in your source control system.
- If you use PLATFORM_INC, then copy build/platform.inc to your location, define PLATFORM_INC to point to that location, then modify that file to build whatever files from whatever location you want. You must choose unique file names so that make's vpath feature doesn't pull the wrong file.
  - include $(PLATFORM_INC) is found in build/platform_modules.inc

Here is an example of how a customized platform.inc can compile a blend of Broadcom platform code and custom platform code:

```
NEXUS_PLATFORM_SOURCES += \
    /home/user/nexus_mods/nexus_platform_pinmux.c \
    $(NEXUS_TOP)/platforms/$(NEXUS_PLATFORM)/src/nexus_platform_core.c \
    $(NEXUS_TOP)/platforms/$(NEXUS_PLATFORM)/src/nexus_platform_interrupt.c \
    $(NEXUS_TOP)/platforms/$(NEXUS_PLATFORM)/src/$(B_REFSW_OS)/nexus_platform_os.c
\
    $(NEXUS_TOP)/platforms/$(NEXUS_PLATFORM)/src/nexus_platform.c \
    $(NEXUS_TOP)/platforms/$(NEXUS_PLATFORM)/src/nexus_platform_vcxo.c \
    /home/user/nexus_mods/nexus_platform_custom_config.c \
    /home/user/nexus_mods/nexus_platform_custom_frontend.c \
    /home/user/nexus_mods/nexus_platform_custom_settings.c
```

Bring-Up Hints

- If you want to perform any board bring-up inside NEXUS_Platform_Init, you can do that inside nexus_platform_config.c, which is called at the end of NEXUS_Platform_Init.
- Limit the number of modules in build/platform_modules.inc as you bring up new modules. Just comment out the modules you don't want to build.
  - This only works well for high-level modules that no one else depends on. For instance, playback or record.
  - If you disable audio, you will also have to disable playback.
  - If you disable dma or i2c, you will have to disable a whole list of modules.
- Test your new platform by running the Nexus example applications in nexus/examples. Start with simple ones (boot, graphics, message) and build up to more complex ones (decode, playback, tune_xxx, etc.)

Debugging

Below are some tips and tricks to help with debugging a nexus application. These are runtime flags that can help debug the system, including disabling certain features and/or enabling log messages. In Linux, these flags are set as environment variables (e.g., export `force_vsync=y` enables the force_vsync flag). Other operating systems have different methods of passing these variables into nexus, but the same variable names apply.

Also see BSEAV/app/brutus/docs/RefswDebugGuide.doc for other tips.

Log Messages

Nexus uses the standard magnum BDBG interface for debugging. To enable various runtime debugging, you can set several runtime variables. Each source file will have a BDBG_MODULE tag in the file that describes what the module name is. To enable messages, do the following:

```
export msg_modules=moduleName1,moduleName2,...,moduleNameN
```

and/or

```
export trace_modules=moduleName1,moduleName2,…,moduleNameN
```

msg_modules will enable all BDBG_MSG prints in the module. trace_modules will enable function entry and exit prints (BDBG_ENTER/BDBG_LEAVE). By default, warning (BDBG_WRN) and error (BDBG_ERR) messages will always print.

Useful msg_modules to capture

The following are some common msg_modules settings.

Msg_module setting	***Code***	***Purpose***
`int`	`magnum/basemodules/int` `nexus/modules/core`	Prints a digest of L1 and L2 interrupts coming into magnum every 5 seconds.
`BXVD_DMTSM`	`magnum/portinginterface/xvd`	Prints video TSM equation
`nexus_video_decoder_priv`	`nexus/modules/video_decoder`	Prints picture data ready interrupt information flowing from XVD to VDC

Overriding Features

Often, it's useful to disable a feature in Nexus without needing to disable it in the application. Nexus allows for this in several places for debugging the system. The following options are provided to disable features for debugging:

Table 8: Nexus Debugging Options

Environment Variable	*Purpose*
`astm_disabled=y`	disable ASTM functionality
`cont_count_ignore=y`	disable continuity count checking in transport
`sync_disabled=y`	disable high-precision lip sync (i.e. SyncChannel) functionality
`force_vsync=y`	disable basic TSM in VideoDecoder and AudioDecoder
`audio_equalizer_disabled=y`	disable audio equalizer functionality
`audio_processing_disabled=y`	disable audio post-processing functionality
`audio_mixing_disabled=y`	disable mixing audio playback with other channel types
`no_independent_delay=y`	disable audio independent delay
`multichannel_disabled=y`	disable multichannel AudioDecoder output
`not_realtime_isr=y`	disable real-time prioritization for the ISR thread in Linux user mode. This can be useful to debug system hangs.
`no_watchdog=y`	disable VideoDecoder and AudioDecoder watchdog handling. This is needed if you want to interact with AVD using the outer ARC uart.
`no_dynamic_rts=y`	DTV only: disable Nexus Dynamic RTS (i.e. VDB restrictions)
`debug_mem=y`	Validate Magnum MEM heaps every five seconds. This will find guard violations and other corruptions.
`avd_monitor=0`	Print the AVD0 debug log to the console. Use 1 for AVD1.
`main_memory_size=X`	Limit the amount of memory used for heap[0]. X is in MB.
`sw_destripe=y`	Bypass hardware destripe of StillDecoder stills
`pq_disabled=y`	Disable all picture quality features

Coding Conventions

Overview

The Nexus coding convention is a stricter version of the Magnum coding convention.

The Magnum architecture's coding convention has been in wide use. Broadcom chose to build on the successful style of Magnum and extend it with three goals in mind:

- Maintain enough of a difference from the Magnum API to tell easily whether a given API is inside Nexus or Magnum.
- Stricter enforcement of some style elements, such as variable naming, indenting style, etc.
- Ensure the Nexus API can be automatically proxied.

These rules are needed to give Nexus a consistent, clean appearance. This consistency makes Nexus easier to use as a system-wide API. It allows different parts of the system to interconnect more easily, both in implementation and in the user's understanding.

Some of the rules are needed to allow Nexus to function across kernel or process boundaries. If certain rules are not followed, the API might work in Linux user mode, but will not work in Linux kernel mode.

Source Code

All code will be written in C and will conform to the C89 (ANSI C) specification.

Basic data types are set by the Magnum architecture. Types like bool and uint32_t are defined in Magnum's bstd_defs.h. See the Magnum architecture documentation for details.

The float data type is not allowed in the API or implementation.

The build system will require the Make utility. Platform sample code will use Perl but it is not required that every Platform use Perl.

The remainder of this coding convention applies to the Interface definition, not Module implementation.

CamelCase

Nexus uses CamelCase naming for all functions, structures and enums. CamelCase means writing a compound word using a single capital letter to delineate each word.[1] The underscore separator is limited to specific cases noted below, otherwise CamelCase should be used to separate words.

Table 9: CamelCase Naming Examples

Bad Example	*Good Example*
NEXUS_VideoDecoder_start	NEXUS_VideoDecoder_Start
NEXUS_Inputband_SetSettings	NEXUS_InputBand_SetSettings
NEXUS_AnalogVideoDecoder_getConnector	NEXUS_AnalogVideoDecoder_GetConnector
NEXUS_Display_enable_mad	NEXUS_Display_EnableMad

Acronyms are promoted to words and follow CamelCase.

Table 10: CamelCase Acronym Naming Examples

Bad Example	*Good Example*
NEXUS_VideoFormat_eNTSC	NEXUS_VideoFormat_eNtsc
NEXUS_VideoWindow_SetCSC	NEXUS_VideoWindow_SetCsc

[1] ThisIsASentenceWrittenInCamelCase

Limited Hungarian Notation

Hungarian notation is used in only two cases:

- "p" for pointers to defined types

```
void NEXUS_VideoDecoder_GetSettings(
    NEXUS_VideoDecoderHandle handle,
    NEXUS_VideoDecoderSettings *pSettings /* [out] */
    );
```

- "e" as a prefix in enums to separate the specific enum from its type.

```
typedef enum NEXUS_VideoFormat
{
    NEXUS_VideoFormat_eNtsc,
    NEXUS_VideoFormat_ePal
} NEXUS_VideoFormat;
```

- In all other cases, Hungarian notation is not allowed. Here are examples which are not allowed:

```
typedef struct NEXUS_Example
{
    bool bEnable;
    int iData;
    NEXUS_EnumType eMember;
    NEXUS_StructType sThing;
} NEXUS_Example;
```

Indentation

Indentation is done with four spaces and not hard tabs. This is compatible with all modern editors and avoids the confusion of mixing tabs and spaces.

Example:

```
typedef struct Settings
{
    int member; /* This is a multi-line comment which lines
                   up with the previous line in all editors. */
} Settings;
```

Curly Braces

Curly braces for structures and enums will start on the line following the definition.

Good example:

```
typedef struct Settings
{
    int x;
} Settings;
```

Bad example:

```
typedef struct Settings {
    int x;
} Settings;
```

NEXUS_ namespace Prefix

In order to distinguish Nexus code from Magnum code as well as user code, all public APIs, structures, enums, enum tags, and macros must start with "`NEXUS_`". This provides a clear distinction between the two code bases and a unique name space that does not conflict with other code in a user system.

Interface Handles

NEXUS Interface handles should be opaque types, just like Magnum handles. They must state the Interface name and end with "Handle."

Example:

```
typedef struct NEXUS_VideoDecoder *NEXUS_VideoDecoderHandle;
```

Interface Function Naming

Functions will be named with the following syntax:

```
ReturnType NEXUS_Interface_FunctionName();
```

"Interface" (corresponding to the Nexus Interface) should be provided on all functions. "FunctionName" should generally be in VerbNoun form.

Table 11: Interface Function Naming

Bad Example	*Good Example*
NEXUS_Video_Decoder_Start	NEXUS_VideoDecoder_Start
NEXUS_InputBand_Set_Settings	NEXUS_InputBand_SetSettings
NEXUS_AnalogVideo_getConnector	NEXUS_AnalogVideo_GetConnector
NEXUS_VideoWindow_Matrix_Configure	NEXUS_VideoWindow_ConfigureMatrix

The name of the Nexus Module that provides the Nexus Interface should not appear in the function name, unless it just happens to match the Nexus Interface. The reason for this is that the synchronization context may need to be changed as requirements change and/or Broadcom discovers internal modularity issues that require changing the Nexus Module layout.

A private API must either end in "_priv" if it's a task function or "_isr" if it is a Magnum ISR function.

Function Parameters

Parameters can be one of the following:

- Opaque handle
- Simple type passed by value (e.g. int, bool, enum, void *)
- Pointer to simple type or structure

Function parameters should occur in the following order:

- Interface handle or index for opening a handle
- All “in” parameters
- Any “out” parameters

All parameters must be either [in] or [out]. Any [in] parameters which are passed by reference (pointers to variables) should be marked as the “const.” [in/out] parameters are not allowed.

All callbacks must use the NEXUS_CallbackDesc structure. See nexus_base_types.h for its definition. This structure contains the function pointer used for a callback. No other use of function pointers is allowed. A direct function pointer violates synchronization rules and cannot work across a proxy like Linux kernel mode.

Public API functions cannot receive or pass Magnum ISR function pointers as parameters or in structures. This is allowed in a module’s private API.

Function Return Types

These may be one of three varieties:

1. **void: Function can never fail**

This is either because it is so simple it's impossible to fail (such as GetSettings), or there is no possible recovery for a failure.

Most GetSettings or GetDefaultSettings functions are inherently impossible to fail because they only return settings that the user last set. In contrast, GetStatus can fail because it has to query the status of hardware which may be unavailable.

Functions like Stop or Close have no possible recovery from a failure. Any failure is an internal system failure and should be reported as a bug. If we pass out an error code, there is no meaningful work the application can do. Application programmers would just be improvising and likely crashing later. Instead, we should use void, catch any errors internally and get them fixed.

2. Opaque handles: Used when an Interface instance is created or retrieved.

Returned by Open or Get functions

If the call fails, NEXUS should issue a BDBG_ERR, leave the system in a clean state, and then return NULL.

3. NEXUS_Error: All other cases

NEXUS_Error codes are returned in all other cases. A value of zero (NEXUS_SUCCESS) is success; all non-zero values are errors. Functions can designate meaning to specific error codes, but those should be documented in the header file comments. Applications should check error codes and, at a minimum, print an error. If a function returns an error, its out parameters are in an undefined state unless an exception is noted in the header file comments. If a Set, Start or Connection function returns an error, the caller should assume that the new state is not applied.

Function Prototypes

Parameter comments should follow these rules:

- [out] comments are required because they are used by the auto-generated proxy layer.
- [in] comments are forbidden because they can be assumed if not [out].
- Parameter-line comments should add to the user's understanding of the parameter's usage. Comments never simply restate the parameter type and/or name. If they do not add information, it's best to have no comment

Function prototypes should be structured as follows:

- Each function parameter is on a separate line, four spaces (that is, one tab) indented.
- Return type is on the same line as the function name.
- Opening "(" is on the same line as the function name.
- Closing ")" is on its own line. This allows for consistent parameter-line comments.
- Functions that have no parameters should have a "void" parameter in the prototype all in one line:

Example prototypes:

```
NEXUS_Error NEXUS_Timebase_SetSettings(
    NEXUS_Timebase timebase,
    const NEXUS_TimebaseSettings *pSettings
    );

NEXUS_VideoDecoderHandle NEXUS_VideoDecoder_Open(
    unsigned index,
```

```
        const NEXUS_VideoDecoderSettings *pSettings
        );

    void NEXUS_VideoDecoder_Close(
        NEXUS_VideoDecoderHandle handle
        );

    void NEXUS_VideoDecoder_GetSettings(
        NEXUS_VideoDecoderHandle handle,
        NEXUS_VideoDecoderSettings *pSettings /* [out] */
        );

    void NEXUS_VideoDecoder_CheckGlobalState(void);
```

Structures

The structure naming convention is as follows:

```
NEXUS_[Interface]StructureName
```

The structure name should be repeated both in the struct declaration and as the typedef. Do not use two different names for a single structure.

Examples:

```
typedef struct NEXUS_ParserBandSettings
{
    NEXUS_ParserType parserType;
} NEXUS_ParserBandSettings;

void NEXUS_ParserBand_GetSettings(
    NEXUS_ParserBand parserBand,
    NEXUS_ParserBandSettings *pSettings /* [out] */
    );
```

In most cases, structures should end with “Settings.” For a structure that is used with NEXUS_Interface_GetSettings and NEXUS_Interface_SetSettings, it should be named NEXUS_InterfaceSettings.

For any other function (e.g. NEXUS_Interface_Function) it should be NEXUS_InterfaceFunctionSettings. An example is:

```
typedef struct NEXUS_VideoDecoderStartSettings {
    int x;
    int y;
} NEXUS_VideoDecoderStartSettings;

void NEXUS_VideoDecoder_Start(
    NEXUS_VideoDecoderHandle handle,
    const NEXUS_VideoDecoderStartSettings *pSettings
    );
```

Structures cannot contain pointers to variables. The reason is that Nexus does not perform any deep copy when marshalling data across a proxy.

Not allowed:

```
typedef struct NEXUS_VideoDecoderStartSettings
{
    NEXUS_MoreSettings *pMoreSettings;
} NEXUS_VideoDecoderStartSettings;
```

This should be:

```
typedef struct NEXUS_VideoDecoderStartSettings
{
    NEXUS_MoreSettings moreSettings;
} NEXUS_VideoDecoderStartSettings;
```

Enums

Enums are named like structures. Each specific enum value should repeat the enum type name followed by "_e" followed by the specific enum value.

Example:

```
typedef enum NEXUS_VideoDecodeChannelType
{
    NEXUS_VideoDecodeChannelType_eVideo,
    NEXUS_VideoDecodeChannelType_eStill,
    NEXUS_VideoDecodeChannelType_eMax
} NEXUS_VideoDecodeChannelType;
```

If you need to index an array based upon an enumeration, there should be a last member named xxx_eMax. Enumerations should have an eMax value that indicates the maximum value used in the enumeration if the values of the enum are continuous. This can be used for sanity checking and can also be used for creating arrays when that functionality is desired. If that is done, the array size must be made explicit, that is:

```
bool supportedChannelTypes[NEXUS_VideoDecodeChannelType_eMax] =
{true, true};
```

This will force a compiler warning if values are added/removed from the enumeration without updating such an array.

Module Function Names

Every module supports an Interface for initializing and uninitializing the module. This is normally called by the Platform, not by the application. These functions are normally placed in nexus_MODULE_init.h in the public API. The word "Module" should appear as part of the Interface name. The rule is:

```
NEXUS_NameModule_FunctionName();
```

Examples:

```
NEXUS_TransportModule_Init();
NEXUS_TransportModule_Uninit();
NEXUS_TransportModule_SetSettings();
```

The same applies to structures and enums used in the module API.

Macro Names

Macros should be prefixed with NEXUS and be all upper case. Because CamelCase is not possible, use underscores to separate words.

Examples:

```
#define NEXUS_NUM_VIDEO_DECODERS
#define NEXUS_TRANSPORT_IS_DSS
```

#includes

A Nexus Interface may #include another Interface header file. However, every #include creates a binding. Bindings should be minimized.

Nexus cannot #include any Magnum header file in its public API. This includes anything in magnum/basemodules and magnum/commonutils.[2]

[2] nexus_base.h may use a source-control symlink to bstd_defs.h so that basic types are compatible with Magnum. This is transparent to the Nexus user.

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.
secondary_destructor	function	Optional secondary destructor for handles that are destroyed as second params. A primary destructor is still required. See NEXUS_Playpump_ClosePidChannel.

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.
fixed_length=X	function param	Input parameter which is a fixed length byte array. Required if input size may be unknown to thunk.
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.