



Reference Software Debugging Guide

Revision History

Revision	Date	Change Description
STB_RefSw-SWUM200-R	06/12/2008	Initial version, from D. Erickson v 1.34
STB_RefSw-SWUM201-R	04/26/2010	Added ARC and GISB and real-time sections.
1.1	5/16/2012	Added AVD/Raaga UART capture Remove Linux 2.4 note
1.2	8/27/2013	Clarify B_REFSW_DEBUG, add BERR_TRACE, remove section on Brutus
1.3	5/23/2014	Remove Brutus. Convert examples to MMA and ARM.
1.4	10/5/2015	Remove section on MRC jail. Minor other changes.
1.5	10/16/2015	Add B_REFSW_DEBUG=minimal
1.6	12/7/2015	Rework Debug Logging section
1.7	2/2/2016	Update description of B_REFSW_DEBUG Build Options
1.8	2/14/2017	Update section on priority inversion
1.9	4/3/2017	Reorder logging section, rework /proc, add sections on mutex tracking, slow call/callback, stuck callback, nexus call trace, tracelog

Table of Contents

Debug Logging	4
B_REFSW_DEBUG Build Options	5
Logger application.....	6
Module-Level MSG Control	7
Enabling MSG level at Run-Time.....	7
Frequently used msg_modules.....	8
Video Decoder (HVD) Logging	9
Audio Decoder (DSP) Logging	9
Important Errors	10
BKNI_Fail.....	10
BDBG_OBJECT_ASSERT	10
BERR_TRACE	11
0xDEADDA7A and 0xDA7ADEAD	11
/proc Filesystem.....	12
Using GDB and GDBServer	13
Capturing a Core Dump	13
Analyzing Core Dumps with the GNU Project Debugger GDB.....	13
Monitoring a Running (or Hung) Program with GDB.....	14
Kernel Mode Oops	14
Step-Through and Graphical Debugging.....	15
Address Range Checkers (ARC[H] and MRC)	16
Overview	16
Interpreting ARC violations.....	17
Interpreting Secure ARC violations.....	17
Interpreting ARC Registers.....	18
Magnum's Auto-Programming of the ARCs	18
GISB Arbiter Timeouts	20
Magnum Mutex Tracking.....	21
Nexus Slow Call and Slow Callback Tracking	21
Nexus Stuck Callback Tracking.....	21
Runaway Interrupts	22
Nexus Call Trace	22
Tracelog.....	23
Debugging Real-Time Threads	24
Background on Real-Time Threads.....	24
Magnum and RT Threads.....	24
Debugging RT Scheduling Problems	25
Solutions for RT Scheduling Problems	26
Appendix A: Setting Run-Time Variables	27

Introduction

Debugging software on embedded systems has never been easy. This document describes the debug capabilities that are included in the Magnum and Nexus Set-Top Reference Software to help. It also provides some recommendations and best-practices for debugging.

This document is written primarily for the Linux environment.

Debug Logging

Magnum and Nexus have a Debug Interface (DBG) which allows software to log ASCII data to the console or a storage device. Debug output is categorized into levels which can be distinguished in the log by a three-character prefix as follows:

Table 1: Levels of Debug Output

Level	Abbreviation	Prefix	Description
Error	ERR	###	Something is wrong and must be fixed.
Error Trace	BERR_TRACE	!!!	Something is wrong and must be fixed. Unlike ERR, this only provides error call stack information.
Log	LOG	(none)	Important information which will not be compiled out in release build.
Warning	WRN	***	Important information, compiled out in release build.
Message	MSG	---	Internal debugging, compiled out in release build. Defaults off in order to not overwhelm the console. Can be enabled on a per-module basis, usually on request from a Broadcom developer.
Trace	TRACE	...	Internal debugging for function entry/exit points, compiled out in release build. Can be enabled on a per-module basis, usually on request from a Broadcom developer.

The most important levels of debug to monitor and address are ERR and BERR_TRACE. If Nexus/Magnum has a misconfiguration, or detects a bad parameter, or rejects a bad handle, or runs out of system or device memory, or detects an Address Range Checker (ARC) violation, or if an internal processor has a watchdog reset – all of these events result in ### or !!! in the console. They will always print an ERR or BERR_TRACE.

Because of their importance, we strive to never have “normal” errors in the console. ERR and BERR_TRACE should be fixed in Nexus, Magnum or the application.

Applications can also detect many errors by calling Nexus status functions, but these require correct code to be written, to have access to handles, and to output their results to some place where it can be seen. Because any of these steps may not happen, debug logging is the most important part of system debugging.

B_REFSW_DEBUG Build Options

There are four main build options for DBG, controlled by the B_REFSW_DEBUG build-time environment variable:

B_REFSW_DEBUG	Name	Description
y	Debug mode (default)	Full debug is available, including MSG, WRN, LOG and ERR. Include GDB debug information. Minimal compiler optimization. Binary will be large.
n	Release mode	MSG and WRN are compiled out. ERR, LOG and /proc work. BDBG_ASSERT only prints an error. No GDB debug information. Compiler optimization. This is the recommended “release mode”.
minimal	Minimal mode	Like “Release mode”, but LOG and /proc are also compiled out. Error messages are compacted (file and line number only).
no_error_messages	Nothing on the console	Not Recommended: Nothing prints, not even error messages. This is only to be used if it is absolutely critical to reduce code size to a bare minimum, and only after <i>*all*</i> debug is complete.

The highest levels provide the most debug information, but also increase code size. It is recommended that you run with the highest debug level possible on your system.

You can also make fine-tuned adjustments to DBG using other variables. See `magnum/basemodules/dbg/bdbg.inc` for detail. Variables include:

Export	Description
<code>B_REFSW_DEBUG_LEVEL={err log wrn}</code>	Adjust the global debug level. <code>B_REFSW_DEBUG=n</code> sets this to "log".
<code>B_REFSW_DEBUG_COMPACT_ERR=y</code>	Convert all <code>BDBG_ERR</code> and <code>BERR_TRACE</code> to a <code>printf</code> of file and line number. This reduces code size. <code>B_REFSW_DEBUG=n</code> and <code>=minimal</code> set this to "y".
<code>B_REFSW_DEBUG_ASSERT_FAIL=n</code>	Change <code>BDBG_ASSERT</code> into a print, but no <code>BKNI_Fail</code> . <code>B_REFSW_DEBUG=n</code> and <code>=minimal</code> set this to "y".

See `nexus/build/nexus_defs.inc` to see how `B_REFSW_DEBUG` is mapped into various BDBG options in `bdbg.inc`.

Logger application

Nexus starts a separate "logger" process to print DBG output from a memory buffer. This allows Nexus/Magnum code to not be backed up by slow UART output. It also allows Nexus to interleave whole lines of kernel and user mode DBG output.

The logger application does introduce some delay before DBG output is seen on the console. You may see lines from kernel context print after lines from user context even though the kernel prints were called before. You may also see `__OVERFLOW__` messages on your console and some lines may be missing. To avoid overflow, you can disable the logger by running with export `debug_log_size=0` or by deleting the `logger` application. However, this will cause UART delay to slow down the code and will cause some lines of debug output to be scrambled.

See `nexus/utls/logger` for source code. It is a simple app which can be modified to route output to other debug facilities in the system.

The standard logger app directs all output to `stderr`, not `stdout`. If you want to capture output from the app, either with or separate from application `stdout`, do the following:

```
# with stdout
nexus decode 2>&1|tee CAPTUREFILE

# separate from stdout
nexus decode 2>CAPTUREFILE
```

Module-Level MSG Control

Debug output can be controlled either on a global level, or on a module-by-module level. ERR and WRN are usually globally enabled, but MSG is enabled by module.

At the top of most *.c and *.cpp files there is a BDBG_MODULE() macro. The text inside the parentheses is the module name.

Instances of the debug interface can run in either kernel mode, user mode, or both. You have independent control of each instance.

In user mode, you control the MSG level using the `msg_modules` environment variable. The `msg_modules` variables should contain a case-sensitive, comma-delimited list of modules. For instance:

```
# user mode
export msg_modules=nexus_video_decoder_priv,BMMA_Alloc
nexus decode

# kernel mode
export config=msg_modules=nexus_video_decoder_priv,BMMA_Alloc
nexus decode
```

Enabling MSG level at Run-Time

After `insmod`'ing the driver, you can dynamically enable more MSG modules by writing to the `/proc` interface. For Linux kernel mode, do this:

```
echo "nexus_video_decoder_priv msg" >/proc/brcm/debug
echo "BMMA_Alloc msg" >/proc/brcm/debug
```

Use "wrn" instead of "msg" to revert to WRN debug level.

This `/proc` feature is not supported in Linux user mode.

Frequently used msg_modules

Module	Purpose
nexus_flow_*	Print open/close/start/open for common Nexus interfaces
nexus_video_decoder_priv	Monitor pictures being delivered to VDC in the XVD data ready interrupt
bape_startprint	Print audio filter graph when starting audio decode
nexus_platform_settings	Print Nexus heap configuration at init time
int	Print summary of Magnum interrupts every 5 seconds
nexus_trace_*	Use when building with NEXUS_EXTRA_CFLAGS=-DNEXUS_P_TRACE=1 to get a Nexus call trace.
nexus_statistics_api	Print every Nexus API call that took at least 2 msec every 5 seconds
BMMA_Alloc	Print every Magnum heap allocation
BVDC_MEMCONFIG	VDC heap memory configuration
BXDM_PPQM,BXDM_PPDBG	The requested log for video decoder TSM debug
BXVD	See XVD channel configuration

Video Decoder (HVD) Logging

The HVD video decoder can output debug information to specific UART's. But your board may not have those UART's wired, or you may not know the correct pinmuxing, so the simplest way to get this logging is to tell Nexus to redirect it to the Linux console.

To redirect HVD (or SVD or AVD) output, set this at runtime before starting Nexus.

```
# for HVD0
export avd_monitor=0

# for HVD0 and HVD1
export avd_monitor=01

# for all HVD's
export avd_monitor=y
```

You will see this output:

```
00:00:01.234 AVD: 0: DramLogControl=1
00:00:01.234 AVD: 0: DramLog Base=0x0fffd000 size=00000a00
00:00:02.233 AVD: 0:
00:00:02.234 AVD: 0: :0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:
00:00:03.233 AVD: 0: 0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:
00:00:04.233 AVD: 0: 0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:
00:00:05.233 AVD: 0: 0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:0:
```

This output can be analyzed by our video decoder team.

Audio Decoder (DSP) Logging

RAAGA DSP output can be captured as follows:

```
export audio_uart_file=audio.uart
export audio_debug_file=audio.debug
export audio_core_file=audio.core
```

This generates binary files which can be analyzed by our audio decoder team.

Important Errors

BKNI_Fail

Nexus and Magnum call BKNI_Fail whenever an unrecoverable error occurs. The system is terminated immediately so that we get information as close as possible to the point of failure. The Linux implementation of BKNI_Fail will dereference zero, which causes a segmentation fault. It will look like this:

```
BKNI_Fail is intentionally causing a segfault. Please inspect any prior
error messages or get a core dump stack trace to determine the cause of
failure.
Segmentation fault      (core dumped) nexus playback
```

When reporting this error, it's important to not report it as a segmentation fault. That was only the means to stop execution. Instead, investigate what happened before that led Nexus/Magnum to call BKNI_Fail.

Also, BKNI_Fail should never be called for recoverable errors.

BDBG_OBJECT_ASSERT

The DBG interface has a set of macros that validate instances. Magnum and Nexus may use this to validate any pointer it receives from the caller or from other modules in the code. The BDBG_OBJECT_ASSERT macro will kill your application in one of the following conditions:

- The pointer is NULL.
- The pointer references an object that has been closed (that is, deallocated).
- The pointer is an invalid, random, or uninitialized value.

The failure you will see on the console will look like this:

```
BDBG_OBJECT_ASSERT on object 0x29217350 (400:400 bytes) was allocated at
nexus/modules/transport/src/nexus_stc_channel.c:164 and freed at
nexus/modules/transport/src/nexus_stc_channel.c:118
!!! Assert 'Bad object of expected type #NEXUS_StcChannel:0x29217350
(0xda7adeadda7adead:0x7f87bcebc0)' Failed at
nexus/modules/transport/src/nexus_stc_channel.c:121
```

When this happens, get a stack trace from the core dump and find the offending call.

See `magnum/basemodules/bdbg/bdbg.h` for documentation on how to use BDBG_OBJECT to protect your code.

BERR_TRACE

Most Nexus and Magnum code use BERR_TRACE to provide a console-based stack trace for failing code paths. A typical instance looks like this:

```
!!!Error rc(0x8) at
nexus/modules/video_decoder/src/nexus_video_decoder.c:2624
!!!Error rc(0x8) at
nexus/modules/video_decoder/src/nexus_video_decoder.c:2258
!!!Error rc(0x8) at
nexus/modules/simple_decoder/src/nexus_simple_video_decoder.c:849
```

To interpret this, you must open the source code and find the given line of code. If you are reporting the bug, be sure to include both the console and the related source file, especially if the source file has been modified from the released code. This will allow the developer to get the same context.

0xDEADDA7A and 0xDA7ADEAD

Nexus and Magnum call BKNI_Malloc and BKNI_Free to allocate and free memory from the OS. This abstraction allows us to work with multiple OSes. It also allows us to help find errors. After BKNI_Malloc calls malloc(), it wipes the memory with 0xDEADDA7A. Before BKNI_Free calls free(), it wipes the memory with 0xDA7ADEAD. So, if you see this in your log:

```
nexus_base_mmap: NEXUS_AddrToOffset: unknown address 0xda7adead
nexus_base_mmap: NEXUS_AddrToOffset: unknown address 0xda7adead
nexus_base_mmap: NEXUS_AddrToOffset: unknown address 0xda7adead
nexus_base_mmap: NEXUS_AddrToOffset: unknown address 0xda7adead
```

It means someone is using freed memory. Try to find that.

/proc Filesystem

Nexus provides Linux a `/proc` interface to get module-level debug information at run-time from Nexus.

In kernel mode, you read from each module's `/proc` to get information. This is the standard use of Linux `proc`.

```
# cat /proc/brcm/display
DisplayModule:
display 0: format=1920x1080p(32) 59.940hz
graphics: enabled fb=0x32bd30 1280x720 pixelFormat=19
...
```

In user mode, we have to use a non-standard `proc` interface in order to request the information in the kernel, but provide the information from a user mode process. Instead of reading from the `proc` entry, you write a nexus module name to a generic `/proc/bcmddriver/debug` entry and the module's information will be printed to the kernel's console. For example, enter command on a telnet session:

```
# echo display >/proc/bcmddriver/debug
DisplayModule:
display 0: format=1920x1080p(32) 59.940hz
graphics: enabled fb=0x32bd30 1280x720 pixelFormat=19
...
```

Module names with `proc` output are:

```
# ls /proc/brcm/
astm      audio      audio_graph  audio_output  cec
core display frontend  gpio          graphics2d    hdmi_input
hdmi_output platform sage          simple_decoder surface
surface_compositor sync_channel transport      video_decoder
video_encoder
```

Using GDB and GDBServer

Capturing a Core Dump

This configures Linux to generate a core file:

- Run `ulimit -c unlimited` on the set-top to allow core dumps to be created. It defaults to 0, which means no core file.
- Make sure you are running the application where the current directory has write access (for example, on a hard drive or NFS mount) so the kernel can create a core file. If you are running from read-only flash, it is not going to work.

Analyzing Core Dumps with the GNU Project Debugger GDB

You can use the GNU Project Debugger (GDB), `arm-linux-gdb`, on the build server to analyze core dumps.

Perform the following steps:

1. After you get the core dump, make sure the build server has read permissions to the core file with this command:

```
chmod a+r core
```

2. On the build server, change to the directory that has the application and core file.
3. Start the `gdb` debugger by entering the following:

```
arm-linux-gdb <your_app>
```

4. You may need to tell `gdb` where to find the required shared library files. This is done with something like the following command:

```
set solib-search-path .:  
/opt/toolchains/stbgcc-4.8-1.5/arm-linux-gnueabi/lib
```

The **Tab** key can help complete directory names while typing this command.

5. Load the core file with this command:

```
gdb> core core
```

6. If you cannot get a stack trace, compare the version of `libc` on your root filesystem to the version of `libc` in your toolchain. Change kernel or toolchain to match.

Monitoring a Running (or Hung) Program with GDB

You can use `gdbserver` on the set-top and `arm-linux-gdb` on the build server to monitor a program.

Perform the following steps:

- While the app is running, discover its process ID (PID) using `ps -ef` from another login. You will see multiple entries because of pthreads, so use the lowest number.
- On the set-top, run `gdbserver buildserver:port --attach PID` where `buildserver` is the IP address of your build server and `port` might be 5000.
- On the build server, run `arm-linux-gdb <your_app>`, and then run `target remote <settop>:port` where `<settop>` is the IP address of your set-top box and `port` is the same port you specified in the previous step.
- If the application is hung, one helpful command is `thread apply all bt`.

Instead of attaching with `gdbserver`, a simpler method is to send a ABORT signal to the running or hung process. It will terminate with a core dump. Send the signal from another console as follows:

```
# settop
ps -a
killall -ABRT <APPLICATION>
chmod a+r core

# build server
arm-linux-gdb <APPLICATION> core
gdb>thread apply all bt
```

Kernel Mode Oops

If you are running in kernel mode, a crash in the kernel will generate a kernel Oops. This will include a stack trace. However, the stack trace will only go back to the proxy. It cannot extend back into the application.

If you need to get the stack track into the application, you should either switch to kernel mode, or try to trap the error before the crash, then issue a `BKNI_Fail()` in the user mode `libnexus.so` proxy.

If you need a unified stack trace from kernel to user space, you may need to purchase a MIPS debugging tool (see <http://developer.mips.com/tools/debuggers/>) or ARM debugging tool (see <http://www.arm.com>).

Step-Through and Graphical Debugging

The gdb debugger can be used on the build server to perform step-through debugging. You can also use graphic tools on top of gdb.

Step-through debugging in Linux kernel mode requires patches to the kernel. Broadcom does not offer this support, but information can be found on the Internet.

Be aware that Nexus runs using seven or more threads, and delays in interrupt processing can dramatically alter system behavior. Sometimes a well-placed `printf` can be the best tool.

Address Range Checkers (ARC[H] and MRC)

Overview

Each Broadcom memory controller (MEMC) has a set of Address Range Checkers (ARCs or ARCHs) to monitor and protect memory access based on hardware client IDs.

Nexus programs the ARCs to block writes and to print an ERR on any read or write on any memory outside of Nexus-allocated device memory. Magnum's code for the ARCs is called MRC (memory range checker). It is located at `magnum/commonutils/mrc`. Any ARC violation will appear in the console as ERR (###) with BMRC_MONITOR.

To get familiar with the ARC and MRC, run your Nexus application, then look at the MEMC_0 ARC registers. You will see:

- ARC0 protects all memory below the lowest Nexus/Magnum device memory. This generally protects Linux memory on MEMC0.
- ARC7 (the last ARC) protects all memory above the highest Nexus/Magnum device memory.
- All other ARC's (1 through 6) are programmed to protect unused regions of memory within Nexus device memory.

MRC divides the memory clients (SCB clients) into two categories: Nexus/Magnum controlled and OS (Linux) controlled. We only monitor access by Nexus/Magnum controlled clients. All OS controlled clients are allowed to access any memory.

Interpreting ARC violations

The following is typical output:

```
Address Range Checker 0 (ARC0) has detected a memory access violation in MEMC0
violation access in prohibited range: 0x00000000..0x054811fff
violation start address: 0x00000000
violation end address: 0x0000000f8
violation client: 76(HVD0_OLCPU) request type: 0x028(Linear Write - LW)
transfer length 288 bytes (9 Jwords)
ARC[0]: 0x000000000...0x054812000 0x000000006fffffffffe0000000
blocked[WR] HW: VEC_VBI,VEC,SID,M2MC,BVN,AVD,AUD
match 0x054812000..0x061d06000
next 0x061d06000..0x068834c02
```

From this you can learn:

- HVD (video decoder) went out of bounds
- It was trying to write 288 bytes to address 0x0, which is kernel memory.
- ARC0 blocked it.

Interpreting Secure ARC violations

SAGE configures a set of secure ARC's to protect secure memory. Three levels of notification can occur:

1. When Nexus receives a security violation interrupt, it terminates after printing the following:

```
nexus_platform_core: Detected SECURE MEMC ARCH violation. Terminating....
```

2. For later versions of BFW, Nexus is given information about the secure ARCH violation:

```
nexus_security: MEMC ARCH Violation. MEMC[0]ARCH[2] Addr start [0x035800000]
end[0x035800018] numBlocks[0] scbClientId[251:MSA] requestType[0x1:Linear Read -
LR]
```

3. Nexus configures its own unsecure ARC's to mirror the secure ARC's. This was to get violation information on earlier versions of BFW where secure ARCH violation information was not available.

Interpreting ARC Registers

If you want to read the ARC registers directly, be aware of a few things:

1. All offsets are specified in O-WORD units. One O-WORD is 8 bytes. So multiply all offsets by 8 to get byte offsets.
2. The client lists (ARC_x_READ_RIGHTS and ARC_x_WRITE_RIGHTS) are SCB client IDs. These IDs vary per chip. The meaning of the IDs is stored in `bmrc_clienttable_priv.c`. This table is hard to read. You can use `BMRC_Monitor_PrintClients()` to do a translation from HW bits to human readable client names.
3. You can map offsets to allocations by using MMA debug output. Call `NEXUS_Heap_Dump()` on each heap and you will see output like this:

```

BMEM_ALLOCATED: 0x011ab2000,102400 ,sid/src/bsid_priv.c:53
BMEM_ALLOCATED: 0x011acb000,102400 ,sid/src/bsid_priv.c:53
BMEM_ALLOCATED: 0x011ae4000,102400 ,sid/src/bsid_priv.c:53
BMEM_ALLOCATED: 0x011afd000,102400 ,sid/src/bsid_priv.c:53
BMEM_ALLOCATED: 0x011b16000,1875968 ,xvd/src/bxvd_core_avd_reve0.c:939
BMEM_ALLOCATED: 0x011ce0000,23579968,xvd/src/bxvd_core_avd_reve0.c:1019
BMEM_ALLOCATED: 0x01335cd40,280      , raaga/bdsp_raaga_mm_priv.c:1593
BMEM_ALLOCATED: 0x01335ce80,280      , raaga/bdsp_raaga_mm_priv.c:1593
BMEM_ALLOCATED: 0x01335cfc0,4        , raaga/bdsp_raaga_mm_priv.c:1436
BMEM_ALLOCATED: 0x01335d000,1875968 ,xvd/src/bxvd_core_avd_reve0.c:939
BMEM_ALLOCATED: 0x013527000,9658368 ,xvd/src/bxvd_core_avd_reve0.c:1019
BMEM_ALLOCATED: 0x013e5d000,1875968 ,xvd/src/bxvd_core_avd_reve0.c:939
BMEM_ALLOCATED: 0x014027000,9658368 ,xvd/src/bxvd_core_avd_reve0.c:1019
BMEM_ALLOCATED: 0x01495d000,524288   ,vce/src/bvce_buffer.c:130
BMEM_ALLOCATED: 0x0149dd000,524288   ,vce/src/bvce_buffer.c:130
BMEM_ALLOCATED: 0x014a5d000,524288   ,vce/src/bvce_buffer.c:130

```

Where the first two numbers are offset and size. This should allow you to know what's going on.

Magnum's Auto-Programming of the ARCs

The programming of ARC1...7 is done using based on Magnum MMA allocations. Magnum looks at the file name of the code that's making the allocation (for example, `bxvd_*`) and associates "bxvd" with the AVD HW clients. This automatic association is done based on tables in `bmrc_monitor_clients.c`.

Because ARCs are in limited supply, there is no way to precisely protect all allocations. They must be grouped. The `magnum/commonutils/mrc` code has an algorithm to perform this grouping.

If you would like to program the ARCs manually, Nexus allows you to set the runtime variable `"export custom_arc=y."` Then you can set the ARCs as you please. However, you may need to reboot the set-top box so that leftover ARC configurations do not block you.

GISB Arbiter Timeouts

When software writes to or reads from a register, it assumes that the register is accessible. There is no error return code for register read/write functions. However, sometimes the register is not accessible. This can be for two reasons:

1. The register does not exist. This is a software bug. The register might exist on chip X, but not on chip Y. The software should be fixed to only access registers that exist.
2. The register exists but is inactive due to power management. Power management can power-down or disable clocks to hardware blocks, rendering them non-responsive. From the software's perspective, it's as if the hardware does not exist.

In both cases, the GISB bus will have a timeout. This timeout will raise an interrupt and Linux will print an error like this:

```
[ 3177.516308] brcmstb_gisb_arb_decode_addr: timeout at 0xf1b4edc0 [R timeout], core: leap_0
```

There are no harmless GISB timeout errors for two reasons:

1. When this occurs, the software state is out of sync with the hardware state. It is difficult to predict what might happen.
2. The GISB arbiter can only store one GISB timeout at a time. Therefore, during the time delay between the GISB timeout and Nexus printing and clearing that register, you do not know how many other GISB timeouts may have occurred and whether they are harmful or not. Your only option is to fix the ones you see, and then see if there are any more.

The solution is simple: look up the register address in the RDB, then search the entire code tree (magnum and tree and possibly the application) looking for that register access. You may find more than one. Apply debug code to those sections until you isolate the bug.

Be aware that simple Magnum applications (not using Nexus) will often not register for the GISB timeout interrupt. GISB timeouts can still occur, but they will not be reported on the console. When you first run Nexus, it will report any leftover timeout. You can also use BBS to manually read and clear SUN_GISB_ARB timeout errors.

Magnum Mutex Tracking

Nexus can become “stuck” if calls block for a long time or forever on a mutex. There can be many causes for deadlocks or slow code paths, but fixing the problem usually requires finding out where you are blocking on a mutex.

Magnum’s KNI has a feature to track mutexes that are being held for a long time. Edit `magnum/basemodules/kni/generic/bkni.h` or `linuxkernel/bkni.h` and change `#define BKNI_DEBUG_MUTEX_TRACKING` from 0 to 1. Recompile and run. Look for messages like:

```
Mutex:0x9938383(FILENAME:LINE) was acquired for 100 msec
```

Then go look at that file and line number for context.

Be aware that one deadlock or slow code path will usually cause multiple other paths to back up. You will have to piece together a puzzle to discern cause and symptom.

Nexus Slow Call and Slow Callback Tracking

They will look something like this:

```
*** 00:08:07.627 nexus_statistics_module: surface_compositor[Default]
    nexus/base/src/nexus_base_scheduler.c:828 300193 usec
*** 00:08:18.357 nexus_statistics_callback: Timer callback 0x70778:0xb6307490 from
    modules/surface_compositor/src/nexus_surface_compositor.c:242 700589 usec
```

We warn on anything over 100 msec (i.e. 100000 usec). A bit over 100 msec is normal (unfortunately), but if you see 5 second, 10 second or 20 second delays, something is going very wrong.

Nexus Stuck Callback Tracking

If any Nexus callback is stuck for more than 5 seconds, you'll see something like this for user mode:

```
stuck callback FILENAME:LINE for 5200 msec
```

and this for kernel mode:

```
stuck callback 0x3030338 for 5100 msec. You can go from function pointer to name using 'nm'.
```

The kernel mode one requires the extra step of debug to translate to `FILENAME:LINE`. But it points you in the right direction. Please take the hint and figure out which callback is stuck. Then trace through the app to see why it might be stuck.

Runaway Interrupts

Look for a message like "#### BVNF0 (W0, bit 12) fired 10000 times" which could give a clue about which interrupt is run-away. Then debug SW in that direction.

If user mode, set to non-real-time interrupt handling. Run with "export not_realtime_isr=y". Your interrupt performance will drop, but a runaway interrupt should hopefully not cause the box to hang.

There is no equivalent for kernel mode.

Nexus Call Trace

Build Nexus with NEXUS_EXTRA_CFLAGS=-DNEXUS_P_TRACE=1, then run msg_modules=nexus_trace*. You will see output like this:

```
nexus_trace_graphics2d: >NEXUS_Graphics2D_Fill: handle:0x84f044a8 pSetting
nexus_trace_graphics2d: <NEXUS_Graphics2D_Fill(0): result:0 handle:0x84f04
nexus_trace_graphics2d: >NEXUS_Graphics2D_Fill: handle:0x84f044a8 pSetting
nexus_trace_graphics2d: <NEXUS_Graphics2D_Fill(0): result:0 handle:0x84f04
nexus_trace_graphics2d: >NEXUS_Graphics2D_Checkpoint: handle:0x84f044a8 pL
nexus_trace_graphics2d: <NEXUS_Graphics2D_Checkpoint(9): result:16842754 h
nexus_trace_surface_compositor: >NEXUS_SurfaceClient_PushSurface: handle:0
nexus_trace_surface_compositor: <NEXUS_SurfaceClient_PushSurface(0): resul
nexus_trace_surface_compositor: >NEXUS_SurfaceClient_RecycleSurface: handl
nexus_trace_surface_compositor: <NEXUS_SurfaceClient_RecycleSurface(0): re
nexus_trace_surface_compositor: >NEXUS_SurfaceClient_RecycleSurface: handl
nexus_trace_surface_compositor: <NEXUS_SurfaceClient_RecycleSurface(0): re
nexus_trace_surface_compositor: >NEXUS_SurfaceClient_RecycleSurface: handl
nexus_trace_surface_compositor: <NEXUS_SurfaceClient_RecycleSurface(0): re
nexus_trace_graphics2d: >NEXUS_Graphics2D_Fill: handle:0x84f044a8 pSetting
nexus_trace_graphics2d: <NEXUS_Graphics2D_Fill(0): result:0 handle:0x84f04
nexus_trace_graphics2d: >NEXUS_Graphics2D_Fill: handle:0x84f044a8 pSetting
nexus_trace_graphics2d: <NEXUS_Graphics2D_Fill(0): result:0 handle:0x84f04
nexus_trace_graphics2d: >NEXUS_Graphics2D_Checkpoint: handle:0x84f044a8 pL
nexus_trace_graphics2d: <NEXUS_Graphics2D_Checkpoint(9): result:16842754 h
nexus_trace_surface_compositor: >NEXUS_SurfaceClient_PushSurface: handle:0
nexus_trace_surface_compositor: <NEXUS_SurfaceClient_PushSurface(0): resul
nexus_trace_surface_compositor: >NEXUS_SurfaceClient_RecycleSurface: handl
nexus_trace_surface_compositor: <NEXUS_SurfaceClient_RecycleSurface(0): re
nexus_trace_surface_compositor: >NEXUS_SurfaceClient_RecycleSurface: handl
nexus_trace_surface_compositor: <NEXUS_SurfaceClient_RecycleSurface(0): re
```

The trace only works for calls going across the proxy, not for single process user mode calls. To see all public API calls, including inter-module calls, use NEXUS_P_SYNC_TRACE.

Tracelog

Tracelog is a hardware block which snoops register (GISB) and memory (SCB) busses and can log accesses into on-chip memory or DRAM based on certain triggers.

There is a web-based tool that works with The Broadcom EJTAG debugger. However, it usually requires a touch to the software to learn memory locations.

Refsw now includes a small tracelog utility which can be inserted into applications or Nexus/Magnum to exercise the same functionality, at the point where those memory locations are known. The results are printed to the console. The following shows a capture of interleaved memory (MEM) and register (IO) accesses at time T and address A from the M2MC and CPU hardware clients.

```
T:0x17979ca38 A:0x06a9a6dc0 WR MEM(M2MC_2,149) RR Linear(64 bytes) ID:2
T:0x17979ca6a A:0x06a9a7900 WR MEM(M2MC_2,149) RR Linear(64 bytes) ID:2
T:0x17979ca8d A:0x06a9a8440 WR MEM(M2MC_2,149) RR Linear(64 bytes) ID:2
T:0x17979cad9 A:0x06a9a8f80 WR MEM(M2MC_2,149) RR Linear(64 bytes) ID:2
T:0x17979cb06 A:0x06a9a9ac0 WR MEM(M2MC_2,149) RR Linear(64 bytes) ID:2
T:0x17979cb31 A:0x06a9aa600 WR MEM(M2MC_2,149) RR Linear(64 bytes) ID:2
T:0x17979cb4c A:0x06a9ab140 WR MEM(M2MC_2,149) RR Linear(64 bytes) ID:2
T:0x17979cb78 A:0x06a9abc80 WR MEM(M2MC_2,149) RR Linear(64 bytes) ID:2
T:0x17979cb93 A:0x06a9ac7c0 WR MEM(M2MC_2,149) RR Linear(64 bytes) ID:2
T:0x17979cbbb A:0x06fb34880 RD MEM(M2MC_0,147) RR Linear(32 bytes) ID:1
T:0x17979cc18 A:0x06fb348a0 RD MEM(M2MC_0,147) RR Linear(256 bytes) ID:8
T:0x17979ce4d A:0x06fb349a0 RD MEM(M2MC_0,147) RR Linear(64 bytes) ID:2
T:0x17979cf57 A:0x06fb4e380 WR MEM(M2MC_2,149) RR Linear(32 bytes) ID:1
T:0x1797a81f6 A:0x0f09b0018 RD IO(CPU_0,6)
T:0x1797a829a A:0x0f09b001c RD IO(CPU_0,6)
T:0x1797a832a A:0x0f09b0010 RD IO(CPU_0,6)
T:0x1797a8b57 A:0x06fb4e300 RD MEM(HOST_CPU_MCP_R_HI,200) PR Cache(256 bytes) ID:8
T:0x17979c070 A:0x06a986840 WR MEM(M2MC_2,149) RR Linear(64 bytes) ID:2
T:0x17979c0b4 A:0x06a987380 WR MEM(M2MC_2,149) RR Linear(64 bytes) ID:2
T:0x17979c0e3 A:0x06a987ec0 WR MEM(M2MC_2,149) RR Linear(64 bytes) ID:2
T:0x17979c111 A:0x06a988a00 WR MEM(M2MC_2,149) RR Linear(64 bytes) ID:2
T:0x17979c12c A:0x06a989540 WR MEM(M2MC_2,149) RR Linear(64 bytes) ID:2
T:0x17979c1c9 A:0x06a98a080 WR MEM(M2MC_2,149) RR Linear(64 bytes) ID:2
T:0x17979c1f1 A:0x06a98abc0 WR MEM(M2MC_2,149) RR Linear(64 bytes) ID:2
```

See [rockford/applications/nexus/memc/tracelog_example.c](#) for a starting point.

Debugging Real-Time Threads

Some customers make extensive use of pthread SCHED_FIFO and SCHED_RR real-time scheduling. Any use of real-time (RT) threads requires special care.

Background on Real-Time Threads

Linux real-time threading is provided with NPTL and the pthread interface.

1. By default, pthreads are non-real time. This is called SCHED_OTHER.
2. Two flavors of RT thread scheduling are available: SCHED_FIFO and SCHED_RR.
3. RT threads can be preempted. SCHED_RR and SCHED_FIFO threads will be preempted by higher priority RT threads. SCHED_RR threads will be preempted with equal priority SCHED_RR threads.
4. Any RT thread will take priority over any non-RT thread. If there is any SCHED_FIFO or SCHED_RR work to do, SCHED_OTHER threads will not run.

Magnum and RT Threads

Magnum ISR processing is vulnerable to priority inversion in a system with real-time threads. In Nexus user mode, the Magnum ISR thread is set to be a high-priority SCHED_FIFO real-time thread. This thread calls only Nexus and Magnum ISR code. Magnum ISR code is also synchronized with non-ISR code using BKNI_EnterCriticalSection(). Any application thread, including SCHED_OTHER threads, can enter Nexus as a task call and then call BKNI_EnterCriticalSection().

The danger occurs because BKNI_EnterCriticalSection() does not prevent task switches. This means that while that SCHED_OTHER task is holding the Magnum critical section, it might be preempted by another thread. If that thread is not RT, then the critical section can be delayed for a few time slices. If that thread is RT, then it could be delayed forever. Even if the preempting RT thread is lower priority than the Magnum ISR RT thread, it can starve the Magnum ISR thread because the Magnum ISR thread is waiting on the SCHED_OTHER task to release the critical section. This situation is called priority inversion.

To solve this problem, KNI opens the critical section pthread mutex with the PTHREAD_PRIO_INHERIT attribute. If your version of libc does not support this, you should consider upgrading.

Debugging RT Scheduling Problems

You can determine if there are any RT threads in your system using “top.” You will see the letters “RT” in the PR column.

```
top - 00:30:13 up 30 min,  0 users,  load average: 0.00, 0.00, 0.00
Tasks:  31 total,   1 running, 30 sleeping,   0 stopped,   0 zombie
Cpu(s):  0.0%us,   0.1%sy,   0.0%ni, 99.9%id,   0.0%wa,   0.0%hi,   0.0%si,
0.0%st
Mem:   319464k total,   15132k used,   304332k free,        0k buffers
Swap:        0k total,        0k used,        0k free,   7240k cached
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
58	root	15	0	1380	816	432	R	2	0.3	0:00.10	top
2	root	RT	0	0	0	0	S	0	0.0	0:00.00	migration/0
3	root	34	19	0	0	0	S	0	0.0	0:00.00	ksoftirqd/0
4	root	RT	0	0	0	0	S	0	0.0	0:00.00	migration/1
1	root	18	0	1204	376	328	S	0	0.1	0:02.64	init
6	root	10	-5	0	0	0	S	0	0.0	0:00.00	events/0
7	root	10	-5	0	0	0	S	0	0.0	0:00.00	events/1
8	root	20	-5	0	0	0	S	0	0.0	0:00.00	khelper

Broadcom recommends disabling all RT scheduling in the system to prove this is the problem. Most customers are reluctant to do this because their RT systems are designed to be a careful calibrated system of dozens of RT threads. But your goal is to identify whether a priority-inversion problem exists or not. Minor service interrupts caused by non-RT scheduling should not be fatal to the system and will get you a valuable debug data point.

Instead of trying to root out every RT thread in application code, just modify the kernel to fake out the scheduling priority, like this:

```
static int
do_sched_setscheduler(pid_t pid, int policy, struct sched_param __user
*param)
{
    struct sched_param lparam;
    struct task_struct *p;
    int retval;

    return 0; /* this prevents all RT scheduling */

    if (!param || pid < 0)
        return -EINVAL;
    if (copy_from_user(&lparam, param, sizeof(struct sched_param)))
        return -EFAULT;

    ...
}
```

Solutions for RT Scheduling Problems

Please consider not using any RT threads in your application. Instead, use “nice” numbers in the PR column to adjust non-real-time scheduler priority. The scheduler will use the “nice” value to portion out scheduler time. This will allow higher priority threads to get more CPU time. This is usually the general desire that motivates RT scheduling.

If you must use RT threads, make minimal use of them. RT threads should only be used for work with hard real time CPU processing requirements.

All RT threads should be fast and not CPU-bound. If it is CPU-bound, then the thread will never yield and allow a lower priority or a non-real time thread to run. Be aware that it is hard for SW engineers to write RT code with this system-wide sensitivity. Someone with that system-level view should inspect all code runs in RT threads.

If you have any RT threads, you will need to consider making them all threads that call Nexus or Magnum RT threads. This is the only way to prevent the priority inversion that starves out the Magnum ISR thread.

The more RT threads you add to the system, the greater chance of having other cases of priority inversion and live lock. You must be careful.

If you want to identify the thread or group of threads that’s causing the problem, you’ll need to slowly modify your system, likely one thread at a time until you identify the problematic code. Be aware that it could be the combined effect of several RT threads and not just one RT thread.

Appendix A: Setting Run-Time Variables

Nexus has a variety of run-time variables that are useful for debugging. Inside Nexus, `NEXUS_GetEnv()` retrieves these variables. For Linux user mode, these configuration variables are set using environment variables. For Linux kernel mode, other techniques are provided.

Here is an example of how to set some variables in Linux user mode:

```
export msg_modules=nexus_video_decoder_priv,BMMA_Alloc
export force_vsync=y
nexus decode
```

In the kernel mode configuration, variables can be passed in using the configuration environment variable, using the following syntax:

```
export config="msg_modules=BMMA_Alloc,nexus_core force_vsync=y"
nexus decode
```

If you set `msg_modules` but don't set `config`, the nexus script will automatically export `config="msg_modules=$msg_modules"`.

Setting runtime variables after insmod

Some configuration options can also be set in Linux kernel mode after `insmod` using the `/proc` interface. For a Linux kernel do this:

```
echo "force_vsync=y" >/proc/brcm/config
echo "no_watchdog=y" >/proc/brcm/config
```

However, after Nexus starts, setting these variables may have no effect. The only sure way is to set runtime variables before Nexus starts.