Reference Software

Debugging Guide

Broadcom
Corporate Headquarters: San Jose, CA

Web: www.broadcom.com

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	**Updated:** • The document title to “Reference Software Diagnostic Tools” **Added:** • “Address Range Checkers” on page 13 • “GISB Arbiter Timeouts” on page 16 • “Debugging Real-Time Threads” on page 17 **Removed:** References to the BCM740X, changing them to a larger set of chips.
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

Table of Contents

Introduction 3
Debug Logging 4
  B_REFSW_DEBUG Build Options 5
  Capturing Debug Output 6
  Module-Level MSG Control 7
    Run-Time Alternatives to the msg_modules Environment Variable 7
  Interpreting BDBG_OBJECT Asserts in the debug log 8
  Interpreting BERR_TRACE in the debug log 8
Proc Filesystem 9
Video and Audio Decoder UART Output 11
Using GDB and GDBServer 12
  Capturing a Core Dump 12
  Analyzing Core Dumps with the GNU Project Debugger GDB 12
  Monitoring a Running (or Hung) Program with GDB 12
  Kernel Mode Oops 13
  Step-Through and Graphical Debugging 13
Address Range Checkers (BMRC_MONITOR) 14
  Overview 14
  Interpreting ARC Registers 14
  Magnum's Auto-Programming of the ARCs 15
  Getting Results 16
  What Can Corrupt the Kernel? 16
GISB Arbiter Timeouts 17
Debugging Real-Time Threads 18
  Background on Real-Time Threads 18
  Magnum and RT Threads 18
  Debugging RT Scheduling Problems 19
  Solutions for RT Scheduling Problems 20
Appendix A: Setting Run-Time Variables 21

Introduction

Debugging software on embedded systems has never been easy. This document describes the debug capabilities we've added to 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 *all* 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
B_REFSW_DEBUG_LEVEL={err|log|wrn}	Adjust the global debug level. B_REFSW_DEBUG=n sets this to “log”.
B_REFSW_DEBUG_COMPACT_ERR=y	Convert all BDBG_ERR and BERR_TRACE to a printf of file and line number. This reduces code size. B_REFSW_DEBUG=n and =minimal set this to “y”.
B_REFSW_DEBUG_ASSERT_FAIL=n	Change BDBG_ASSERT into a print, but no BKNI_Fail. B_REFSW_DEBUG=n and =minimal 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.

Capturing Debug Output

Nexus starts a separate “logger” process to print DBG output. This allows DBG output from ISR context to only “print” to memory and not be blocked by slow UART output. It also allows Nexus to properly interleave lines of kernel and user mode DBG output. The logger process does introduce some delay before DBG output is seen on the console.

See nexus/utils/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
```

The logger app can be bypassed by running with export debug_log_size=0. Debug output will then print immediately to the console. UART delays will be experienced by the code and kernel and user mode prints may interleave in unreadable ways.

If debug output is getting lost before a system reset, you may need to run debug_log_size=0 to get an immediate print before the reset.

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:

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

See *Appendix A: Setting Run-Time Variables* for details on setting msg_modules at run-time in Linux kernel and user modes.

Run-Time Alternatives to the msg_modules Environment Variable

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 debug debug level.

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

Interpreting BDBG_OBJECT Asserts in the debug log

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:

```
!!! Assert '(window) &&
(window)->bdbg_object_NEXUS_VideoWindow.bdbg_obj_id==bdbg_id__NEXUS_Vi
deoWindow' Failed at
nexus/modules/display/7405/src/nexus_video_window.c:733
```

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.

Interpreting BERR_TRACE in the debug log

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.

Proc Filesystem

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

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

```
# ls /proc/brcm
audio        debug        hdmi_output   video_decoder
config       display      sync_channel

# cat /proc/brcm/display
DisplayModule:
display 0: format=1920x1080p(32) 59.940hz
  graphics: enabled fb=0x32bd30 1280x720 pixelFormat=19
          vsync count=216425, line=79, errors=0
  window 0: visible=y, position=0,0,1920,1080, zorder=0, ar=box,
NEXUS_VideoInput=0x858129a8
          vsync count=216425, line=80, errors=0
  output 0x348ce8: hdmi
  output 0x348be0: component
display 1: format=720x480i(1) 59.940hz
  graphics: enabled fb=0x348a88 720x480 pixelFormat=19
          vsync count=216426, line=260, errors=0
  window 0: visible=y, position=0,0,720,480, zorder=0, ar=box,
NEXUS_VideoInput=0x858129a8
          vsync count=216426, line=260, errors=0
  output 0x348c90: rfm
  output 0x348c38: composite
NEXUS_VideoInput 0x858129a8: link=0x857b14d8, decoder, eMpeg4, ref_cnt=2,
errors=0
```

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/bcmdriver/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/bcmdriver/debug
```

And you see results on the console.

Module names with proc output are:

```
  # ls /proc/brcm/
  audio             debug            hdmi_output        sync_channel
  audio_output     display           platform          transport
  cec              frontend           simple_decoder     video_decoder
  config           graphics2d         surface
  core              hdmi_input        surface_compositor
```

Video and Audio Decoder UART Output

Both the HVD video decoder and RAAGA audio decoder output valuable debug information to specific UART's. If you don't have easy access the AVD and RAAGA, Nexus can redirect this information to the Linux console.

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

```
# HVD0
export avd_monitor=0

# HVD0 and HVD1
export avd_monitor=01

# 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:
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:
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.

Raaga 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.

Using GDB and GDBServer

Capturing a Core Dump

Before you can use GDB in the `uclibc` environment, do the following:

- 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. Tell gdb where to find the required `uclibc` shared library files. This is done via the following command:

```
set solib-search-path .:/opt/toolchains/<version>/arm-linux-uclibc/lib
```

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

5. Load the core file with this command:

```
gdb> core core
```

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 settop 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:

```
ps -a
# find the PID
kill -ABRT <PID>
```

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 (BMRC_MONITOR)

Overview

Each Broadcom memory controller (MEMC) has a set of Address Range Checkers (ARCs) 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 Registers

When reading ARC registers, be aware of two 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…3 is done using based on Magnum MEM 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 settop so that leftover ARC configurations don't block you.

Getting Results

You can get results in one of three ways:

1. Monitor the ARC_x_VIOLATION registers.
2. Set the ARC clients to block read or write access. This is especially useful for using ARC0 to protect kernel corruptions from user mode bugs. Edit bmrc_monitor.c and change BMRC_Checker_SetBlock to BMRC_AccessType_eWrite or BMRC_AccessType_eBoth for the arc_no you want.
3. Look for BMRC_MONITOR errors on the console. They will look something like this:

```
### 00:30:55.426 BMRC_MONITOR: memory range checker error for MEMC 0 ARC 1
### 00:30:55.427 BMRC_MONITOR: violation access in prohibited range: 0x5218900..0xdee7000
### 00:30:55.427 BMRC_MONITOR: violation start address: 0xa5a3a40
### 00:30:55.428 BMRC_MONITOR: violation end address:   0xa5a3a58
### 00:30:55.428 BMRC_MONITOR: violation client: 6(AVD_OLA_0)  request type: 0x001(LR)
### 00:30:55.428 BMRC_MONITOR: transfer length (Jwords) 1
```

What Can Corrupt the Kernel?

If your kernel crashes with an apparent memory corruption, there are a limited number of causes:

1. It could be a bug in the kernel itself.
2. It could be a bug in any kernel mode driver code. Kernel mode SW has free access to all kernel memory.
3. It could be a bug in user mode code, but only by means of incorrectly programming a HW device that then corrupts the kernel memory. Using ARC0 to block writes to the kernel can eliminate this cause.

Of course, if you use ARC0 to protect your kernel, you will not be fixing the bug but only masking it. It may be enough, however, to get you into production and/or it could give you valuable debug information for getting the bug fixed.

Even when you get the bug fixed, you should still leave the ARC protection enabled to catch the next bug.

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.

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 sensitive 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 into 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.

Linux kernel mode does not improve the situation. BKNI_EnterCriticalSection() does not prevent task switches in kernel mode either. The only way to prevent task switches is to disable all interrupts. Magnum ISR code is too slow (sometimes up to 1 millisecond) to disable interrupts. With Linux running at 1000 Hz, disabling interrupts for that length of time causes system failures.

It is often believed that adjusting RT thread priorities should solve problems like this. It does not. In a priority-inversion problem like this, the priority of the Magnum ISR thread is irrelevant and the priority of the RT thread that preempted the SCHED_OTHER task is irrelevant.

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.