Reference Software

Debugging Guide

Broadcom Corporation
5300 California Avenue
Irvine, California, USA 92677
Phone: 949-926-5000
Fax: 949-926-5203

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

Table of Contents

Introduction

Debugging software on embedded systems has never been easy, but Broadcom has built a few checks into its Magnum and Nexus Reference Software that may provide some help. This document provides some tips and starting points for debugging code.

This document is written primarily for the Linux environment.

Setting Run-Time Variables

Brutus and Nexus have a variety of run-time variables that are useful for debugging. Some of these are described in Section 15 of the *Brutus Usage Guide* (STB_Brutus-SWUM30x-R). 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,BMEM
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:

```
config="msg_modules=BMEM,nexus_core force_vsync=y"
nexus decode
```

Configuration options can also be set 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
```

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

Magnum Debug Interface

Overview

The Magnum Debug Interface (DBG) is part of the porting interface's base modules. All code, from porting interface to Nexus to Brutus, uses the Debug Interface to write status and error messages to the console.

The debug interface is designed so that all debug messages can also be compiled out. Broadcom calls this "release mode." By default, reference software compiles in "debug mode." You can compile in release mode by setting `B_REFSW_DEBUG=n` in the environment. Be warned that **release builds have no error messages**; there is no indication that the build is functional. Very little code-level support can be given to applications running in release mode.

As an alternative, we recommend you compile `B_REFSW_DEBUG=y`, but then set `B_REFSW_DEBUG_LEVEL=wrn` to compile out MSG-level. This will leave all errors and warnings in place.

In debug mode you can control which messages are displayed. You seldom want all messages displayed because it will seriously degrade performance and the system will not function normally.

Debug output can be controlled either on a global level, or on a module-by-module level.

As provided in Table 1, all debug output is categorized into four levels.

Table 1: The Four Levels of Debug Output

Level	Abbreviation	Description
Error	ERR	Errors in the code; some errors are "normal," but in general every error should be well understood. These errors are denoted with the `###` prefix.
Warning	WRN	Warning notification; the warnings should be comprehensible to all developers. The number of warnings should be limited in order to not degrade system performance. These warnings are denoted by a `***` prefix.
Message	MSG	Internal debugging; the use of MSG has no restrictions. Some of the messages may be obscure, and the number of messages may overwhelm the system. These messages are denoted with the `---` prefix.

Trace	TRACE	TRACE turns on the BDBG_ENTER/LEAVE macros. Here is the code:
		`#define BDBG_ENTER(function) ((BDBG_eTrace >= dbg_module.level)? BDBG_EnterFunction(&dbg_module, #function) : (void)0)`
		`#define BDBG_LEAVE(function) ((BDBG_eTrace >= dbg_module.level)? BDBG_LeaveFunction(&dbg_module, #function) : (void)0)`
		Here is an example:
		`int foo()`
		`{`
		`  BDBG_ENTER(foo);`
		`  do_foo();`
		`  BDBG_LEAVE(foo);`
		`}`

All debug output on Linux is directed to stderr, not stdout. If you want to capture or grep this, be sure to handle this, as follows:

```
nexus decode 2>&1|grep SEARCH_TEXT
nexus decode 2>capture_file
```

In the Nexus and Brutus, ERR and WRN messages are displayed by default. Enabling global MSG level is highly discouraged. There are so many MSG-level messages that your system performance will degrade and it may not be operable at all. MSG level output, however, is very useful on a module-by-module basis.

When running in Linux kernel mode, the Nexus will collect all of the kernel mode output and pipe it to the console running the proxy layer, such as telnet.

Module-Level MSG Control

MSG-level messages can be enabled per 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,BMEM
nexus decode
```

See

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

If you are running Brutus with the `--exec` option, you can enable modules as follows:

```
settop brutus --exec
Brutus>dbglevel(bvdc,msg)
```

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 "BMEM 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.

BDBG_OBJECT Checks

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_Video
Window' 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.

Using Linux /proc for run-time module status

Nexus provides Linux a `/proc` interface to get module-level debug information at run-time from any application using 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=1
  graphics: vdcWindow=82c54000 fb=8049c280 720x1080 pixelFormat=19
  window 0: visible=1, position=260,115,360,270, NEXUS_VideoInput=c1377770
  window 1: visible=0, position=360,0,360,240, NEXUS_VideoInput=00000000
  output 80465b80
  output 80465c00
  output 80465c80
  output 80464100
NEXUS_VideoInput c1377770: link=825ae800, BAVC_SourceId_eMpeg0, ref_cnt=1
     NEXUS_VideoInputResumeMode_eAuto, in source_pending? y
```

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 like this on the console:

```
nexus_display_module: DisplayModule:
nexus_display_module: display 0: format=1
nexus_display_module:   graphics: vdcWindow=(nil) fb=(nil) 0x0
pixelFormat=19
nexus_display_module:   window 0: visible=1, position=0,0,720,480,
NEXUS_VideoInput=0x2b225eb0
nexus_display_module:   output 0x4c53c8
nexus_display_module:   output 0x4c53a0
nexus_display_module: NEXUS_VideoInput 0x2b225eb0: link=0x4c5bb0,
BAVC_SourceId_eMpeg0, ref_cnt=1
nexus_display_module:      NEXUS_VideoInputResumeMode_eAuto, in
source_pending? y
```

Module names should match the directory name where the module source code is stored.

Common module names include:

- audio
- display
- video_decoder
- video_encoder
- transport

Decoder UART Output

Both the AVD 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 AVD0 output, set this at runtime before starting Nexus.

```
export avd_monitor=0
```

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:
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:
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
```

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 Brutus 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), mipsel-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:

   ```
   mipsel-linux-gdb brutus
   ```

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>/mipsel-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 mipsel-linux-gdb on the build server to monitor a program.

Perform the following steps:

- While Brutus 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 `mipsel-linux-gdb brutus`, 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 Brutus 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/.

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 Brutus/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

Overview

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

By default, Nexus will configure Magnum to automatically program the ARCs to monitor Magnum and Nexus heap allocations. Magnum's code for the ARCs is called MRC (memory range checker). It is located at magnum/commonutils/mrc.

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

1. ARC0 is programmed to protect the kernel from all non-kernel HW clients.
2. ARC1 through 3 are programmed to protect various ranges with various client lists.

A "kernel HW client" is one that the kernel configures and which normally accesses kernel memory. These include MIPS, USB, SATA, and other clients. All other HW clients, including audio, video, and transport-related clients, should not read or write to/from kernel 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 MEM debug output. Run your application with export `msg_modules=BMEM`. You will see output like this:

```
--- 00:00:00.718 BMEM: BMEM_AllocAligned(heap 0x53a038), offset=0x43c6900,
             ptr=0x2c5c6900,size=24883200, align=8,
             code=magnum/portinginterface/vdc/7405/
             bvdc_bufferheap_priv.c:711)
--- 00:00:00.719 BMEM: BMEM_AllocAligned(heap 0x53a038), offset=0x5b81940,
             ptr=0x2dd81940,size=1792, align=2,
             code=magnum/portinginterface/rap/7405/brap_priv.c:2996)
--- 00:00:00.719 BMEM: BMEM_AllocAligned(heap 0x53a038), offset=0x5b82100,
             ptr=0x2dd82100,size=4147200, align=8,
             code=magnum/portinginterface/vdc/7405/
             bvdc_bufferheap_priv.c:711)
--- 00:00:00.720 BMEM: BMEM_AllocAligned(heap 0x53a038), offset=0x5f76940,
            ptr=0x2e176940,size=30400, align=2,
            code=magnum/portinginterface/rap/7405/brap_priv.c:2996)
--- 00:00:01.054 BMEM: BMEM_AllocAligned(heap 0x53a038), offset=0x5f7ec40,
            ptr=0x2e17ec40,size=5056, align=2,
            code=magnum/portinginterface/rap/7405/brap_priv.c:2996)
--- 00:00:01.055 BMEM: BMEM_AllocAligned(heap 0x53a038), offset=0x5f80040,
            ptr=0x2e180040,size=53056, align=2,
            code=magnum/portinginterface/rap/7405/brap_priv.c:2996)
--- 00:00:01.055 BMEM: BMEM_AllocAligned(heap 0x53a038), offset=0x5f8cfc0,
            ptr=0x2e18cfc0, size=57152, align=2,
            code=magnum/portinginterface/rap/7405/brap_priv.c:2996)
--- 00:00:01.056 BMEM: BMEM_AllocAligned(heap 0x53a038), offset=0x5f9af40,
            ptr=0x2e19af40,size=379648, align=2,
            code=magnum/portinginterface/rap/7405/brap_priv.c:2996)
--- 00:00:01.056 BMEM: BMEM_AllocAligned(heap 0x53a038), offset=0x5ff7a80,
            ptr=0x2e1f7a80,size=10816, align=2,
            code=magnum/portinginterface/rap/7405/brap_priv.c:2996)
```

These offsets and sizes 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.

Jailing Clients

The MRC auto-programming algorithm allows you to priorities certain sets of clients for more focused debug. This will allow you to monitor certain clients more closely and other clients more loosely (or possibly not at all).

Nexus supports the following jails:

```
export jail_vdc=y
export jail_xvd=y
export jail_rap=y
export jail_xpt=y
```

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. nexus_platform_core.c registers for this interrupt and will print an error message on the console like this:

```
### 00:00:26.019 nexus_platform_core: **********************************
### 00:00:26.019 nexus_platform_core: GISB Timeout reading addr 0x10bb0590 by core CPU
### 00:00:26.019 nexus_platform_core: **********************************
```

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.

Brutus Tips

Here are a few hints to divide and conquer when you are having a problem building or running the Brutus reference application.

Cannot Build

If mipsel-linux-gcc cannot be found (or some other part of the toolchain), you need to install your toolchain or set the path. See the *Reference Software User Manual* appendices for help.

Be aware that a single error can generate volumes of C compiler messages. Run `make 2>&1|more` to find the first error.

Brutus Crashes

The first thing is to do a `make clean && make` and try again.

There can be compiler dependency problems.

Process the core dump using mipsel-linux-gdb (see above for notes). Type `bt 20` to view a stack trace. See if that tells you where the problem is.

Notice that the core file may be called `core` with a process ID suffix. It can be hard to know which core file you want. Remove all core files and recreate the problem.

Back up to a previous good state (or previous release) and see if you can get it working. If you can, `diff` the code.

Brutus Is Running But There Is No Decoding, Recording, Playing, Etc.

In this situation, Broadcom has provided a set of utilities within the context of Nexus to test decode capabilities and PVR issues. These utilities are found in nexus /utils.

The utilities are built as follows:

```
cd nexus/utils
make
```

To see command-line options, enter the following:

```
nexus decode
nexus playback
nexus record
```

Here is a sample commands:

```
nexus decode –video 0x11 –audio 0x14
nexus playback stream.mpg stream.nav
playback>?
playback>rate(2)
```

Other Tips

- Run `settop brutus --help` and `settop brutus --help-cfg` to see a large array of options.
- Turn off live decoding by entering `settop brutus –ch none`.
- Turn off audio or video decode by adding `VIDEODECODE_ENABLED=0` or `AUDIODECODE_ENABLED=0` to brutus.cfg.
- You can view the current audio/video PTS and the current STC using these commands:

```
export monitor=1
settop brutus
```