Power Management Overview

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

Revision History

Revision	*Date*	*Change Description*
STB_RefSw-SWUM401-R	11/23/11	**Updated:** • Table 1: "The Four Nexus Power States," on page 6 and the text below • "The Active Standby State (S1)" on page 7 • "The Passive Standby State (S2)" on page 8 • The bulleted list in "Linux Power Management" on page 12, adding two more bullets **Added:** • "The Deep Sleep Standby State" on page 10 • Table 2: "Nexus Dynamic Power Management Functions," on page 11 • "Kernel Power State Changes" on page 12 • "SATA" on page 15 • "USB" on page 16 • "Network Interfaces" on page 16 • "Wake-Up Events" on page 17 • "sysfs Attributes" on page 18
STB_RefSw-SWUM400-R	06/13/11	Initial release

Broadcom Corporation
5300 California Avenue
Irvine, CA 92617

Table of Contents

About This Document 5
Purpose and Audience 5
Acronyms and Abbreviations 5
Document Conventions 5
Technical Support 5
Introduction 6
Power States 6
The Active Standby State (S1) 7
The Passive Standby State (S2) 8
Setting Up Wake-Up Devices 8
The Deep Sleep Standby State 10
Dynamic Power Management 11
Nexus Functions That Enable Dynamic PM 11
Linux Power Management 12
Kernel Power State Changes 12
Full Power 12
Standby 13
Suspend-to-RAM 13
Halt (S3 with Cold Boot) 13
pmlib 14
SATA 15
USB 16
Network Interfaces 16
Ethernet 16
MOCA 17
Wake-Up Events 17
Wake-up Timer 17
Ethernet Wake-on-LAN 17
sysfs Attributes 18
Broadcom-Specific Attributes 19
memc1_power 19
standby_flags 19
ddr_timeout 19
time_at_wakeup 20

List of Tables

Table 1: The Four Nexus Power States 6
Table 2: Nexus Dynamic Power Management Functions 11
Table 3: System Suspend Standby Flags 19

About This Document

Purpose and Audience

This document describes the design and implementation of power management support in the Broadcom Set-Top Box (STB) Reference Software stack.

The primary intended audience for this document is application and middleware developers writing software on top of Nexus. It can also apply to customers calling Magnum directly.

This document applies to range of Broadcom chips and does not have detail about chip-specific power targets, boot sequence, or similar topics. Please consult the chip application notes for this information.

Acronyms and Abbreviations

In most cases, acronyms and abbreviations are defined on first use.

For a comprehensive list of acronyms and other terms used in Broadcom documents, go to: http://www.broadcom.com/press/glossary.php.

Document Conventions

The following conventions may be used in this document:

Convention	*Description*
Bold	User input and actions: for example, type **exit**, click **OK,** press **Alt+C**
`Monospace`	Code: `#include <iostream>` HTML: `<td rowspan = 3>` Command line commands and parameters: `wl [-l] <command>`
< >	Placeholders for *required* elements: enter your <username> or `wl <command>`
[ ]	Indicates *optional* command-line parameters: `wl [-l]` Indicates bit and byte ranges (inclusive): [0:3] or [7:0]

Technical Support

Broadcom provides customer access to a wide range of information, including technical documentation, schematic diagrams, product bill of materials, PCB layout information, and software updates through its customer support portal (https://support.broadcom.com). For a CSP account, contact your Sales or Engineering support representative.

In addition, Broadcom provides other product support through its Downloads & Support site (http://www.broadcom.com/support/).

Introduction

This document describes the design and implementation of power management support in the Broadcom Set-Top Box (STB) Reference Software stack.

The primary intended audience for this document is application and middleware developers writing software on top of Nexus. It can also apply to customers calling Magnum directly.

This document applies to range of Broadcom chips and does not have detail about chip-specific power targets, boot sequence, or similar topics. Please consult the chip application notes for this information.

Power States

From the Nexus perspective, Broadcom defines four power states, given in Table 1.

Table 1: The Four Nexus Power States

Power State	*Functionality*	*Power Level*
On (S0)	Display, decode, and record are all active.	Full power
Active Standby (S1)	No display or decode, but the box must listen for network messages or capture EPG data. Programs can be recorded. Momentary disruption of data processing is allowed in order to achieve minimum power.	Minimal power for the functionality
Passive Standby (S2)	No display, decode, DVR, front end, or transport. No data processing. The system configures the supported wake-up devices that exist in AON and ON-OFF block.	Minimal power
Deep Sleep Standby (S3)	No display, decode, DVR, front end, or transport. The entire chip is power-gated except for the AON block. The system configures the supported wake-up devices that exist only in AON block.	Lowest power level

For the remainder of this document, these states will be referred to in *italics* to reduce ambiguity (for example, the *On* state or the *Passive Standby* state). Please be aware that general terms like “standby” or “power management” can mean different things, depending on the context.

Four power states, S0, S1, S2 and S3, have been introduced in Broadcom’s 40 nm chips. The mapping from each of these states to Nexus power states is shown in Table 1.

It is possible to transition between any of the states. A Nexus platform can be initialized into any of the states during the execution of the NEXUS_Platform_Init() function.

Chip-specific documentation and customer product specifications may use different names for these states. A mapping should be possible.

Broadcom's 40 nm chips (BCM7231, BCM7425, etc.) have a Power Management State Machine in the AON (Always ON) block that monitors all wake-up devices (like IR, CEC, Timer, Keypad, and GPIO). Not all wake-up devices are part of the AON block. For instance, WOL, UHF, and XPT are not part of the AON block and hence these wake-up devices are not supported in *Deep Sleep* mode. They can only be used in *Passive Standby* mode. 65 nm chips do not have an AON block and *Deep Sleep* mode is not supported at all in 65 nm SoCs. The wake-up devices in the ON-OFF block differ by chip. Please refer to chip-specific documentation for supported wake-up devices. For more information, refer to "Setting Up Wake-Up Devices" on page 8.

The Active Standby State (S1)

To enter the *Active Standby* state, the application must follow these general steps:

1. Stop playback and live decode.
2. Shut down all display output and remove all window inputs.
3. Call the function NEXUS_Platform_SetStandbySettings(), using Active Standby settings.
4. The application uses kernel interface to power-down kernel-controlled peripherals. This includes the following devices:
   a. Power-down SATA, USB, Ethernet, and MoCA.
   b. Set the CPU clock divisor.
   c. Set the DDR self-refresh rate.
   d. Perform a MEMC1 power-down (only available on certain platforms).

The CPU is not in power-down mode, but runs in a low-power mode. Appropriate hardware blocks are kept on to support any functionality that may be required in *Active Standby*. This may include listening to network data, capturing EPG data, recording from a network or front-end, etc.

The code fragment below summarizes all the above steps and shows how an application can put Nexus in *Active Standby*. (Linux® kernel standby code is covered in "pmlib" on page 14.)

```
NEXUS_PlatformStandbySettings nexusStandbySettings;
/* Stop decoders */
NEXUS_VideoDecoder_Stop(videoDecoder);
NEXUS_AudioDecoder_Stop(audioDecoder);
/*Stop playback */
NEXUS_Playback_Stop(playback);
/* Close File. Required for umount */
NEXUS_FilePlay_Close(file);
NEXUS_VideoWindow_RemoveInput(window, input);
NEXUS_VideoInput_Shutdown(input);
NEXUS_VideoWindow_Close(window);
NEXUS_Display_Close(display);
NEXUS_Platform_GetStandbySettings(&nexusStandbySettings);
nexusStandbySettings.mode = NEXUS_PlatformStandbyMode_eActive;
NEXUS_Platform_SetStandbySettings(&nexusStandbySettings);
```

Note: See BSEAV/app/standby/active_standby.c and BSEAV/app/standby/standby.c for *Active Standby* state example code.

For more information about Nexus Standby APIs, refer to the Nexus platform header file /nexus/platform/$(NEXUS_PLATFORM)/include/nexus_platform_standby.h.

The Passive Standby State (S2)

The *Passive Standby* state is entered by calling the Nexus_Platform_SetStandbySettings() function. This will clock-gate most of the cores except the CPU and set up the wake-up devices in the AON and ON-OFF islands. It also powers down the SRAMs in certain A/V blocks. The CPU is put in standby with the help of pmlib or the Linux commands. The general steps for entering the *Passive Standby* state are:

1. Stop all playback/record/live decode. Remove all display output and window inputs and close all displays.
2. Call NEXUS_Platform_SetStandbySettings(), using `NEXUS_PlatformStandbyMode_ePassive` mode and correct the wake-up settings (see "Setting Up Wake-Up Devices" on page 8).
3. Call pmlib to power down linux controlled peripherals and put the CPU in SUSPEND mode.

Linux Power Management is done by means of a library called pmlib and the Linux Power Management interface. The pmlib library and its usage is explained in "Linux Power Management" on page 12.

Setting Up Wake-Up Devices

Wake-up devices are programmed using the Nexus API. The NEXUS_PlatformStandbySettings structures allows for wake-up devices to be programmed and passed to NEXUS_Platform_SetStandbySettings. This only applies to the *Passive Standby* and *Deep Sleep Standby* states. Wake-up need not be programmed in *Active Standby*, because the Linux kernel is never put into standby and the software monitors the wake-up events.

Any event from any one of the programmed wake up devices will bring the CPU out of standby. Examples of wake up devices are IR input, UHF input, Keypad, GPIO, HDMI CEC, Timer, etc. Wake-up devices may vary by chip, so refer to chip-specific documentation for availability of wake-up devices.

IR and UHF inputs can also be programmed to wake-up the CPU on specific key presses. These need to be programmed using the Nexus functions for those modules. Refer to Nexus API documentation in the software release for more information about programming these devices.

The `NEXUS_PlatformStandbyStatus` data structure provides wake-up status. The function NEXUS_Platform_GetStandbyStatus() will provide information about which wake-up device was used to bring the CPU out of standby.

Note: The above-mentioned wake-up scheme is not applicable to all chips. Some chips like the BCM7400, BCM7405, etc. cannot be woken up using this method. For these, software monitoring of wake-up devices is required and the CPU can only be put in low-power mode. Refer to chip-specific documentation to determine whether hardware-assisted wake-up is possible for a given chip.

Note: Not all Linux kernel versions support waking up the CPU by programming the wake-up devices. Kernel version 2.6.31 or higher is required to support this wake-up mechanism.

The code below summarizes all the steps given above and demonstrates how an application can use Nexus functions to enter *Passive Standby*, including wake-up device programming. Linux kernel standby code is covered in"Linux Power Management" on page 12.

```
NEXUS_PlatformStandbySettings nexusStandbySettings;
 /* Stop decoders */
NEXUS_VideoDecoder_Stop(videoDecoder);
NEXUS_AudioDecoder_Stop(audioDecoder);
/*Stop playback */
NEXUS_Playback_Stop(playback);
/* Close File. Required for umount */
NEXUS_FilePlay_Close(file);
NEXUS_Platform_GetStandbySettings(&nexusStandbySettings);
nexusStandbySettings.mode = NEXUS_PlatformStandbyMode_ePassive;
nexusStandbySettings.wakeupSettings.ir=true; /* Wake-up from IR*/
nexusStandbySettings.wakeupSettings.uhf=true; /* Wake-up from UHF */
nexusStandbySettings.wakeupSettings.timeout=10; /* Timeout in seconds */
NEXUS_Platform_SetStandbySettings(&nexusStandbySettings);
```

The following code shows how to enable wake-up from transport packet

```
Nexus_TransportWakeup_Filter Filter[16] =
{
    { 0x47, 0xFF, 1 }, { 0x12, 0xFF, 1 }, { 0x34, 0xFF, 1 }, { 0x15, 0xFF, 1 },
    { 0x12, 0xFF, 2 }, { 0x34, 0xFF, 2 }, { 0x03, 0xFF, 2 }, { 0x46, 0xFF, 2 },
    { 0x66, 0xFF, 2 }, { 0x4F, 0xFF, 3 }, { 0x31, 0xFF, 3 }, { 0x00, 0xFF, 3 },
    { 0x88, 0xFF, 2 }, { 0x77, 0xFF, 2 }, { 0x66, 0xFF, 2 }, { 0x55, 0xFF, 2 },
};

NEXUS_PlatformStandbySettings nexusStandbySettings;
NEXUS_TransportWakeup_Settings xptWakeupSettings;

 NEXUS_Platform_GetStreamerInputBand(0, &inputBand);
NEXUS_TransportWakeup_GetSettings(&xptWakeupSettings);
BKNI_Memcpy(xptWakeupSettings.filter[0].packet, Filter, sizeof(Filter));
xptWakeupSettings.inputBand = inputBand;
 xptWakeupSettings.packetLength = sizeof(Filter);
 xptWakeupSettings.wakeupCallback.callback = transportWakeupCallback;
 xptWakeupSettings.wakeupCallback.context = NULL;
 xptWakeupSettings.enabled = true;
 rc = NEXUS_TransportWakeup_SetSettings(&xptWakeupSettings);

NEXUS_Platform_GetStandbySettings(&nexusStandbySettings);
 nexusStandbySettings.mode = NEXUS_PlatformStandbyMode_ePassive;
    nexusStandbySettings.wakeupSettings.transport = true;
 rc = NEXUS_Platform_SetStandbySettings(&nexusStandbySettings);
```

Note: See BSEAV/app/standby/standby.c for a *Passive Standby* example. Refer to nexus_transport_wakeup.h for more description about the Nexus_TransportWakeup_Filter structure.

The Deep Sleep Standby State

Deep Sleep Standby state is entered by calling the Nexus_Platform_SetStandbySettings() function. This will clock-gate all the AV cores. It will also power gate the SRAMs and save the register content. Nexus will not power gate the chip. The chip is power gated by kernel. The general steps for entering the *Deep Sleep Standby* state are:

1. Stop all playback/record/live decode. Remove all display output, window inputs and close all displays.
2. Call NEXUS_Platform_SetStandbySettings(), using `NEXUS_PlatformStandbyMode_eDeepSleep` mode and correct the wake-up settings (see "Setting Up Wake-Up Devices" on page 8).
3. Power gate the chip by writing to the kernel sysfs interface (see "sysfs Attributes" on page 18).

Wake up devices are programmed in the same way as described in "The Passive Standby State (S2)" on page 8. Refer to "Setting Up Wake-Up Devices" on page 8 for more information.

The code below shows how an application can use Nexus functions to put the system into the *Deep Sleep Standby* state, including wake-up device programming. Linux Kernel standby code is covered in "Linux Power Management" on page 12.

```
NEXUS_PlatformStandbySettings nexusStandbySettings;
 /* Stop decoders */
NEXUS_VideoDecoder_Stop(videoDecoder);
NEXUS_AudioDecoder_Stop(audioDecoder);
/*Stop playback */
NEXUS_Playback_Stop(playback);
/* Close File. Required for umount */
NEXUS_FilePlay_Close(file);
NEXUS_Platform_GetStandbySettings(&nexusStandbySettings);
nexusStandbySettings.mode = NEXUS_PlatformStandbyMode_eDeepSleep;
nexusStandbySettings.wakeupSettings.ir=true;
nexusStandbySettings.wakeupSettings.uhf=true;
nexusStandbySettings.wakeupSettings.timeout=10; /* Timeout in seconds */
NEXUS_Platform_SetStandbySettings(&nexusStandbySettings);
```

Notes:

- See BSEAV/app/standby/active_standby.c for an *Active Standby* state example.
- *Deep Sleep Standby* state standby and wake-up is only supported in Linux version 2.6.37-2.2 or higher.
- The above discussion about *Deep Sleep Standby* state standby is limited to warm boot only. *Deep Standby* state cold boot is also supported but is not likely to be used in production systems due to the long wake up time.
- Only wake-up devices that are part of the AON block can be used to bring the system out of *Deep Sleep Standby*. Wake-up devices residing in the ON-OFF block are not supported in this state.

Dynamic Power Management

Dynamic power management (dynamic PM) is not a power management state like *Active Standby*, *Passive Standby,* or *Deep Sleep Standby*. Instead, dynamic PM is an internal feature of the Nexus/Magnum software stack in which software will automatically turn off power to hardware cores that it "knows" are unused. This technique is used in both the *On* and *Active Standby* states. When the user stops, disables, or disconnects components with software calls, Nexus and Magnum will check whether it can turn off power. This process is transparent to the user.

There are no explicit dynamic PM functions. The user's responsibility is to consistently stop, disable, or disconnect components that are no longer in use. For instance, if you are not decoding from a transport input band, there's no requirement that it be disabled. If you don't disable it, however, there's no way for the underlying software to know that it can be powered down. This means that if the software consistently stops, disables, and/or disconnects components when they are no longer in use, dynamic PM will minimize the power consumption of the system.

Nexus Functions That Enable Dynamic PM

Table 2 provides list of Nexus functions that can result in dynamic power reductions.

Table 2: Nexus Dynamic Power Management Functions

Call	*Behavior*	*Procedure/Comments*
Frontend	When no tuner is active, power will be reduced.	Call NEXUS_Frontend_Untune() to make a tuner inactive.
HdmiInput	When all HdmiInput's are disconnected from any video window, power will be reduced.	NEXUS_VideoWindow_RemoveInput() will power down HDMI Rx.
VideoDecoder	When all decodes are stopped and disconnected from any video window, power will be reduced.	Stop decode, then call: • NEXUS_VideoWindow_RemoveInput() • NEXUS_VideoInput_Shutdown()
Transport	When all inputs are disabled, power will be automatically reduced.	• Stop all playbacks using NEXUS_Playpump_Stop() or NEXUS_Playback_Stop() and NEXUS_PidChannel_Close() • Disable all input bands by setting `NEXUS_InputBandSettings.enabled = false`.
Display	When all outputs are disconnected, power will be reduced.	Call NEXUS_Display_RemoveOutput() for each output handle.
Security	When all key slots have been closed, power will be reduced.	Call NEXUS_Security_FreeKeySlot() for every keyslot that has been created.
DMA	When all DMA jobs have been terminated, power will be reduced.	Call NEXUS_DmaJob_Destroy() for every job that was created.
Graphics2D	No dynamic power management at this time.	–

Table 2: Nexus Dynamic Power Management Functions (Cont.)

Call	*Behavior*	*Procedure/Comments*
Graphics3D	No dynamic power management at this time.	–
Audio	No dynamic power management at this time.	–

Note: See nexus/examples/power_down.c for a dynamic power management example.

Linux Power Management

Not all hardware blocks are controlled by Nexus and Magnum. For instance, many of the peripherals are controlled by the Linux® kernel. Linux handles all power management for these peripherals. This includes but is not limited to:

- Power up/down for SATA, Ethernet, C, and USB
- CPU clock scaling and DDR self-refresh rate.
- Suspending the CPU and resuming from the wake-up device and/or timer.
- Powering off or suspending secondary memory controller.
- Hotplugging a secondary thread processor.

Kernel Power State Changes

Linux kernel recognizes the following power states:

- S0: Full power; It maps into the *On* (S0) *or Active Standby* (S1) states defined earlier in this document.
- S1: Standby; maps to the *Passive Standby* (S2) state.
- S3: Suspend to RAM (STR); maps to *Deep Sleep Standby* (S3) with warm boot.
- Halt: Maps to power off or S3 with cold boot.

Full Power

In this state you may have all or some peripherals fully or partially powered so that they could operate properly. The CPU is powered but may stay in idle state if no processing is needed. Memory is active, but can be configured to go to self-refresh mode if no memory access occurs for a predetermined period of time. The system is responsive to user input and any other external events and processes are scheduled normally. The user may selectively disable some nonessential components if needed.

Standby

In this state, the CPU is halted and memory is in self-refresh. All user and kernel processes are frozen; all device drivers have been notified about incoming suspension and should have put their respective hardware block to the inactive state. No data transfer between devices and memory or CPU is allowed or expected.

When suspending the device drivers, the kernel PM code traverses the device tree in the order opposite to which devices were registered during boot or later when loadable modules are initialized. This guarantees that each driver is able to communicate to its hardware during the suspend phase. The device driver should be careful not to power down or gate resources that its hardware may share with other blocks—this is the easiest way to crash the system during suspend/resume.

Only the boot CPU is running at the last phases of suspend; all other TPs are stopped.

To support resume from standby state, the kernel locks the interrupt handler and a small piece of code responsible for recovering the memory controller state into the I-cache. Only certain interrupts from predefined wake-up sources are enabled.

Upon wake-up, the code starts execution from I-cache. After the DDR is reconfigured and becomes accessible, the kernel code unlocks itself from the cache and continues running from memory. It now reverses all the actions taken by the suspend sequence by notifying each driver of the resume. Device drivers are resumed in the same order they were registered during boot or module loading, which is in the order opposite of being suspended. Kernel and usermode threads are “thawed” at the late stages of resume when all the device drivers must be fully operational.

Suspend-to-RAM

The Linux executes the same sequence as in *Standby* with the following exceptions:

- Some device drivers may need to save its hardware state in memory if the hardware is on ON-OFF block. This state will be reloaded during resume from STR.
- The interrupt handler and early resume code is not locked in the I-cache, because the CPU core (including caches) are in the ON-OFF block.
- the entire D- and I-cache are flushed to memory.
- The data necessary to resume is saved in the AON system RAM.
- The kernel (optionally) performs memory authentication operations.

Halt (S3 with Cold Boot)

The system executes the regular shutdown sequence, that is closes all files, synchronizes file system with storage, unloads device drivers. All interrupts except for wake-up interrupt are disabled.

pmlib

The pmlib library is a C-language library wrapper for Linux Power Management functions. It provides a simple function that can be called by application to put the CPU in standby or low power mode. The pmlib functions are located at BSEAV/lib/power_standby. The following code snippet shows how pmlib functions can be used to put the CPU In a lower-power mode and/or SUSPEND mode

```
void *brcm_pm_ctx;
struct brcm_pm_state pmlib_state;

brcm_pm_ctx = brcm_pm_init();

brcm_pm_get_status(brcm_pm_ctx, &pmlib_state);

/*false : power down USB, true : power up USB */
pmlib_state.usb_status =false;
/*false : power down SATA, true : power up SATA */
pmlib_state.sata_status = false;
/*false : power down MEMC1, true : power up MEMC1 */
 pmlib_state.memc1_status = false;
/*false : power down TP1, true : power up TP1 */
pmlib_state.tp1_status = false;
pmlib_state.cpu_divisor = cpu_divisor;  /*Scale the CPU clock/*/
pmlib_state.ddr_timeout = ddr_timeout; /* Set DDR self refresh timeout */

rc = brcm_pm_set_status(brcm_pm_ctx, &pmlib_state);

/*SUSPEND CPU ("Passive Standby")*/
brcm_pm_suspend(brcm_pm_ctx, BRCM_PM_STANDBY);
```

Note: Before putting the CPU in standby or powering down the USB/SATA/ENET interfaces, it is recommended that nonroot file systems (such as SATA, USB, NFS, etc.) be unmounted. If your application runs over NFS (i.e., Ethernet) or if you have a SATA/USB HDD-based file system, you may need to keep power to those devices in Passive Standby.

The Linux rootfs comes with a pmtest application that performs individual pmlib operations.

SATA

When pmlib is used to control SATA, it is recommended that unmount all hard drives prior to powering off the controller be unmounted. Besides clock/power gating the block, pmlib forcefully removes all detected SATA devices from the system device tree. That also removes their respective devnodes, and may unload the drivers (if they are loadable). After powering off, device insertion/removal will no longer be detected by the driver.

If an application uses pmlib to power SATA back on, the library will initiate rescan and reattach all the devices found during the scan. When power-on is complete, detection mechanism is restored.

Examples using pmtest:

```
# pmtest sata 0    # power off
# pmtest sata 1    # power on
```

Alternatively, the hdparm utility provides a standard mechanism to spin down a hard drive and put it into standby mode. Standby mode is enabled by code similar to this:

```
hdparm –y /dev/<devname>
```

For example:

```
hdparm –y /dev/sda
```

To disable standby mode, set the timeout value to 0

```
hdparm –S0 /dev/sda
```

Notes:

1. Spinning down a hard drive does not involve AHCI controller's clock and power gating, which means that power savings will be lower than in a complete power-off.
2. For hdparm to work properly, AHCI must be powered. That is, the following sequence is correct:

   ```
   hdparm –y /dev/sda
   pmtest sata 0
   ```

   Reversing the order of operations is not acceptable.

USB

USB power management is based on the Linux run-time power management mechanism, which in the case of USB is called autosuspend. The run-time PM automatically suspends (clock and power gate) any USB device when it is not being used for a predetermined period of time. Default timeout for USB devices is two seconds, but can be changed on “per device” basis.

In order for USB device to be autosuspended, it must closed by all drivers or applications.

By default autosuspend is supported in the kernel, but USB controllers start up with this feature being disabled. To enable autosuspend, user can run the script pmusb.

pmusb without arguments shows run-time PM status of all USB controllers in the system.

To enable autosuspend on an individual controller, run this:

```
# pmusb –m auto <devname>
```

For example:

```
# pmusb –m auto usb1
```

To enable autosuspend on all controllers, run

```
# pmusb –m auto
```

To enable autosuspend on all USB devices, run with ‘recursive’ option:

```
# pmusb –m auto –r
```

To disable autosuspend, use `–m on` option:

```
# pmusb –m on –r
# pmusb –m on usb1
# pmusb –m on
```

Network Interfaces

MoCA and Ethernet power are not controlled through pmlib. To power and clock gate a network adapter block, application needs to close the interface.

Ethernet

To power off an Ethernet block, close its interface by running this:

```
# ifdown eth0
```

To power up an Ethernet block, open its interface by running this:

```
# ifup eth0
```

MOCA

To power off MoCA block, stop the MoCA daemon and close its interface; for example:

```
# mocactl stop
# ifdown eth1
```

To power up the MoCA block, start the MoCA daemon or open its interface; for example:

```
# mocactl start
# ifup eth1
```

mocactl also powers up and down the external MoCA BCM3450 LNA.

Wake-Up Events

S2 and S3 support different sets of wake-up events. The full list of wake-up events is chip-dependent. Only those events that are controlled through Linux kernel are described in this section.

Wake-up Timer

If S2 suspend is initiated with `standby_flags bit0` set, WKTMR is set to wake up the system after one second. While there is no kernel API to set an arbitrary alarm interval, software can do it by directly manipulating mapped I/O registers, as in the following procedure:

1. Enable the timer interrupt in the PM/AON L2 interrupt control.
2. Enable the wake-up timer in the WKTMR_EVENT register.
3. Set timeout in seconds in WKTMR_ALARM register.

   Do NOT set bit 0 of `standby_flags`.

Ethernet Wake-on-LAN

Wake-on-LAN allows remote wake up the system by sending a pre-formatted Ethernet packet to the network node. There are different types of wake-on-LAN packets, STB kernel supports Magic Packet and ARP packet (ACPI pattern).

Note: When performing S2 suspend with WOL enabled, some portions of network block may have to be kept active which increases power consumption.

To enable Ethernet Magic packet and ACPI WOL on interface ethX, issue the following command:

```
# ethtool -s ethX wol g
```

To enable Ethernet ACPI WOL only on interface ethX, issue the following command:

```
# ethtool -s ethX wol a
```

To disable Ethernet Magic packet and ACPI WOL on interface ethX, use the following command:

```
# ethtool -s ethX wol d
```

MoCA Wake-on-LAN

To enable MoCA Magic packet and ACPI WOL on interface ethX, issue these commands:

```
# mocactl wol --enable
# ethtool -s ethX wol g
```

To enable MoCA ACPI WOL only on interface ethX, issue these commands:

```
# mocactl wol --enable
# ethtool -s ethX wol a
```

To disable Ethernet Magic packet and ACPI WOL on interface ethX, use one of the following command:

```
# mocactl wol --disable
# ethtool -s ethX wol d
```

sysfs Attributes

Most power management operations can be performed by modifying sysfs entries

Linux Kernel Attributes

To initiate S3 standby with warm boot, run the following:

```
# echo mem > /sys/power/state
```

To initiate S2 standby, run the following:

```
# echo standby > /sys/power/state
```

To initiate S3 standby with cold boot, run the following:

```
# echo 1 > /sys/devices/platform/brcmstb/halt_mode
# halt
```

To offline TP1, run the following:

```
# echo 0 > /sys/devices/system/cpu/cpu1/online
```

To online tp1, run the following:

```
# echo 1 > /sys/devices/system/cpu/cpu1/online
```

Broadcom-Specific Attributes

All Broadcom specific power management sysfs attributes are located in the /sys/devices/platform/brcmstb folder.

memc1_power

Controls MEMC1/DDR1 state

- 0 - powered down
- 1 - fully powered
- 2 - in SSPD

```
# echo 0 > /sys/devices/platform/brcmstb/memc1_power
# echo 1 > /sys/devices/platform/brcmstb/memc1_power
# echo 2 > /sys/devices/platform/brcmstb/memc1_power
```

standby_flags

A hex value defining how system suspend is implemented. The default value is 0x0. See Table 3 for individual bit descriptions.

Table 3: System Suspend Standby Flags

Bit #	*Bit Mask*	*Description*
0	0x01	Wake the system in one second.
1	0x02	Sleep for 180 seconds immediately prior to system suspend — useful for BBS verification of register settings. The system is not fully suspended at that time, however.
2	0x04	Print progress characters during low-level suspend/resume; useful for debugging
3	0x08	Do not go to S2 suspend — wait for five seconds and resume. If Bit 1 is also set, timeout is 12 seconds.

```
# echo 0x5 > /sys/devices/platform/brcmstb/standby_flags
```

ddr_timeout

Controls MEMC0 self-refresh. 0 - self-refresh disabled, 1 - self-refresh enabled. Default value is 0.

```
# echo 1 > /sys/devices/platform/brcmstb/ddr_timeout
```

time_at_wakeup

A read-only file that stores WKTMR, reading at the time of wake-up in the form of:

```
hex(COUNTER):hex(PRESCALER_VAL)

# cat /sys/devices/platform/brcmstb/time_at_wakeup
0:0
```

Connecting everything®

BROADCOM

BROADCOM CORPORATION
5300 California Avenue
Irvine, CA 92617

Phone: 949-926-5000
Fax: 949-926-5203
E-mail: info@broadcom.com
Web: www.broadcom.com

STB_RefSw-SWUM401-R November 23, 2011

3/14/2012 88CFX