Settop Box Reference Software
Power Management Overview

Broadcom
Corporate Headquarters: San Jose, CA
Web: www.broadcom.com

Revision History

Revision	Date	Change Description
0.7	07/29/16	Added: NxClient Power Management.
0.6	05/14/15	Added: Thermal Management.
0.5	05/28/14	Updated: • Nexus Dynamic Power Management Functions Table. • Linux Power States. • Sysfs Attributes.
0.4	02/25/14	Updated: • Nexus Dynamic Power Management Functions Table. Added: • Availability of Wakeup Devices Table.
0.3	10/30/12	Updated: • The procedure for Active, Passive and Deep Sleep standby states.
0.2	11/23/11	Added: • NEXUS_Platform_Standby interface and use of pmlib for CPU standby. • The Deep Sleep Standby State. • Nexus Dynamic Power Management Functions. • Kernel Power State Changes. • SATA, USB, Network Interfaces, Wakeup Events, sysfs Attributes.
0.1	06/13/11	Initial Release.

Table of Contents

Introduction 1
Broadcom Settop Box Power States 1
Nexus Power Management 2
Nexus API and Power States 2
Active Standby (S1) 3
Passive Standby (S2) 5
Deep Sleep Standby (S3) 7
Nexus Wakeup Devices 9
Dynamic Power Management 10
NxClient Power Management 12
Linux Power Management 14
Linux Power States 14
Full Power 14
Standby 14
Suspend to Ram (STR) 15
Power Off (S3 cold boot) 15
Peripheral Power Management 15
SATA 15
Network Interfaces 16
Linux Wake-up Devices 17
Ethernet Wake-on-Lan 17
MoCA Wake-on-Lan 17
PMlib 18
Sysfs Attributes 18
Linux Kernel Attributes 18
Broadcom Specific Attributes 19
Linux Test Utilities 20
pml 20
pmtest 20
Thermal Management 21

Introduction

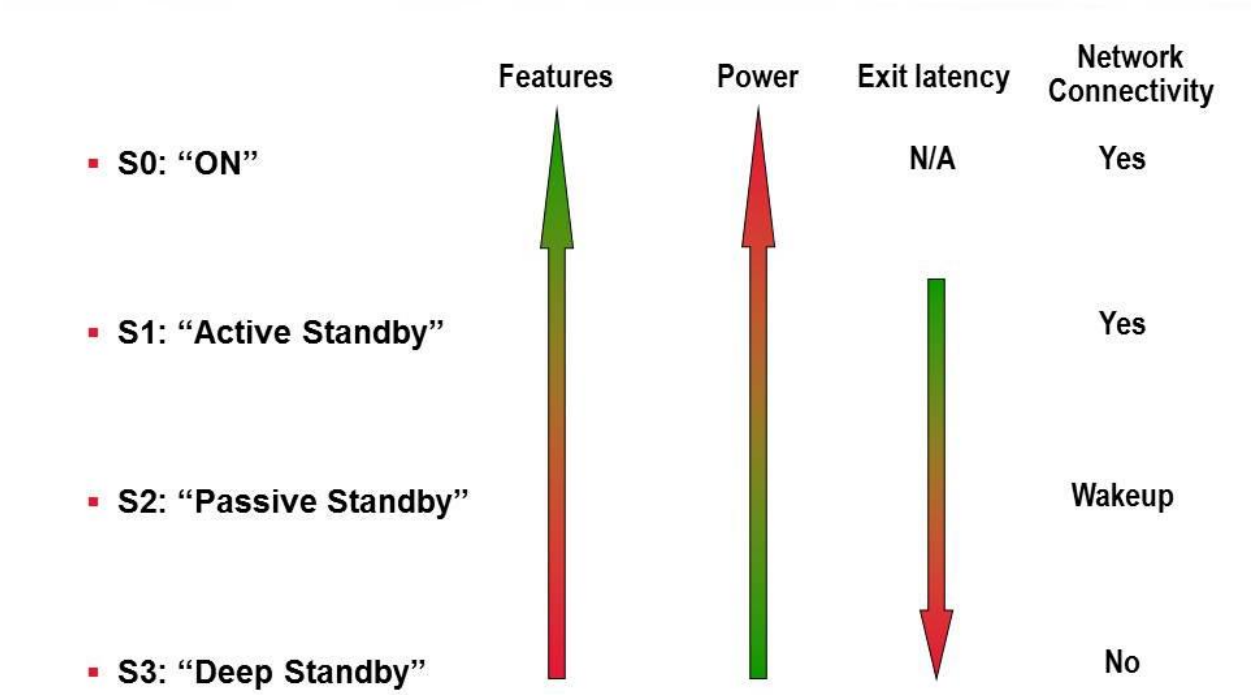
This document describes the design and implementation of power management support in the Broadcom set-top box reference software stack.

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 such information.

Broadcom Settop Box Power States

Figure 1: STB Power States



Nexus Power Management

Nexus API and Power States

Nexus Power Management API is defined in
/nexus/platforms/common/include/nexus_platform_standby.h

Nexus defines 4 power states as shown in the Table 1

Table 1: The Four Nexus Power States

Nexus Power State	Functionality	Power Level
On (S0)	• Full functionality. • Dynamic Power Management turns off unused cores.	Max power
Active Standby (S1)	• No outputs, display, decode, encode. • Box could listen for network messages or EPG data. • Programs can be recorded	Minimal power for the functionality
Passive Standby (S2)	• No outputs, display, decode, encode, PVR, frontend. • The system only configures a set of wake-up devices.	Minimal power
Deep Sleep Standby (S3)	• No outputs, display, decode, DVR, front end. • The system configures wake-up devices that exist in AON block.	Lowest power level

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

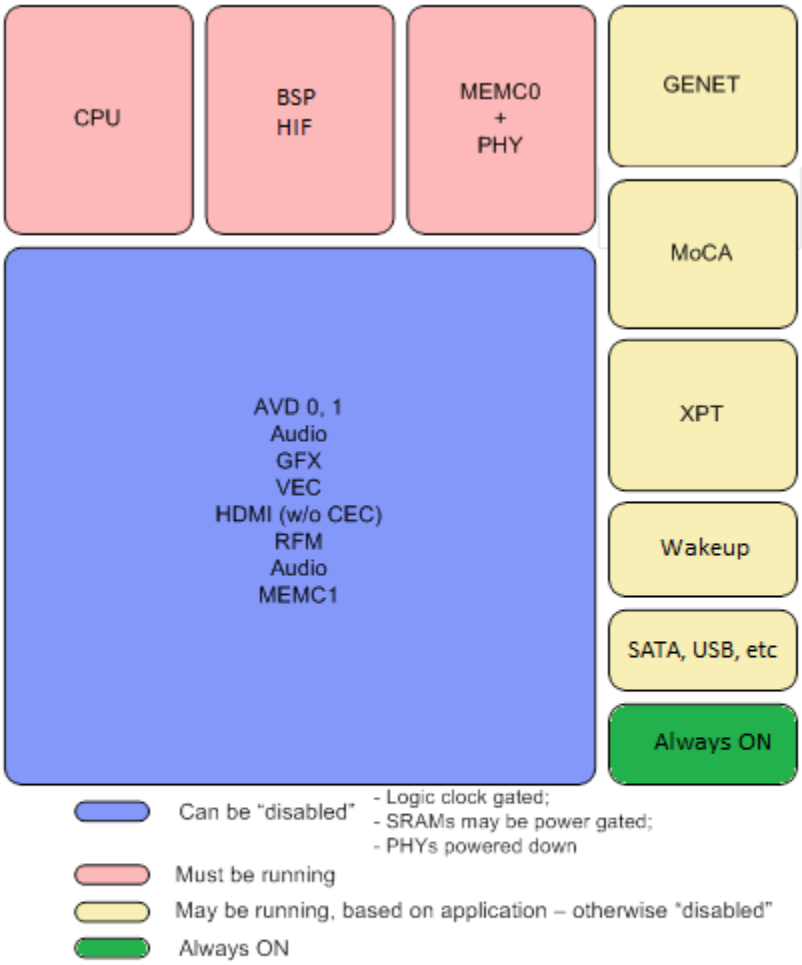
Broadcom Settop Box supports 4 power states S0, S1, S2 and S3. The mapping from each of these states to nexus power states is shown in the Table 1.

It is possible to transition between any of states. Nexus platform can be initialized into any of the states during NEXUS_Platform_Init.

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

Active Standby (S1)

Figure 2: Cores disabled in Active Standby



CPU is not powered down or clock gated in this mode. CPU could run at slower frequency or non-boot CPUs could be powered down. Appropriate hardware blocks are kept on to support any functionality that may be required in "*Active Standby*". This may include listening to network data, capture EPG data, record from network or frontend, etc. To enter "Active Standby", application needs to follow these steps:

1. Stop playback, decode, encode and any other operations which are not supported in this state.

2. Call *NEXUS_Platform_SetStandbySettings*, and set the mode to *NEXUS_PlatformStandbyMode_eActive*.
3. Application uses kernel API to power down kernel controlled peripherals. This includes the following:
    - Power down SATA, Ethernet and MoCA.
    - Set DDR self-refresh timeout.
    - CPU frequency scaling and hot-plugging.
    - MEMC1 power down (only available on certain platforms)

The sample code below shows how an application can put Nexus in "*Active Standby*" state.

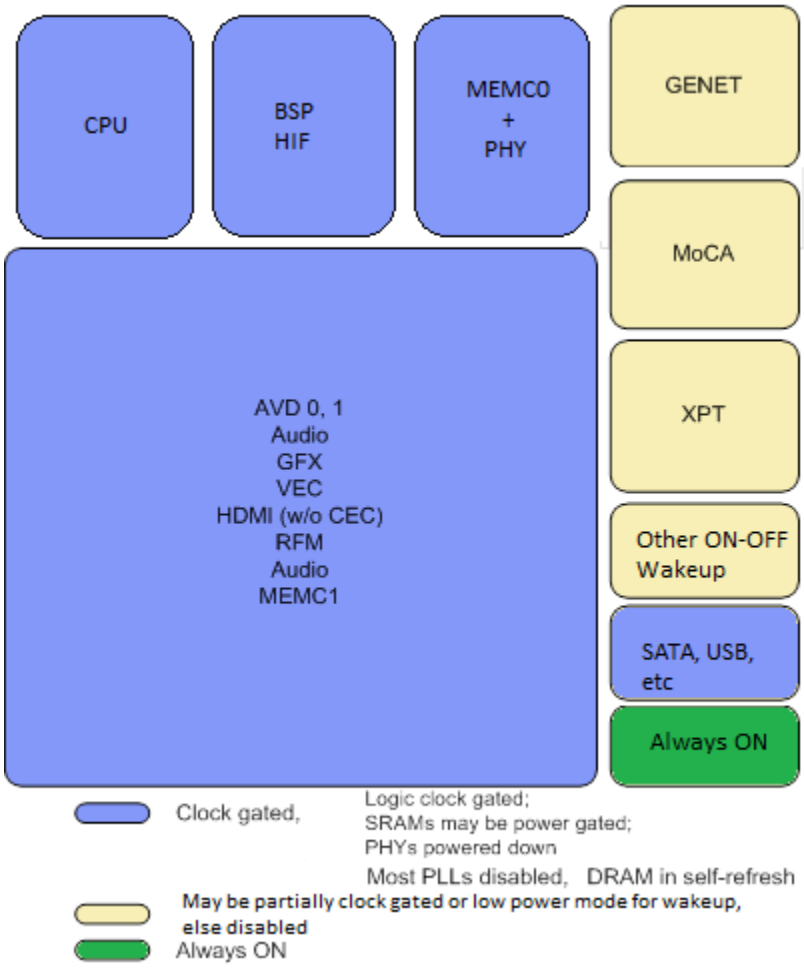
```
NEXUS_PlatformStandbySettings nexusStandbySettings;

/* 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_eActive;
NEXUS_Platform_SetStandbySettings(&nexusStandbySettings);
```

Notes:

1. See '*BSEAV/app/standby/standby.c*' for "*Active Standby*" example.
2. For more information about Nexus Standby APIs refer to the nexus platform header file nexus/platforms/common/include/nexus_platform_standby.h

Passive Standby (S2)

Figure 3: Cores disabled in Passive Standby



Nexus will clock gate most of the nexus controlled cores. Nexus also sets up the wakeup devices and SRAMS may be powered down. System enters standby using pmlib or sysfs interface. The general steps for entering "*Passive Standby*" are:

1. Application needs to stop playback, record, decode, encode operations and make sure system is idle.
2. Call *NEXUS_Platform_SetStandbySettings*, and set standby mode to *NEXUS_PlatformStandbyMode_ePassive* mode with the desired wake up.
3. Call pmlib or write to sysfs to put system in S2 mode.

The sample code below shows how an application can use Nexus APIs to enter "*Passive Standby*" and set up Wake-up devices. Linux Kernel standby code is covered in section Linux Power Management.

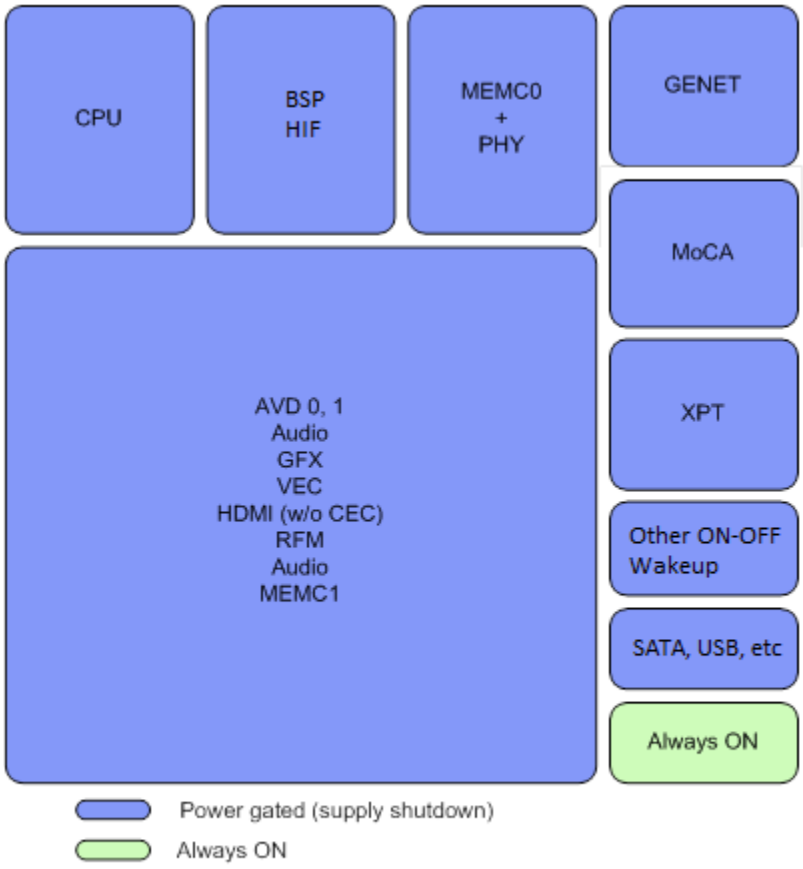
```
NEXUS_PlatformStandbySettings nexusStandbySettings;

/* 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; /* IR Wakeup */
nexusStandbySettings.wakeupSettings.cec = true; /* CEC Wakeup */
nexusStandbySettings.wakeupSettings.timeout = 10; /* Timer Wakeup */
nexusStandbySettings.wakeupSettings.transport = true; /* XPT Wakeup */

NEXUS_Platform_SetStandbySettings(&nexusStandbySettings);
```

Notes:

1. See '*BSEAV/app/standby/standby.c*' for a "Passive Standby" example.
2. Refer to nexus_transport_wakeup.h for more description about Nexus_TransportWakeup_Filter structure.
3. Linux controlled peripherals are powered down using pmlib or sysfs interface. pmlib and its usage is explained in the section Linux Power Management.

Deep Sleep Standby (S3)

Figure 4: Cores disabled in Deep Sleep Standby



Nexus will clock gate all the nexus controlled 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 after application calls Pmlib api or writes to sysfs to put system in S3. The general steps for entering "*Deep Sleep Standby*" are:

1. Application needs to stop playback, record, decode, encode operations and make sure system is idle.
2. Call *NEXUS_Platform_SetStandbySettings*, and set standby mode to *NEXUS_PlatformStandbyMode_eDeepSleep* mode with the desired wake up.
3. Call pmlib or write to sysfs to put system in S3 mode.

The sample code below shows how an application can use Nexus APIs to put the system in "Deep Sleep Standby" including wake-up device programming. Linux Kernel standby code is covered in section Linux Power Management.

```
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.cec = true;
nexusStandbySettings.wakeupSettings.timeout = 10;
NEXUS_Platform_SetStandbySettings(&nexusStandbySettings);
```

Notes:

1. See '*BSEAV/app/standby/standby.c*' for a "Deep Sleep Standby" example.
2. S3 standby and wake up is only supported in linux version 2.6.37-2.2 or higher.
3. S3 standby is supported on 40nm and 28nm platform only.

Nexus Wakeup Devices

The 40nm and 28nm platforms have a Power Management State Machine (PMSM) on the Always On (AON) power island which monitors for interrupts from wake up devices in S2 and S3 states. AON power island stays powered up during S3 mode. Wake-up devices which are part of AON Power Island can be used to wake-up from "Deep Sleep" mode. Wake-up devices which are not part of AON Power Island can only be used to wake up from "Passive Standby". This includes Transport, Ethernet and MoCA wake up. Please refer to chip specific documentation for supported wake up devices.

Wake-up devices, except for Ethernet, MoCA Wake On Lan and RF4CE are programmed using the Nexus API. Application enabled the wake-up devices using the *NEXUS_PlatformStandbySettings* structure. This only applies to "*Passive Standby*" and "*Deep Sleep Standby*" state. Application does not need to program wake-up devices in "*Active Standby*". CPU is powered On in "*Active Standby*" and the application software continues to monitor for wake-up event. Nexus callbacks are used to notify the application of any wake-up events.

Refer to Linux Power Management section for more information on Ethernet and MoCA Wake On Lan

IR and UHF inputs can be programmed to wake up the chip on specific key presses. These need to be programmed using the respective module's Nexus API. Refer to Nexus API Documentation for more information about programming the filters for those devices.

Transport wakeup packet is set using the nexus transport module API. Refer to Nexus API Documentation and nexus_transport_wakeup.h for more information. The sample code below shows how to enable wakeup 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_GetStandbyStatus structure will provide information about which wake up device was used to bring system out of Standby.

Table 2: Availability of Wakeup Devices in Standby Modes

Wake-up Device	S1	S2	S3
Ir Input	✓	✓	✓
Hdmi Cec	✓	✓	✓
Timer	✓	✓	✓
Keypad	✓	✓	✓
AON Gpio	✓	✓	✓
RF4CE	✓	✓	✓
Uhf Input	✓	✓	✗
Transport	✓	✓	✗
Ethernet WOL	✓	✓	✗
MoCA WOL	✓	✓	✗

Dynamic Power Management

Dynamic power management is not a power management state. Dynamic PM is an internal feature of the Nexus/Magnum software stack where software will automatically turn off power to hardware cores that it knows are unused. It is the application's responsibility to stop, disables or disconnects components that are no longer in use. Nexus and Magnum will internally power down unused components. There are no explicit Dynamic PM API's. For instance, disconnecting the outputs from a display will power down the unused output and reduce power.

Table 3 provides a list of Nexus functions that can result in dynamic power reductions

Table 3: Nexus Dynamic Power Management Functions

Module	Behavior	Procedure/Comments
Audio Dacs	Audio Dacs will be powered down when disconnected from decoder	• Call `NEXUS_AudioOutput_RemoveInput` for audio dacs
Dma	When all DMA jobs have been destroyed, power will be reduced.	• Call `NEXUS_DmaJob_Destroy` for every job that was created.
Frontend	Tuners are in low power mode when not in use	• Call `NEXUS_Frontend_Untune` to power down tuners.
Graphics2D	Graphics 2D enters low power state when hardware idle	
Graphics3D	Graphics 3D enters low power state when hardware idle	
Hdmi Input	Hdmi Input is powered down when it is disconnected from video window	• `NEXUS_VideoWindow_RemoveInput` will power down Hdmi Input
Hdmi Output	Hdmi Output is powered down when it is disconnected from display	• Call `NEXUS_Display_RemoveOutput` to power down Hdmi Phy.
Transport	Unused sub modules are clock gated.	• Close all Remux Channels using `NEXUS_Remux_Close`. • Close all Dma Channels using `NEXUS_Dma_Close`.
Video Dacs	Analog outputs are powered down when they are disconnected from display	• Call `NEXUS_Display_RemoveOutput` to power down Analog Video outputs.
Video Decoder	Decoders are clock gated when all decodes are stopped and disconnected from the video window.	• Stop decode then call `NEXUS_VideoWindow_RemoveInput`.

Notes:

1. Some modules have automatic clock gating in hardware which results in power savings when the core is idle.
2. Hdmi Output and Video Dacs are powered down when cable is physically disconnected.

NxClient Power Management

Following APIs are provided for standby:

*NEXUS_Error NxClient_GetStandbyStatus(NxClient_StandbyStatus *pStatus);*

unsigned NxClient_RegisterAcknowledgeStandby(void);

void NxClient_UnregisterAcknowledgeStandby(unsigned id);

void NxClient_AcknowledgeStandby(unsigned id);

*void NxClient_GetDefaultStandbySettings(NxClient_StandbySettings *pSettings);*

*NEXUS_Error NxClient_SetStandbySettings(const NxClient_StandbySettings *pSettings);*

Nxserver handles the standby requests from clients and puts nexus is standby mode. The steps are as follows

- Standby client application calls the *NxClient_SetStandbySettings* API to request for standby.
- When the server receives a request for standby, it updates the standby status and notifies clients by setting the *NxClient_StandbyStatus.transition* to *NxClient_StandbyTransition_eAckNeeded*. Server then waits for all registered clients to acknowledge the changes in state.
- Client processes need to poll for change in standby status using the *NxClient_GetStandbyStatus* API.
- When the client sees the change in state to *NxClient_StandbyTransition_eAckNeeded*, it needs to stop all its activities and prepare for standby. This may include stopping encode, decode, playback, graphics operations, etc. Once the client has stopped all its activities it needs to acknowledge the change in state using the *NxClient_AcknowledgeStandby*.
- After all clients have acknowledged, nxserver will put nexus to standby. The server waits for all acknowledges with a default timeout of 10s. If all clients do not acknowledge within the timeout, server will go ahead and attempt to put nexus in standby. The timeout is configurable during nxserver init.

- Once nexus is in standby, nxserver will update the *standbyStatus.transition* flag to *NxClient_StandbyTransition_eDone*. At that point standby client application can call the linux APIs to put system into standby.

Registering and Unregistering Clients

Clients can use the *NxClient_RegisterAcknowledgeStandby* and *NxClient_UnregisterAcknowledgeStandby* APIs to register/unregister with the server. By default, all clients are implicitly registered to acknowledge for standby during Join and are not required to register explicitly. A client may also choose to register multiple times. Each time it registers, an id is assigned which is returned by *NxClient_RegisterAcknowledgeStandby*. The id that is implicitly assigned during Join is unknown to the application and can be obtained when *NxClient_RegisterAcknowledgeStandby* is called the first time. Any subsequent calls will generate a new id. If a client does not wish to acknowledge, it can simply unregister itself using *NxClient_UnregisterAcknowledgeStandby.* Both *NxClient_AcknowledgeStandby* and *NxClient_UnregisterAcknowledgeStandby* require an id as an input. This could be either the implicit or the explicit id. If a client has not made any calls to *NxClient_RegisterAcknowledgeStandby ,* then *NxClient_AcknowledgeStandby* will use the implicit id by default and ignore the id that was passed to it. To unregister, the client has to make at least one call to *NxClient_RegisterAcknowledgeStandby* in order to obtain the id to unregister. Each client must acknowledge standby as many times as it has registered (including the implicit register).

Notes:

1. Refer to '*nexus/nxclient/apps/standby.c* ' for standby client code.
2. Refer to '*nexus/nxclient/apps/play.c* ' for client acknolwdgement.
3. Refer to 'nexus/nxclient/include/nxclient_standby.h" for standby APIs.

Linux Power Management

Linux handles the Power Management functions for all the cores that are controlled by Linux. Linux Power Management functionality includes:

- Power up/down peripherals : SATA, Ethernet, MoCA, etc.
- CPU frequency scaling.
- CPU hotplug.
- Enabling DDR self refresh.
- Enabling Standby.
- Wake on Lan.

Linux Power States

Linux kernel recognizes the following power states:

- Full Power: Used in S0 or S1
- Standby: Maps to Passive Standby (S2) state.
- Suspend to RAM (STR): Maps to Deep Sleep Standby (S3) with warm boot.
- Power Off: Maps to 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. 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. System is responsive to user input and any other external events, processes are scheduled normally. User may selectively disable some non-essential components if needed.

Standby

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

When suspending the device drivers, 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 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 boot CPU is running at the last phases of suspend, all other TPs are stopped.

To support resume from standby state, kernel locks interrupt handler and a small piece of code responsible for recovering memory controller state into boot Sram. Only certain interrupts from pre-defined wakeup sources are enabled.

Upon wakeup the code starts execution from boot Sram. After DDR is reconfigured and becomes accessible, kernel code unlocks itself from the Sram and continues running from memory. It now reverses all the actions taken by suspend sequence by notifying each driver of resume. Device drivers are resumed in the same order as they were registered during boot or module loading, which is in the order opposite to suspend. Kernel and user mode threads are "thawed" at the late stages of resume when all the device drivers must be fully operational.

Suspend to Ram (STR)

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.
- Interrupt handler and early resume code is not locked in boot Sram, because CPU core is in ON-OFF block.
- Entire d- and i-cache are flushed to memory.
- Data necessary to resume is saved in AON system RAM.
- Linux performs memory authentication operations.

Power Off (S3 cold boot)

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

Peripheral Power Management

SATA

When `pmlib` is used to control SATA, please note that it is recommended to unmount all harddrives prior to powering off the controller. 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 re-attach 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, `hdparm` utility provides standard mechanism to spin down a harddrive and put it into standby mode. Standby mode is enabled by

```
hdparm -y /dev/<devname>
```

e.g.

```
hdparm -y /dev/sda
```

To disable standby mode, set 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, therefore 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.

Network Interfaces

Ethernet

To power off an Ethernet block, close its interface

```
ifdown eth0
```

To power up an Ethernet block, open its interface

```
ifup eth0
```

MoCA

To power off MoCA block, stop MoCA daemon and close its interface, e.g.

```
mocactl stop      → MoCA 1.1
mocap set -stop   → MoCA 2.0
ifdown eth1
```

To power up MoCA block, start MoCA daemon or open its interface, e.g.

```
mocactl start     → MoCA 1.1
mocap set -start  → MoCA 2.0
ifup eth1
```

Linux Wake-up Devices

S2 and S3 support different sets of wake-up events. Full list of wakeup events is chip dependent. Here are described only those events that are controlled through Linux kernel.

Ethernet Wake-on-Lan

Wake-on-LAN allows remotely wakeup the system by sending a pre-formatted Ethernet packet to the network node. Linux supports Magic Packet and ARP packet wakeup. ARP wakeup is not supported in Linux 3.14 and higher.

Notes

1. 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

MoCA 1.1

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

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

To enable MoCA ACPI WOL only on interface ethX, issue the 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
```

MoCA 2.0

```
mocap set --wom_mode 1
mocap set --wom_magic_mac val `hw_address`
mocap set --wom_magic_enable 1
mocap set --wom_pattern mask 0 0xff 15
```

PMlib

Pmlib is a 'C' library wrapper for Linux Power Management functions. It provides a simple API that can be called by application to put CPU in standby or low power mode. The following code shows Pmlib API's usage

```
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);

pmlib_state.sata_status = 0;    /* 1=on, 0=off */
pmlib_state.tp1_status  = 0;    /* 1=on, 0=off */
pmlib_state.tp2_status  = 0;    /* 1=on, 0=off */
pmlib_state.tp3_status  = 0;    /* 1=on, 0=off */
/* scaling setting, in kHz */
pmlib_state. cpufreq_setspeed = cpu_speed;
/* 0=no PM, >0 = timeout for self-refresh */
pmlib_state. srpd_status = timeout;

rc = brcm_pm_set_status(brcm_pm_ctx, &pmlib_state);

/* Standby (S2) */
brcm_pm_suspend(brcm_pm_ctx, BRCM_PM_STANDBY);


/* Suspend (S3) */
brcm_pm_suspend(brcm_pm_ctx, BRCM_PM_SUSPEND);
```

Notes

1. pmtest application can be used to perform individual pmlib operations.
2. It is recommended to un-mount non-root file systems (e.g. SATA, USB, NFS, etc) before powering down those interfaces or entering standby.

Sysfs Attributes

Most power management operations can be performed by writing to sysfs

Linux Kernel Attributes

Initiate Suspend to Ram (S3 warm boot)

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

Initiate Standby (S2)

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

Turning ON/OFF CPU n

```
echo 1 > /sys/devices/system/cpu/cpu<n>/online
echo 0 > /sys/devices/system/cpu/cpu<n>/online
```

Scaling CPU frequency

Available on certain platforms which support frequency scaling. Certain platforms may only support a limited range of frequencies. Requires Linux 3.14 and above.

Change to the directory with all of the cpufreq sysfs parameters

```
cd /sys/devices/system/cpu/cpu0/cpufreq
```

This sets the CPUFREQ governor to userspace control

```
echo userspace > scaling_governor
```

This requests a CPU frequency change to the lowest supported

```
cat cpuinfo_min_freq > scaling_setspeed
```

This shows you what the current clockspeed in kHz is (it should be equal to min)

```
cat scaling_setspeed
```

This requests a CPU frequency change to the highest supported

```
cat cpuinfo_max_freq > scaling_setspeed
```

Broadcom Specific Attributes

Memc1_power

Memc1 power can be controlled on certain 40nm platforms that have more than 1 memory controller

Powered down

```
echo 0 > /sys/devices/platform/brcmstb/memc1_power
```

Fully powered

```
echo 1 > /sys/devices/platform/brcmstb/memc1_power
```

SSPD

```
echo 2 > /sys/devices/platform/brcmstb/memc1_power
```

DDR Self-Refresh

40nm platform using linux 3.3

```
echo `timeout` > /sys/devices/platform/brcmstb/ddr_timeout
```

28nm platform using linux 3.14

```
echo `timeout` > /sys/bus/platform/drivers/brcmstb_memc/*/srpd
```

Linux Test Utilities

Broadcom Linux release provides some utilities to test Linux power Management. These are useful for testing standby modes and linux peripheral power management.

pml

pml is a power management script that can be used for testing various standby modes in linux without having to run the entire nexus/magnum software stack. It is useful for debugging standby issue in Linux, BOLT or at the board level. Some of the useful commands are

```
pml      # S2 standby
pml -d   # S3 standby (S3 warm boot)
pml -p   # S5 standby (S3 cold boot)
```

For a complete list of options and usage run 'pml –h' at the linux prompt.

pmtest

pmtest is another useful command line utility to test linux peripheral power management for devices like SATA, CPU frequency scaling, DDR self-refresh, etc. Some examples are

```
pmtest sata 0   # Power down SATA controller
pmtest tp1 0    # Power down TP1 (second CPU thread)
pmtest srpd 64  # Enable self-refresh on all MEMCs after 64 cycles
```

For a complete list of all options and usage run 'pmtest –h'.

Thermal Management

28nm Platforms support a thermal sensor driver for the AVS TMON temperature monitoring hardware. The AVS TMON core provides temperature readings, a pair of configurable high- and low-temperature threshold interrupts, and an emergency over-temperature chip reset. The Linux thermal driver utilizes the first two to provide temperature readings and high-temperature notifications to applications. The over-temperature reset is not exposed to applications; this reset threshold is critical to the system and should be set with care within the bootloader.

Linux provides the ability to setup an arbitrary number or temperature thresholds (trip points). These trip points also have an option hysteresis threshold. Trip point notifications are triggered when the temperature increases above the temperature threshold and reduces below the hysteresis threshold. By default the notifications are rate-limited to every 10 seconds. The trip point parameters are encoded in the device tree and can be configured from BOLT.

Linux supports a cooling device based on the Intel Power Clamp driver. This driver allows a user to enforce a particular percentage of time that a CPU must be idle. For instance, setting the cooling device to a setting of '40' will mean that the system will be at least 40% idle. Currently this driver is limited to a maximum of 50% idle time.

The thermal subsystem provides both a sysfs API and a signaling-based uevent interface. The sysfs API is simpler and can be utilized synchronously in polling loops, while the uevent interface is useful for receiving asynchronous event notifications. Both can be used in conjunction. Some examples of using the sysfs API as follows:

Check the thermal zone type:

```
# cat /sys/class/thermal/thermal_zone0/type
avs_tmon
```

Check the current temperature (millidegrees celsius):

```
# cat /sys/class/thermal/thermal_zone0/temp
46738
```

Check the cooling device type:

```
# cat /sys/class/thermal/cooling_device0/type
intel_powerclamp
```

Check the maximum state (i.e., maximum throttling allowed):

```
# cat /sys/class/thermal/cooling_device0/max_state
50
```

Check the current state (clamping is disabled):

```
# cat /sys/class/thermal/cooling_device0/cur_state
-1
```

Enable maximum clamping:

```
# echo 20 > /sys/class/thermal/cooling_device0/cur_state
[248354.452000] intel_powerclamp: Start idle injection to reduce power
```

Check current state again; notice that the system is idle:

```
# cat /sys/class/thermal/cooling_device0/cur_state
99
```

Nexus utilizes the uevent interface to receive thermal notifications from Linux and throttle the Graphics 2D and Graphics 3D operating frequency.

For a more complete and detailed description of Thermal Management, please refer to Linux Release Notes.