# Settop Box Reference Software Power Management Overview

**Broadcom**
Corporate Headquarters: San Jose, CA

Web: www.broadcom.com

## Revision History

| Revision | Date | Change Description |
|---|---|---|
| 0.8 | 1/29/2018 | Updated:<br>• Nexus dynamic power management function<br>• Description of Linux standby mode<br>• Linux peripheral power management |
| 0.7 | 07/29/16 | Added:<br>NxClient Power Management. |
| 0.6 | 05/14/15 | Added:<br>Thermal Management. |
| 0.5 | 05/28/14 | Updated:<br>• Nexus Dynamic Power Management Functions Table.<br>• Linux Power States.<br>• Sysfs Attributes. |
| 0.4 | 02/25/14 | Updated:<br>• Nexus Dynamic Power Management Functions Table.<br>Added:<br>• Availability of Wakeup Devices Table. |
| 0.3 | 10/30/12 | Updated:<br>• The procedure for Active, Passive and Deep Sleep standby states. |
| 0.2 | 11/23/11 | Added:<br>• NEXUS_Platform_Standby interface and use of pmlib for CPU standby.<br>• The Deep Sleep Standby State.<br>• Nexus Dynamic Power Management Functions.<br>• Kernel Power State Changes.<br>• SATA, USB, Network Interfaces, Wakeup Events, sysfs Attributes. |
| 0.1 | 06/13/11 | Initial Release. |

## Table of Contents

## Introduction

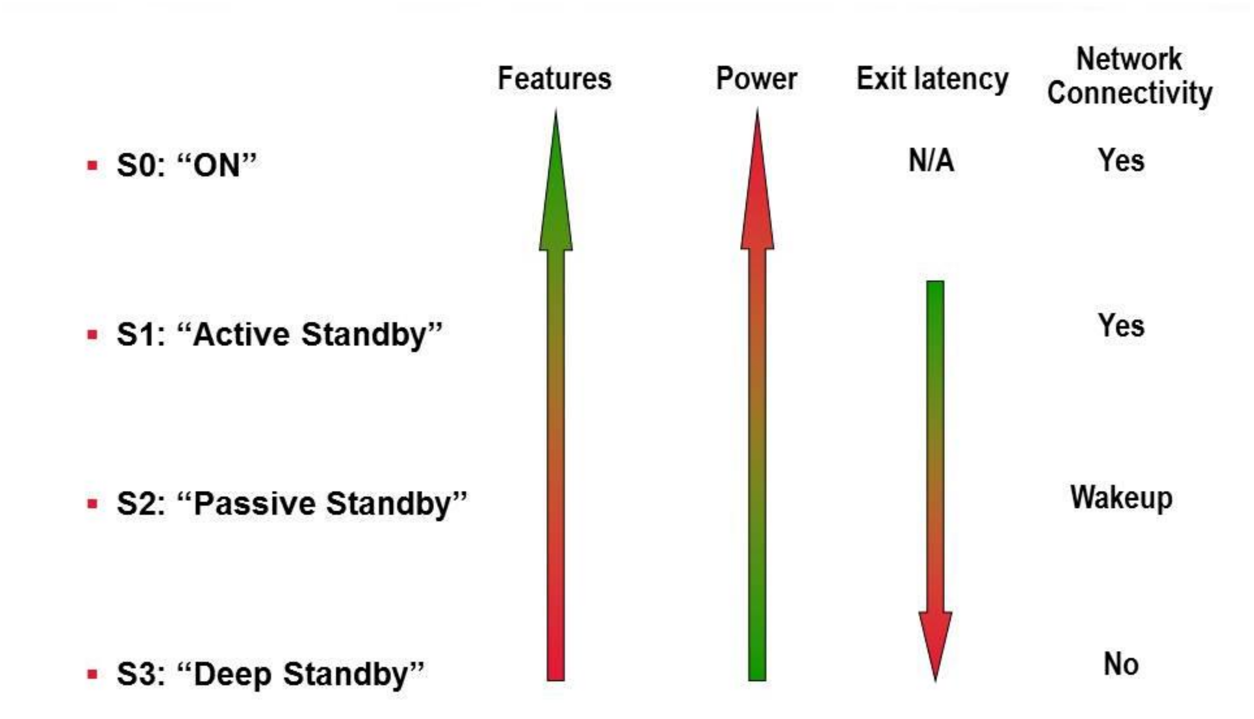
This document describes the design and implementation of power management support in the Broadcom set-top box reference software stack.

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<br><br>(S0) | • Full functionality.<br>• Dynamic Power Management turns off unused cores. | Max power |
| Active Standby<br><br>(S1) | • No outputs, display, decode, encode.<br>• Box could listen for network messages or EPG data.<br>• Programs can be recorded | Minimal power for the functionality |
| Passive Standby<br><br>(S2) | • No outputs, display, decode, encode, PVR, frontend.<br>• The system only configures a set of wake-up devices. | Minimal power |
| Deep Sleep Standby<br><br>(S3) | • No outputs, display, decode, DVR, front end.<br>• 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.

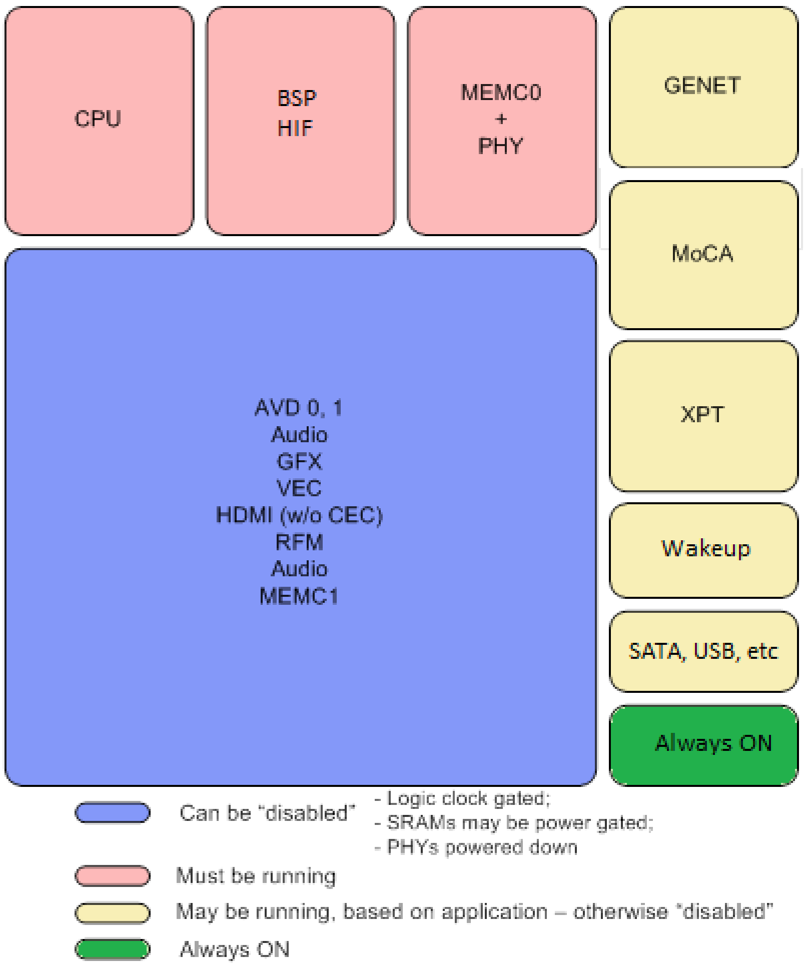
Broadcom Settop Box supports 4 power states S0, S1, S2 and S3. The mapping from each of these states to nexus power states is shown in the Table 1.

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)

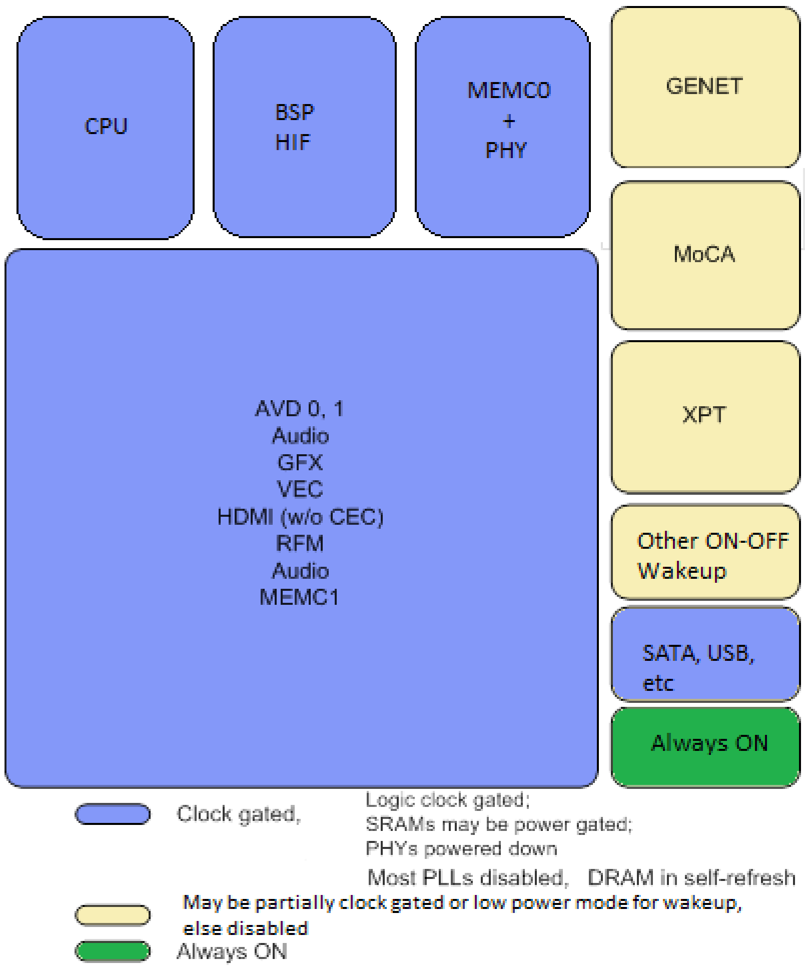
The sample code below shows how an application can put Nexus in "*Active Standby*" state.

```
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.

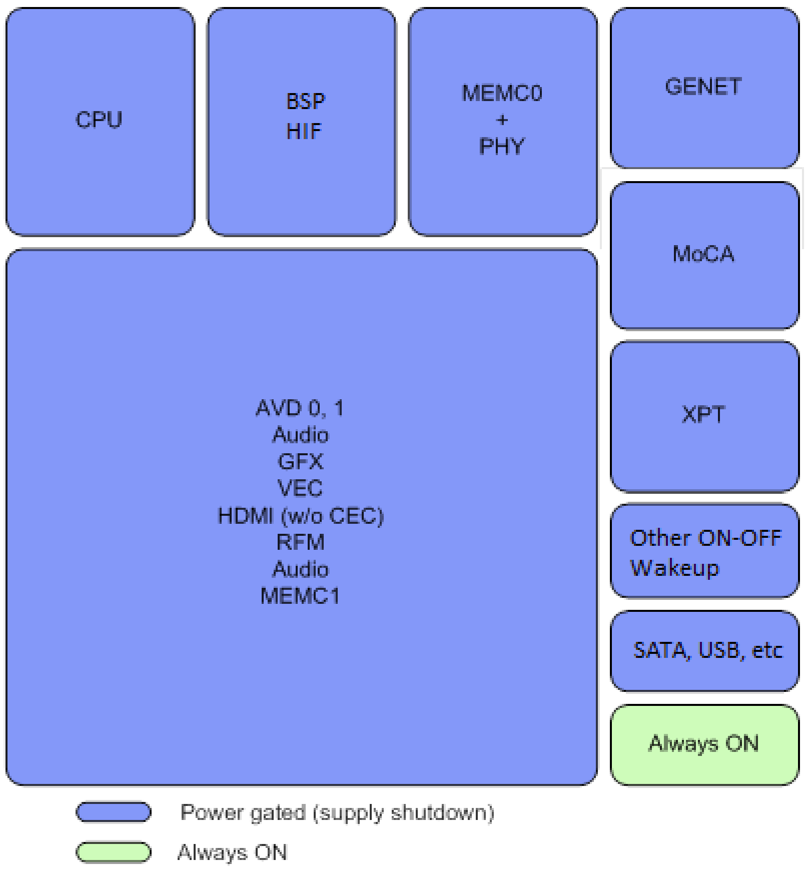
```
NEXUS_PlatformStandbySettings nexusStandbySettings;

/* 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.

Below is a list of Nexus modules that support dynamic Power Management along with the API calls that application needs to make

### Audio Dacs

Audio Dacs are powered down when the Dac output is disconnected from decoder. Application needs to call `NEXUS_AudioOutput_RemoveInput.`

### Audio Decoder

Certain platforms support DSP frequency scaling when audio decode is stopped. Application needs to call `NEXUS_AudioDecoder_Stop.`

### Frontend

Unused tuners are powered down when application calls `NEXUS_Frontend_Untune`.

### Graphics2D

M2MC interblit clock gating is enabled by default which clock gates the M2MC core when it is idle. Graphics output can be completely disabled using

```
NEXUS_SurfaceCompositorDisplaySettings.enabled = false;
```

This will disable the GFD output. The same can also be done using NxClient API

```
NxClient_GraphicsSettings.enabled = false;
```

### Graphics3D

V3D/VC5 core is automatically clock gated between frames when the core is idle. The core may also be power gated on platforms that support it.

### Hdmi Input

Hdmi Input is clocks gated when it disconnected from a video window. Application needs to call `NEXUS_VideoWindow_RemoveInput`.

### Hdmi Output

Hdmi Phy is powered down when the Hdmi cable is disconnected. Hdmi Output is clock gated and Phy is powered down when the output is disconnected from the display. Application needs to call `NEXUS_Display_RemoveOutput`.

### Picture Decoder

Decoder is clock gated when application calls `NEXUS_PictureDecoder_Stop`.

### Transport

Submodule clocks gating is enabled in Transport. Clocks are turned On when a submodule is opened and turned off when submodule is closed.

### Video Dacs

Video Dacs cable detect is enabled by default which will put the Dacs in low power mode when the cable is disconnected. The Dacs are also completely powered down when the Dac output is disconnected from a Display. Application calls `NEXUS_Display_RemoveOutput`.

### Video Decoder

Decoder is clock gated when video decoder is stopped and decode channel is closed. Application needs to call `NEXUS_VideoWindow_RemoveInput`.

### Video Encoder

Certain platforms support VICE frequency scaling. Frequency is scaled down when application calls `NEXUS_VideoEncoderStop`.

## Disabling NEXUS Power Management

Power Management support is enabled by default in Nexus. Nexus Power Management can be disabled using the compile time option

```
NEXUS_POWER_MANAGEMENT=n
```

Nexus will not have any Dynamic Power Management or Standby support when Nexus Power Management Support is disabled.

## 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 Power Management functionality includes:

- Support for standby modes.
- CPU Power Management : CPU Frequency Scaling, CPU Hotplug.
- Peripherals Power Management: SATA, Ethernet, MoCA, PCIe, DDR.
- Wake on Lan.

### Linux Power States

Linux kernel supports the following power states:

- Full Power (S0/S1) :
- Standby (S2).
- Suspend to RAM (S3 warm boot).
- Power Off (S5 or S3 cold boot).

#### Full Power

All peripherals are active and powered on. Unused peripherals may be powered down to save power. For example one or more CPUs may be powered down or frequency could be scaled down. Memory is active, but can be configured to go to self-refresh mode if it is inactive for certain duration.

#### Standby (S2)

All peripherals are turned off. Peripherals used for wakeup maybe in a low power state. In this state CPU is halted and DDR is in self-refresh. The high level suspend and resume sequence is as follows.

##### *Suspend*

All user and kernel processes are frozen and applications are no longer scheduled. Device drivers put their peripherals in suspend mode. All hardware activity is stopped and any wake-on logic, if requested, is enabled. All secondary CPUs are powered off and Linux executes from boot CPU. A small piece of code is copied into the internal SRAM. This code is called and starts executing from SRAM. This piece of code is responsible for finally putting the system in standby mode.

The standby sequence is initialized by writing to sysfs as follows

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

##### *Resume*

Upon wakeup the CPU starts executing from internal Sram. It polls the DDR Phy register to make sure DDR is available for use. CPU completes the power up handshake with the Power Management State Machine (PMSM) and resumes normal execution from memory. At this point secondary CPUs can be brought back online. Device drivers resume their peripherals and disable any wake-on logic. Finally all processes and unfrozen and applications resume their execution.

## Suspend to Ram (S3 warm boot)

All peripherals in On/Off domain and powered down. DRAM is in self-refresh. CPU is powered down. The high level suspend and resume sequence is as follows.

### *Suspend*

All user and kernel processes are frozen and applications are no longer scheduled. Device drivers put their peripherals in suspend mode. All hardware activity is stopped and any wake-on logic, if requested, is enabled. All secondary CPUs are powered off and Linux executes from boot CPU. Linux initiates a handshake with the PMSM/BSP and waits for an acknowledgement. It then performs a memory hash using the transport DMA engine. A parameters structure stores the state of system prior to entering S3, including the physical address of the resume function. The address of this structure is stored in the AON registers. The DDRY Phy is told to power gate its pads and finally the PMSM is programmed to enter S3. At this point the system is in S3 and DDR is is self-refresh mode.

Suspend to Ram is initialized by writing to sysfs as follows

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

### *Resume*

Execution begins with BOLT after resuming from S3. CPU boots into the FSBL which performs basic initialization and turns on the I-cache. FSBL reloads the chip configuration and restarts the DDR Phy. It then obtains the S3 parameters from AON registers and initializes the transport DMA engine for hash verification. After DRAM hash has been verified, it jumps to the re-entry address provided by Linux and starts executing the Linux resume function. After basic initialization and set up, Linux resumes execution in normal environment. The rest of the resume procedure is similar to the S2 resume sequence.

## Power Off (S5 or S3 cold boot)

This mode is similar to S3, except that DRAM is not kept in self-refresh and resume requires a reboot.

### *Suspend*

The system executes the regular shutdown sequence. Linux closes all files, synchronizes file system with storage and unloads device drivers. All interrupts except for wakeup interrupt are disabled.S5 sequence is initiated using the "`poweroff`" command.

### *Resume*

The S5 resume sequence is identical to a power-on reset with the exception that peripherals in the AON domain have been able to retain their state, and can be used to provide the cause of the wake-up for the system.

## CPU Power Management

### CPU Frequency Scaling

CPU frequency scaling  is supported on certain platforms using the cpufreq driver. Requires Linux 3.14 and above. Below is a list of few commands for using cpufreq

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

Sets the governor to userspace

```
echo userspace > scaling_governor
```

Sets the governor to conservative

```
echo conservative > scaling_governor
```

Set the CPU frequency to the lowest supported

```
cat cpuinfo_min_freq > scaling_setspeed
```

Set the CPU frequency to the highest supported

```
cat cpuinfo_max_freq > scaling_setspeed
```

Get the current CPU frequency.

```
cat /sys/devices/system/cpu/cpu0/cpufreq/scaling_setspeed
```

### CPU Hotplug

CPU hotplug can be used to turn off ununsed secondary CPU cores.

Power down CPU <n>

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

Power up CPU <n>

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

## 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>
```

To disable standby mode, set timeout value to 0

```
hdparm -S0 /dev/<devname>
```

**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.

SATA Aggressive Link Power Management (ALPM) can be enabled to allow the host controller to enter into low power state when the link is inactive.

Min Power
```
echo min_power
>/sys/class/scsi_host/host0/link_power_management_policy
```

Medium Power
```
echo medium_power >
/sys/class/scsi_host/host0/link_power_management_policy
```

## Ethernet

Ethernet is powered down when its interface is taken down

```
ifdown eth0
```

Bringing the interface back up, powers up the Ethernet

```
ifup eth0
```

ethtool can be used to set the speed of the interface and enable Energy-Efficient Ethernet (EEE)

```
ethtool -s <interface> autoneg off speed <speed> duplex <duplex>
ethtool --set-eee <interface> eee on
```

## 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
```

## PCIE

PCIE Active State Power Management (ASPM) can be enabled in Linux at boot time. This will put the PCIE interface in lower power when the device to which it connects is not in use. Use the following Linux boot parameters to enable ASPM

```
pcie_aspm=force pcie_aspm.policy=powersave
```

The policy can also be set using sysfs as follows

```
echo powersave > /sys/module/pcie_aspm/parameters/policy
```

To see supported policies,

```
cat /sys/module/pcie_aspm/parameters/policy
```

## DDR

### *DDR Self-Refresh*

Linux 3.14 and above

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

Linux 3.3

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

### *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
```

## 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.

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.

## 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'.

### cpufreq utilities

cpufreq-info and cpufreq-set can be used to get and set settings for cpufreq governors and frequency.

To get current cpufreq information

```
cpufreq-info
```

To set performance governor

```
cpufreq-set -g performance
```

## 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.

## NxClient Thermal Management

NxServer provides a thermal monitoring and throttling mechanism which is launched during initialization. The thermal monitor thread polls the temperature provided by the Linux thermal framework. When an over temperature condition is detected, it applies various cooling agents in an attempt to reduce the temperature. Cooling agents could be internal or user defined Internal cooling agents are applied by nxserver without any user interaction. Internal cooling agents include

- CPU P state limitation
- CPU idle injection
- CPU hotplug
- M2MC Frequency Scaling
- V3D Frequency Scaling

Client applications can apply they own user defined cooling agents which include but are not limited to

- Output Resolution change
- Output Frame Rate change
- Input Frame rate and resolution change
- Stop Pip and/or main decode

Client applications can register for a callback which notifies the client app when a specific user defined cooling agent needs to be applied. The order and priority of cooling agents is specified in a thermal configuration file. Undesired cooling agent can also be disabled in the configuration file. A default thermal.cfg file is provided which enables all available cooling agents. Thermal configuration file specifies

- Over temperature threshold in degrees C at which thermal throttling is applied
- Over temperature threshold at which chip resets
- Hysteresis in degree C
- Thermal polling interval in seconds.
- Delay in seconds before cooling agent is applied/removed.
- Priority and order in which cooling agents are to be applied.

For a sample configuration file format refer to nexus/nxclient/ apps/resources/thermal.cfg.