Settop Box Reference Software

Power Management Overview

Broadcom
Corporate Headquarters: San Jose, CA

Broadcom Proprietary and Confidential

Web: www.broadcom.com

Revision History

Revision	Date	Change Description
0.1	10/14/09	Initial draft
0.2	12/14/09	Add NEXUS_Platform_Standby interface and use of pmlib for CPU standby Clarify use of internal or external power controller as part of “Passive Standby” state.
0.3	03/15/11	Updated for new Nexus Standby APIs.
0.4	09/06/11	Add description for 40nm power states
0.5	07/29/16	Add NxClient section

Broadcom Proprietary and Confidential

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 Proprietary and Confidential

Power States

From the Nexus perspective we define four power states:

Power State	Functionality	Power Level
On (S0)	Display, decode and record are all active. Dynamic Power Management is enabled.	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, frontend or transport. No data processing. The system configures the supported wake-up devices which exist in AON and ON-OFF block.	Minimal power
Deep Sleep Standby (S3)	No display, decode, DVR, frontend or transport. Entire chip is power gated except for AON block. The system configures the supported wake-up devices which exist only 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.

Note that in 40nm chips we introduce 4 power states S0, S1, S2 and S3. The mapping from each of these states to nexus power states is shown in the table above

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.

The 40nm chips (7231, 7425, etc) have a Power Management State Machine in the AON (Always ON) block that monitors all wake up devices like IR, CEC, Timer, Keypad, GPIO. Not all wakeup devices are part of AON block. WOL, UHF, XPT are not part of AON block and hence these wakeup devices are not supported in “Deep Sleep” mode. They can only be used for “Passive Standby” mode. 65nm chips do not have an AON block and “Deep Sleep” mode is not supported at all in 65nm. The wakeup devices in ON-OFF block differ per chip. Please refer to chip specific documentation for supported wake up devices. For more information refer to section on Wakeup Devices.

Active Standby State (S1)

To enter “Active Standby”, application needs to follow these general steps:

Stop playback and live decode. Remove all display output, window inputs and close all displays. Shut down display and remove all output and inputs to display and windows.

1. Call NEXUS_Platform_SetStandbySettings, with Active Standby settings.
2. Application uses kernel interface to power down kernel controlled peripherals. This includes the following:
    a. Power down SATA, USB, Ethernet and MoCA.
    b. Set CPU clock divisor
    c. Set DDR self refresh rate
    d. MEMC1 power down (only available on certain platforms)

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, capture EPG data, record from network or frontend, etc.

The code snippet below summarizes all the above steps and shows how an application can put Nexus in “Active Standby”. (Linux Kernel standby code is covered in section PMlib)

```
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” example.

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

Passive Standby State (S2)

“Passive Standby” state is entered by calling the Nexus_Platform_SetStandbySettings. This will clock gate most of the cores except CPU and setup the wakeup devices in AON island and On-OFF island. It also powers down the SRAMs in certain AV blocks. CPU is put in standby with the help of pmlib or the Linux commands. The general steps for entering “Passive Standby” are:

Stop all playback/record/live decode. Remove all display output, window inputs and close all displays.

1. Call NEXUS_Platform_SetStandbySettings, with NEXUS_PlatformStandbyMode_ePassive mode and correct wake up settings (Wake-up Devices).
2. Call pmlib to power down linux controlled peripherals and put CPU in SUSPEND mode.

Note: Linux Power Management is done by means of a library called pmlib and Linux Power Management interface. Pmlib and its usage is explained in the section Linux Power Management.

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 “Passive Standby” and “Deep Sleep Standby” state. Wake up need not be programmed in “Active Standby” because the Linux kernel is never put into Standby. In “Active Standby” mode the software monitors the wake-up event.

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 per chip. Please refer to chip specific documentation for availability of wake devices.

IR and UHF inputs can also be programmed to wake up CPU on specific key presses. These need to be programmed using the Nexus API’s for those modules. Refer to Nexus API Documentation for more information about programming these devices.

Some chips also support wakeup from transport packet, WOL (ENET) and WOL (MOCA). Transport packet wakeup is enabled by the nexus transport module APIs. Refer to Nexus API Documentation and nexus_transport_wakeup.h for more information.

The NEXUS_PlatformStandbyStatus structure provides wake up status. NEXUS_Platform_GetStandbyStatus will provide information about which wake up device was used to bring CPU out of Standby.

Note: The above mentioned wake up scheme is not applicable to all chips. Some chips like 7400, 7405, etc cannot be woken up using this method. For these chips, software monitoring of wake-up devices is required and 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 give chip.

Note: Not all 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 above steps and shows how an application can use Nexus APIs to enter "Passive 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_ePassive;
nexusStandbySettings.wakeupSettings.ir=true; /* Wakeup from IR*/
nexusStandbySettings.wakeupSettings.uhf=true; /* Wakeup from UHF */
nexusStandbySettings.wakeupSettings.timeout=10; /* Timeout in seconds */
NEXUS_Platform_SetStandbySettings(&nexusStandbySettings);
```

The following code 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_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.

Note: Refer to nexus_transport_wakeup.h for more description about Nexus_TransportWakeup_Filter structure.

Deep Sleep Standby State (S3)

“Deep Sleep” state is entered by calling the Nexus_Platform_SetStandbySettings. 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 “Deep Sleep Standby” are:

Stop all playback/record/live decode. Remove all display output, window inputs and close all displays.
Call NEXUS_Platform_SetStandbySettings, with NEXUS_PlatformStandbyMode_eDeepSleep mode and correct wake up settings (Wake-up Devices).
Power gate the chip by writing to the kernel sysfs interface (Sysfs Attributes).

Wake up devices are programmed in the same way as shown in “Passive Standby”. Refer to (Wake-up Devices) for more information.

The code snippet 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.uhf=true;
nexusStandbySettings.wakeupSettings.timeout=10; /* Timeout in seconds */
NEXUS_Platform_SetStandbySettings(&nexusStandbySettings);
```

Note: See ‘*BSEAV/app/standby/standby.c’* for a “Deep Sleep Standby” example.

Note: S3 standby and wake up is only supported in linux version 2.6.37-2.2 or higher.

Note: The above discussion about S3 standby is limited to warm boot only. S3 cold boot is also supported but is not likely to be used in production systems to due long wake up time.

Note: Only wakeup devices that are part of AON block can be used to bring the system out of “Deep Sleep Standby”. Wakeup devices residing in ON-OFF block are not supported in this state.

Dynamic Power Management

Dynamic power management is not a power management state like “Passive Standby” and “Active Standby”. Instead, 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. 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 if it can turn off power. This process is transparent to the user. There are no explicit Dynamic PM API’s. 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. However, if you don’t disable it, there’s no way for the underlying software to know that it can power it down. So, 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 API's which enable dynamic PM

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	–
Graphics3D	No dynamic power management at this time	–
Audio	No dynamic power management at this time	–

NxClient Power Management

An NxClient application can enter standby by calling NxClient_SetStandbySettings. However, this call only initiates a process; it does not take immediate effect. After setting the desired standby state, the app must wait for nxserver to put all clients into standby. Nxserver notifies all other clients that they must prepare for standby by stopping decode, record and transcode. Each client must monitor that standby request, perform the required stops, then call NxClient_AcknowledgeStandby . After all clients have acknowledged, nxserver changes NxClient_StandbyStatus.transition. The application that started the standby process should be polling on NxClient_GetStandbyStatus to learn this status change. If Nexus was able to go into standby, the application can then put Linux into standby using pmlib. See nexus/nxclient/apps/standby.c for an example standby app. See nexus/nxclient/apps/play.c for an example standby monitor thread.

NxClient also supports dynamic power management. The Frontend, HdmiInput, Transport, Security, Dma, and Graphics2D/3D work the same as single-process Nexus. For VideoDecoder, NxClient_Disconnect will result in NEXUS_VideoWnidow_RemoveInput. For Display, an application can all NxClient_SetDisplaySettings to disable various outputs.

Linux Power Management

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

- Power up/down for SATA, Ethernet, MoCA, USB
- CPU clock scaling and DDR self refresh rate.
- Suspending the CPU and resuming from wakeup 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 On (S0) or Active Standby (S1) state defined earlier in this document.

- S1 – standby. Maps to 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. 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 were notified about incoming suspension and should have 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 i-cache. Only certain interrupts from pre-defined wakeup sources are enabled.

Upon wakeup the code starts execution from i-cache. After DDR is reconfigured and becomes accessible, kernel code unlocks itself from the cache and continue 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 they were registered during boot or module loading, which is in the order opposite to suspend. 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.
- interrupt handler and early resume code is not locked in i-cache, because CPU core (including caches) are in ON-OFF block
- entire d- and i-cache are flushed to memory
- data necessary to resume is saved in AON system RAM
- 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 wakeup interrupt are disabled.

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. Pmlib APIs are located at ‘*BSEAV/lib/power_standby*’. The following code snippet shows how Pmlib API’s can be used to put CPU In 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 CPU in standby or powering downs USB/SATA/ENET interface it is recommended to un-mount non-root file systems (e.g. SATA, USB, NFS). If your application running over NFS (i.e. Ethernet) or if you have a SATA/USB HDD-based filesystem, you may need to keep power to those devices in passive standby.

The linux rootfs comes with a pmtest application which performs individual pmlib operations.

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 harddrive 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.

USB

USB power management is based on Linux runtime power management mechanism which in case of USB is called autosuspend. The runtime 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 2 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 a script `pmusb`

`pmusb` without arguments shows runtime pm status of all USB controllers in the system.

To enable autosuspend on individual controller, run

```
# pmusb –m auto <devname>
```

e.g.

```
# 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.

ENET

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
# ifdown eth1
```

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

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

`mocactl` also power up and down the external MOCA BCM3450 LNA.

Wake up events

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.

Wakeup timer

If S2 suspend is initiated with `standby_flags` bit0 set, WKTMR is set to wake up the system after 1 second. While there is no kernel API to set an arbitrary alarm interval, software can do it by directly manipulating mapped I/O registers.

- Enable timer interrupt in PM/AON L2 interrupt control
- Enable wake timer in WKTMR_EVENT register
- Set timeout in seconds in WKTMR_ALARM register
- Do NOT set bit 0 of `standby_flags`

Ethernet Wake-on-Lan

Wake-on-LAN allows remotely wakeup 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 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
```

Sysfs Attributes

Most power management operations can be performed by modifying sysfs entries

Linux kernel attributes

To initiate S3 standby with warm boot:

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

To initiate S2 standby

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

To initiate S3 standby with cold boot

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

To offline TP1

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

To online tp1

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

Broadcom specific attributes

All Broadcom specific power management sysfs attributes are located in `/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

hex value defining how system suspend is implemented. Default value is 0x0. See table below for individual bit description:

Bit #	Bit Mask	Description
0	0x01	Wake the system in 1 second
1	0x02	Sleep for 180 seconds immediately prior to system suspend - useful for BBS verification of register settings. Note: the system is not fully suspended at that time
2	0x04	Print progress characters during low-level suspend/resume. Useful for debugging
3	0x08	Do not go to S2 suspend - wait for 5 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

Read-only file which 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
```