Revision History

The following table shows the revision history for this document.

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Revision</th>
</tr>
</thead>
<tbody>
<tr>
<td>12/15/2016</td>
<td>2.0</td>
<td>Modified Xpm_SystemShutdown in Chapter 2.</td>
</tr>
<tr>
<td>09/21/2016</td>
<td>1.0</td>
<td>Initial Xilinx release.</td>
</tr>
</tbody>
</table>
# Table of Contents

Chapter 1: Introduction to the Power Management Framework
- Introduction ........................................................................................................... 5
- About this Guide .................................................................................................... 5
- Zynq UltraScale+ MPSoC Power Management Overview ........................................ 6
- Zynq UltraScale+ MPSoC Power Management Software Architecture ....................... 8

Chapter 2: Xilpm API Functions
- Introduction ........................................................................................................... 17
- Xilpm API Functions for Suspending Processor Units ................................................. 17
- Xilpm API Functions for Managing PM Slaves .......................................................... 25
- Requesting PM Power Nodes .................................................................................. 26
- Miscellaneous System-level API Functions .............................................................. 29
- Direct Control API Functions .................................................................................. 38

Chapter 3: Using the API for Power Management
- Introduction ........................................................................................................... 40
- Interacting With Other Processing Units .................................................................. 45

Appendix A: Argument Value Definitions
- Introduction ........................................................................................................... 47
- Node IDs: XPMNodeId ............................................................................................... 47
- Acknowledge Request Types: XPMRequestAck ......................................................... 49
- Abort Reasons: XPMAbortReason ............................................................................. 49
- Suspend Reasons ..................................................................................................... 49
- Operating Characteristic Types: XPMOpCharType .................................................... 50
- Notify Event Types: XPMNotifyEvent .................................................................... 50
- Reset Line IDs ......................................................................................................... 50
- Notify Event Types: XPMNotifyEvent .................................................................... 51
- Reset Line IDs ......................................................................................................... 52
- XPM_Notifier struct ................................................................................................. 55
- XPM_NodeStatus struct ........................................................................................... 56
Appendix B: Error Codes

Error Codes ................................................................. 57

Appendix C: Additional Resources and Legal Notices

Xilinx Resources ........................................................... 58
Solution Centers ........................................................... 58
Documentation Navigator and Design Hubs ............................ 58
References .................................................................. 59
Please Read: Important Legal Notices ................................. 59
Chapter 1

Introduction to the Power Management Framework

Introduction

This document explains the power management framework (PMF) used in the Zynq UltraScale+™ MPSoC device.

About this Guide

This chapter provides a brief introduction to the power management framework (PMF), shows the organization of the document, and explains the uses for bare-metal and other forms of Linux, the OS vendors, and which layers are of use with the framework and how to use them.

Table 1-1: Chapters and Appendixes

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Description</th>
<th>Target Audience</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction to the Power Management Framework.</td>
<td>All users.</td>
</tr>
<tr>
<td>2</td>
<td>Chapter 2, Xilpm API Functions.</td>
<td>Bare-metal, RTOS, and other software developers.</td>
</tr>
<tr>
<td>3</td>
<td>Chapter 3, Using the API for Power Management</td>
<td>Bare-metal RTOS and other software developers.</td>
</tr>
<tr>
<td>Appendix A</td>
<td>Appendix A, Argument Value Definitions lists the defined argument values for the power management controller, which is described</td>
<td>All users.</td>
</tr>
<tr>
<td>Appendix B</td>
<td>Appendix B, Error Codes, describes the PMF error codes.</td>
<td>All users.</td>
</tr>
<tr>
<td>Appendix C</td>
<td>Appendix C, Additional Resources and Legal Notices lists the additional references that are cited in this document, and points to other reference sites that are helpful</td>
<td>All users.</td>
</tr>
</tbody>
</table>
**Zynq UltraScale+ MPSoC Power Management Overview**

The Zynq UltraScale+ MPSoC power management framework is a set of power management options, based upon an implementation of the extensible energy management interface (EEMI). The power management framework allows software components running across different processing units (PUs) on a chip or device to issue or respond to requests for power management.

**Zynq UltraScale+ MPSoC Power Management Hardware Architecture**

The Zynq UltraScale+ MPSoC device is divided into four major power domains:

- Full power domain (FPD): Contains the ARM Cortex-A53 application processor unit (APU) as well as a number of peripherals typically used by the APU.
- Low power domain (LPD): Contains the ARM Cortex-R5 real-time processor unit (RPU), the platform management unit (PMU), and the configuration security unit (CSU), as well as the remaining on-chip peripherals.
- Programmable logic (PL) power domain: Contains the PL.
- Battery-power domain: Contains the real-time clock (RTC) as well as battery-backed RAM (BBRAM).

Other power domains listed in the following figure are not actively managed by the power framework.
The following is a diagram of the Zynq UltraScale+ MPSoC device power domains and islands.

![Zynq UltraScale+ MPSoC Power Domain and Islands Diagram](figure1.png)

**Figure 1-1: Zynq UltraScale+ MPSoC Power Domain and Islands**

Because of the heterogeneous multi-core architecture of the Zynq UltraScale+ MPSoC device, no single processor can make autonomous decisions about power states of individual components or subsystems.

Instead, a collaborative approach is taken, where a power management API delegates all power management control to the platform management unit (PMU). It is the key component coordinating the power management requests received from the other processing units (PUs), such as the APU or the RPU, and the coordination and execution from other processing units through the power management API.
**IMPORTANT:** In the EEMI implementation for Zynq UltraScale+ MPSoC, the platform management unit (PMU) serves as the power management controller for the different processor units (PUs), such as the APU and the RPU. These APU/RPU act as a power management (PM) master node and make power management requests. Based on those requests, the PMU controls the power states of all PM slave nodes as well as the PM masters.

The Zynq UltraScale+ MPSoC device also supports inter-processor interrupts (IPIs), which are used as the basis for power-management related communication between the different processors. See this link to the “Interrupts” chapter of the *Zynq UltraScale+ MPSoC Technical Reference Manual* (UG1085) [Ref 1] for more detail on this topic.

### Zynq UltraScale+ MPSoC Power Management Software Architecture

To enable multiple processing units to cooperate in terms of power management, the software framework for the Zynq UltraScale+ MPSoC device provides an implementation of the power management API for managing heterogeneous multiprocessing systems.

The following figure illustrates the API-based power management software architecture.
Chapter 1: Introduction to the Power Management Framework

Power Management Framework Overview

The Zynq UltraScale+ MPSoC device power management framework (PMF) is based on an implementation of EEMI, see the Embedded Energy Management API Specification (UG1200) [Ref 2]. It includes APIs that consist of functions available to the processor units (PUs) to send messages to the power management controller, as well as callback functions in for the power management controller to send messages to the PUs. The APIs can be grouped into the following functional categories:

- Suspending and waking up PUs
- Slave device power management, such as memories and peripherals
- Miscellaneous
- Direct-access

API Calls and Responses

Power Management Communication using IPIs

In the Zynq UltraScale+ MPSoC device, the power management communication layer is implemented using inter-processor interrupts (IPIs), provided by the IPI block. See this link to the “Interrupts” chapter of the Zynq UltraScale+ MPSoC Technical Reference Manual (UG1085) [Ref 1] for more details on IPIs.

Each PU has a dedicated IPI channel with the power management controller, consisting of an interrupt and a payload buffer. The buffer passes the API ID and up to five arguments. The IPI interrupt to the target triggers the processing of the API, as follows:

- When calling an API function, a PU generates an IPI to the power management unit (PMU), prompting the execution of necessary power management action.
- The PMU performs each PM action atomically, meaning that the action cannot be interrupted.
- To support PM callbacks, which are used for notifications from the PMU to a PU, each PU implements handling of these callback IPIs.
Chapter 1: Introduction to the Power Management Framework

Acknowledge Mechanism

The Zynq UltraScale+ MPSoC power management framework (PMF) supports blocking and non-blocking acknowledges. In most API calls that offer an acknowledge argument, the caller can choose one of the following three acknowledge options:

- **REQUEST_ACK_NO**: No acknowledge requested
- **REQUEST_ACK_BLOCKING**: Blocking acknowledge requested
- **REQUEST_ACK_NON_BLOCKING**: Non-blocking acknowledge using callback requested

Multiple power management API calls are serialized because each processor unit (PU) uses a single IPI channel for the API calls. After one request is sent to the power management controller, the next one can be issued only after the power management controller has completed servicing the first one. Therefore, no matter which acknowledge mechanism is used, the caller can be blocked when issuing subsequent requests.

*No Acknowledge*

If no acknowledge is requested (**REQUEST_ACK_NO**), the power management controller processes the request without returning an acknowledge to the caller, otherwise an acknowledgment is sent.

*Blocking Acknowledge*

After initiating a PM request with the (**REQUEST_ACK_BLOCKING**) specified, a caller remains blocked as long as the power management controller does not provide the acknowledgment.

The platform management unit (PMU) writes the acknowledge values into the response portion of the IPI buffer before it clears the IPI interrupt. The caller reads the acknowledge values from the IPI buffer after the IPI observation register shows that the interrupt is cleared, which is when PMU has completed servicing the issued IPI. The IPI for the PU is disabled until the PMU is ready to handle the next request.

*Non-Blocking Acknowledge*

After initiating a PM request with the (**REQUEST_ACK_NON_BLOCKING**) specified, a caller does not wait for the platform management unit (PMU) to process that request. Moreover, the caller is free to perform some other activities while waiting for the acknowledge from the PMU.

After the PMU completes servicing the request, it writes the acknowledge values into the IPI buffer. Next, the PMU triggers the IPI to the caller PU to interrupt its activities, and to inform it about the sent acknowledge.
Non-blocking acknowledges are implemented using a callback function that is implemented by the calling PU, see `XPm_NotifyCb Callback`.

**Power Management Framework Layers**

There are different API layers in the power management framework (PMF) implementation for Zynq UltraScale+ MPSoC devices, which are, as follows:

- **Xilpm**: This is a library layer used for standalone applications in the different processing units, such as the APU and RPU.
- **ATF**: the ARM Trusted Firmware (ATF) contains its own implementation of the client-side PM framework. It is currently used by Linux operating systems.
- **PMUFW**: The power management unit firmware (PMUFW) runs on the power management unit (PMU) and implements of the power management API.

For more details, see the “Platform Management Unit Programming Model” section in “Chapter 6” of the *Zynq UltraScale+ MPSoC Technical Reference Manual* (UG1085) [Ref 1]

The following figure shows the interaction between the APU, the RPU, and the PMF APIs.

![API Layers Used with Bare-Metal Applications Only](image)
If the APU is running a complete software stack with an operating system, the \texttt{Xilpm} library is not used. Instead, the ATF running on EL3 implements the client-side power management API, and provides an secure monitor call (SMC)-based interface to the upper layers.

The following figure illustrates this behavior. See the ARMv8 manuals [Ref 4] for more details on the ARMv8 architecture and its different execution modes.

The following figure illustrates the PMF layers that are involved when running a full software stack on the APU.

![PM Framework Layers Involved When Running a Full Software Stack on the APU](image)

**Figure 1-4:** PM Framework Layers Involved When Running a Full Software Stack on the APU

**Typical Power Management API Call Flow**

Any entity involved in power management is referred to as a \textit{node}. The following sections describes how the power management framework (PMF) works with slave nodes allocated to the APU and the RPU.

**Requesting and Releasing Slave Nodes**

When a PU requires a slave node, either peripheral or memory, it must request that slave node using the power management API. After the slave node has performed its function and is no longer required, it must be released, allowing the slave node to be powered off.
The following figure shows the call flow for a use-case in which the APU and the RPU are sharing an OCM memory bank, ocm0.

**Figure 1-5: PM Framework Call Sequence for APU and RPU Sharing an OCM Memory Bank**

*Note:* The ocm0 memory remains powered on after the APU calls `Xpm_ReleaseNode`, because the RPU has also requested the same slave node. It is after the power management unit (PMU) also releases the ocm0 node that the PMU powers off the ocm0 memory.
Processor Unit Suspend and Resume

To allow a processor unit (PU) to be powered off, as opposed to just entering an idle state, an external entity is required to take care of the power-down and power-up transitions.

For the Zynq UltraScale+ MPSoC device, the platform management unit (PMU) is the responsible entity for performing all power state changes.

The processor unit (PU) notifies the PMU that a power state transition is being requested. The following figure illustrates the process.

![APU Suspend and Resume Procedure](image_url)

**Figure 1-6:** APU Suspend and Resume Procedure

The Self-Suspending a CPU/PU section details the suspend or resume procedure. Each PU depends on a slave node to be able to operate.
Chapter 1: Introduction to the Power Management Framework

Access Rights

To prevent a processing unit (PU) from making a request to a peripheral it is not supposed to access, the PMF includes a default assignment of peripherals (slave nodes) to each PU, as described in the following sections. See Node IDs: XPmNodeId for basic descriptions of the nodes.

APU Slave Nodes

By default, the APU is allowed to request or release only the following slave nodes (PMSlaves), as shown in the following table. This assignment is not currently modifiable.

Table 1-2: APU Controlled Slave Nodes

<table>
<thead>
<tr>
<th>Slave Nodes for the APU</th>
</tr>
</thead>
<tbody>
<tr>
<td>NODE_ADMA</td>
</tr>
<tr>
<td>NODE_AFI</td>
</tr>
<tr>
<td>NODE_APLL</td>
</tr>
<tr>
<td>NODE_CAN_0..1</td>
</tr>
<tr>
<td>NODE_DDR</td>
</tr>
<tr>
<td>NODE_DPLL</td>
</tr>
<tr>
<td>NODE_ETH_0..3</td>
</tr>
<tr>
<td>NODE(GLFW)</td>
</tr>
<tr>
<td>NODE_I2C_0..1</td>
</tr>
<tr>
<td>NODE_IOPLL</td>
</tr>
<tr>
<td>NODE_QSPI</td>
</tr>
<tr>
<td>NODE_TTC_0..3</td>
</tr>
<tr>
<td>NODE_UART_0..1</td>
</tr>
<tr>
<td>NODE_VPLL</td>
</tr>
</tbody>
</table>

RPU Slave Nodes

By default, the RPU is allowed to request/release the following slave nodes (PMSlaves):}

Table 1-3: Slave Nodes Controller by the RPU

<table>
<thead>
<tr>
<th>RPU Controlled Slave Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>NODE_APLL</td>
</tr>
<tr>
<td>NODE_DDR</td>
</tr>
<tr>
<td>NODE_DPLL</td>
</tr>
<tr>
<td>NODE_IOPLL</td>
</tr>
<tr>
<td>NODE_QCM_0..3</td>
</tr>
<tr>
<td>NODE_RPLL</td>
</tr>
<tr>
<td>NODE_SATA</td>
</tr>
<tr>
<td>NODE_TCM_0_A/B</td>
</tr>
<tr>
<td>NODE_TCM_1_A/B</td>
</tr>
<tr>
<td>NODE_TTC_0</td>
</tr>
<tr>
<td>NODE_USB_0..1</td>
</tr>
<tr>
<td>NODE_VPLL</td>
</tr>
</tbody>
</table>
Default Resources

Each PU depends on a set of slave nodes to be able to operate the PU. Each PU has a number of slave nodes required for the PU to be able to resume operation. The power management unit (PMU) pre-allocates those resources to the respective PU prior to waking the PU. For example, the memory nodes from which a CPU starts fetching instructions after being reset must be accessible before the CPU starts execution. The default requirements are, as follows:

- APU default requirements: L2 Cache, DDR
- RPU default requirements: All TCM memory banks

Allocating Resources at Boot-Time

To support Linux booting, the PMU assigns all nodes required to the APU upon initial boot. The following table lists the node IDs that are allocated to the APU during initialization, as follows:

Table 1-4: PMU Initialized Nodes

<table>
<thead>
<tr>
<th>Nodes Initialized by PMU</th>
</tr>
</thead>
<tbody>
<tr>
<td>NODE_ADMA</td>
</tr>
<tr>
<td>NODE_AFI</td>
</tr>
<tr>
<td>NODE_CAN_0</td>
</tr>
<tr>
<td>NODE_CAN_1</td>
</tr>
<tr>
<td>NODE_DP</td>
</tr>
<tr>
<td>NODE_GDMA</td>
</tr>
<tr>
<td>NODE_I2C_0</td>
</tr>
<tr>
<td>NODE_I2C_1</td>
</tr>
<tr>
<td>NODE_NAND</td>
</tr>
<tr>
<td>NODE_QSPI</td>
</tr>
<tr>
<td>NODE_SATA</td>
</tr>
<tr>
<td>NODE_SD_0</td>
</tr>
<tr>
<td>NODE_SPI_0</td>
</tr>
<tr>
<td>NODE_SPI_1</td>
</tr>
<tr>
<td>NODE_TTC_0</td>
</tr>
<tr>
<td>NODE_TTC_1</td>
</tr>
<tr>
<td>NODE_TTC_2</td>
</tr>
<tr>
<td>NODE_TTC_3</td>
</tr>
<tr>
<td>NODE_GPIO</td>
</tr>
<tr>
<td>NODE_SD_1</td>
</tr>
<tr>
<td>NODE_UART_0</td>
</tr>
<tr>
<td>NODE_UART_1</td>
</tr>
<tr>
<td>NODE_USB_0</td>
</tr>
<tr>
<td>NODE_USB_1</td>
</tr>
</tbody>
</table>

**RECOMMENDED:** When running bare-metal code on the APU, release all resources that are not necessary to allow the PMU to put those devices into lower power states, if possible.
Xilpm API Functions

Introduction

This chapter describes the API functions of the platform management unit (PMU).

Note: Linux software developers do not use the Xilpm library.

Advanced users who want to change or extend the Xilpm library, or other PMF components, need to consult the Embedded Energy Management API Specification (UG1200) [Ref 2]. The Xilpm library implements and extends the power management API.

Xilpm API Functions for Suspending Processor Units

The CPUs use the following API functions either to suspend themselves or to request suspend of other PUs or the entire system. This section describes the system-level functions that suspend processor units (PUs).

XPm_GetBootStatus

Description: This function returns information about the boot reason. If the boot is not a system startup but a resume, the power-down request bit field for this processor is cleared.

Example:

```c
XPmBootStatus XPM_GetBootStatus(void);
```

Returns:

- **PM_RESUME**: The boot reason is because of system resume.
- **PM_INITIAL_BOOT**: The boot is the initial system startup.
**XPlm_SelfSuspend**

**Description:** Request suspend or power down. This function is used by a CPU to declare that it is about to suspend itself.

After the power management controller processes this call, it waits for the requesting CPU to complete the suspend procedure and become ready to be put into a sleep state.

**Example:**

```c
XPmStatus XPlm_SelfSuspend(const enum XPlmNodeId node,
                           const u32 latency,
                           const u8 state,
                           const u64 address);
```

**Arguments:**

- **node:** Node ID of the CPU node to be suspended. See *Self-Suspending a CPU/PU*.
- **latency:** Maximum wake-up latency requirement in μs.
- **state:** Instead of specifying a maximum latency, a CPU can also explicitly request a certain power state. Valid values are:
  - **PM_APU_STATE_CPU_IDLE:** Only the CPU goes down.
  - **PM_APU_STATE_SUSPEND_TO_RAM:** OS is suspending, default resources like memories and PLLs can be released.
- **address:** Address from which to resume when awakened.

**Returns:**

This API returns a status:

- On **success:** XST_SUCCESS
- On **failure:** The appropriate error code from *Appendix B, Error Codes*. 
XPM_SuspendFinalize

**Description:** A CPU calls this function to signal to the power management controller that it is ready to suspend itself. A CPU calls this function signal to the power management controller that the requested suspend procedure is completed, and that it is ready to enter a sleep state. This function must be called after XPM_SelfSuspend().

- In the case of ARM CPUs, the wait for instruction (WFI) is executed as part of XPM_SuspendFinalize(), which in turn triggers an interrupt to the power management controller.
- The function does not give a return if the suspend is successful. Afterwards, the execution flow continues from the resume address provided in the XPM_SelfSuspend call.
- If XPM_SuspendFinalize does return, it means that an error has occurred, which indicates that XPM_SelfSuspend had not been called first.

**Example:**

```c
XPmStatus
XPM_SuspendFinalize(void);
```

**Returns:**

- Nothing.

XPM_RequestSuspend

**Description:** The caller can request a forced power-off of another PU or its power island or power domain. This turns off the power to the node unconditionally without interacting with the affected PU. This can power-off an unresponsive PU, in which case all resources of that PU release automatically.

A PU cannot request a force power-down for itself. A PU must use the full suspend sequence to power itself down, see XPM_SelfSuspend.

**Example:**

```c
XPmStatus
XPM_RequestSuspend(const enum XPmNodeId node,
                    const enum XPmRequestAck ack,
                    const u32 latency,
                    const u8 state);
```
Chapter 2: Xilpm API Functions

Arguments:

- node: Node ID of the PU node to be suspended. (See Acknowledge Mechanism).
- ack: Requested acknowledge type (See Acknowledge Mechanism).
- latency: Maximum wake-up latency requirement in μs.
- state: Instead of specifying a maximum latency, a PU can also explicitly request a certain power state.

Returns:

Based on the settings described in Acknowledge Mechanism, this API returns a status:

- On success: XST_SUCCESS
- On failure: The appropriate error code from Appendix B, Error Codes.

XPM_ForcePowerdown

Description: To force a PU to be powered down. One PU can request a forced power off of another PU or its power island or power domain. This can be used for killing an unresponsive PU, in which case all resources of that PU are XPM_RequestWakeup released automatically. Force power down cannot be requested by a PU for itself.

Example:

```c
XPMStatus
XPM_ForcePowerdown(const enum XPMNodeId node,
                      const enum XPMRequestAck ack);
```

Arguments:

- node: Node ID of the PU node or power island/domain to be powered down.
- ack: Requested acknowledge type. (See Acknowledge Mechanism).

Returns:

Based on the settings described in Acknowledge Mechanism, this API returns a status:

- On success: XST_SUCCESS.
- On failure: The appropriate error code from Appendix B, Error Codes.
XPM_AbortSuspend

**Description:** A CPU calls this function after an XPM_SelfSuspend() call to notify the power management controller that the CPU has aborted suspend, or in response to an init suspend request when the PU refuses to suspend.

For example, a PU begins its suspend procedure by calling XPM_SelfSuspend. Before the suspend procedure completes, an interrupt is received. The PU aborts the suspend procedure, and notifies the power management controller by calling the XPM_AbortSuspend() function while passing ABORT_REASON_WKUP_EVENT.

**Example:**

```c
XPmStatus
XPm_AbortSuspend(const enum XpmAbortReason reason);
```

**Arguments:**

- **reason:** Reason code why the suspend cannot be performed or completed:
  - ABORT_REASON_WKUP_EVENT: Local wakeup-event received.
  - ABORT_REASON_PU_BUSY: PU is busy.
  - ABORT_REASON_NO_PWRDN: No external power-down supported.
  - ABORT_REASON_UNKNOWN: Unknown error during suspend procedure.

**Returns:**

This API returns a status:

- On success: XST_SUCCESS.
- On failure: The appropriate error code from Appendix B, Error Codes.

XPM_RequestWakeup

**Description:** This function requests the power up of a CPU node within the same PU, or to power up another PU. If acknowledge is requested, the calling PU is notified by the power management controller after the wake-up is completed.

**Example:**

```c
XPmStatus
XPm_RequestWakeup(const enum XpmNodeId node,
                   const u8 setAddress,
                   const u64 address,
                   const enum XpmRequestAck ack);
```
Chapter 2: Xilpm API Functions

**Arguments:**

- **node**: Node ID of the CPU or PU to be powered/woken up.
- **setAddress**: Specifies whether the start address argument is being passed.
  - 0: Do not set start address (uses the default address of the processor).
  - 1: Set start address.
- **address**: Address from which to resume when woken up; this is used if `set_address` is 1.
- **ack**: Requested acknowledge type. (See Acknowledge Mechanism).

**Returns:**

This API returns a status:

- **On success**: `XST_SUCCESS`.
- **On failure**: The appropriate error code from Appendix B, Error Codes.

**Xpm_SetWakeUpSource**

**Description**: Declare a node to be used as a wakeup source. A PU calls this function to add or remove a wake-up source prior to going to suspend. The list of wake sources for a PU is cleared automatically when the PU is re-awakened or when one of its CPUs aborts the suspend procedure.

Declaring a node as a wakeup source ensures that the node is not powered off. It also causes the power management controller to configure the GIC proxy accordingly if the FPD is powered off.

**Example:**

```c
XStatus
Xpm_SetWakeUpSource(const enum XpmNodeId target,
                     const enum XpmNodeId wkup_node,
                     const u8 enable);
```

**Arguments:**

- **target**: Node ID of the target to be woken up.
- **wkup_node**: Node ID of the wakeup device.
- **enable**: Enable flag:
  - 1: The wakeup source is added to the list.
  - 0: The wakeup source is removed from the list.
Returns:

This API returns a status:

- On success: XST_SUCCESS.
- On failure: The appropriate error code from Appendix B, Error Codes.

Callback: XPM_InitSuspendCb

Description: Callback function implemented in each PU, allowing the PM controller to request that the PU suspend itself.

If the PU fails to act on this request the PM controller or the requesting PU can choose to perform a forceful power-down by calling XPM_ForcePowerdown.

Example:

```c
void XPM_InitSuspendCb(const enum XPM_SuspendReason reason,
                        const u32 latency,
                        const u32 state,
                        const u32 timeout);
```

Arguments:

- `reason`: Suspend reason:
  - SUSPEND_REASON_PU_REQ: Request by another PU.
  - SUSPEND_REASON_ALERT: Unrecoverable SYSMON alert.
  - SUSPEND_REASON_SHUTDOWN: System shutdown.
  - SUSPEND_REASON_RESTART: System restart.
- `latency`: Maximum wake-up latency in μs. This information can be used by the PU to decide what level of context saving is required.
- `state`: Targeted sleep/suspend state.
- `timeout`: Timeout in ms, specifying how much time a PU has to initiate its suspend procedure before being considered unresponsive.

Returns:

- Nothing.
**XPm_SystemShutdown**

**Description:** A PU can use this function either to shut down or restart the complete device.

In either case, the PM controller calls **Callback: XPm_InitSuspendCb** for each of the other PUs, allowing them to shut down gracefully. If a PU is asleep, the PM controller wakes the PU.

The PU making the `XPm_SystemShutdown()` should perform its own suspend procedure after calling this API. It will not receive an *init* suspend callback.

**Example:**

```c
XPmStatus
XPm_SystemShutdown(u32 type, u32 subtype)
```

**Arguments:**

- **type:** Should the system be restarted automatically?
  - PM_SHUTDOWN: N (0): No restart requested, system is powered off permanently.
  - PM_RESTART: Y (1): Restart is requested, system goes through a full reset.

- **subtype:** Which system?
  - PMF_SHUTDOWN_SUBTYPE_SUBSYSTEM (0): Reset/shut down the subsystem associated with the calling master. ROM-code will not re-run. Components that are not part of the calling subsystem must not be affected.
  - PMF_SHUTDOWN_SUBTYPE_PS_ONLY (1): Trigger a PS-only reset/shut down. PS re-runs full boot sequence when restarting. The PL must not be affected.
  - PMF_SHUTDOWN_SUBTYPE_SYSTEM (2): Trigger system reset/shut down. The whole SOC is reset/shut down.

**Returns:**

This API returns a status:

- On success: XST_SUCCESS.
- On failure: The appropriate error code from Appendix B, Error Codes.
Xilpm API Functions for Managing PM Slaves

The following is the API function for managing an power management slave (PM slave).

**XPM_RequestNode**

**Description**: request usage of a node. Used to request the use of a PM slave. When using this API call, a PU requests access to a slave device and asserts its requirements on that device.

Provided the PU is sufficiently privileged, the power management controller enables access to the memory-mapped region containing the control registers of that device. For devices that can only serve a single PU, any other privileged PU are now blocked from accessing this device until the node is released.

**Example**:

```c
XpmStatus
XPM_RequestNode(const enum XpmNodeId node,
                 const u32 capabilities,
                 const u32 qos,
                 const enum XpmRequestAck ack);
```

**Arguments**:

- **node**: Node ID of the PM slave requested.
- **capabilities**: Slave-specific capabilities required, and can be combined.
  - **PM_CAP_ACCESS**: Full access / functionality.
  - **PM_CAP_CONTEXT**: Preserve context.
  - **PM_CAP_WAKEUP**: Emit wake interrupts.
- **qos**: Quality of Service (0-100) required.
  - **Note**: This argument is not available.
- **ack**: Requested acknowledge type. (See **Acknowledge Mechanism**).

**Returns**:

This API returns a status:

- **On success**: XST_SUCCESS.
- **On failure**: The appropriate error code from Appendix B, Error Codes.
Chapter 2: Xilpm API Functions

Requesting PM Power Nodes

For typical usage scenarios, it is not necessary to request PM power nodes, because those are powered-on as soon as any device within a power island or power domain is requested.

Explicitly requesting a power node is only required in situations where the power domain or island must be on, without any device within that domain or island being required. The primary use case for requesting such a power node is for loading code into the tightly coupled memory (TCM) prior to taking the RPU cores out of reset. In that case, the RPU power island must be on for the TCMs to be externally accessible.

XPm_ReleaseNode

**Description:** A PU uses this function to release the usage of a PM slave node or a PM power node. This informs the power management controller that the node is not needed by that PU, potentially, allowing the node to be placed into an inactive state.

This could also result in power off of the respective power island or power domain, if the released slave was the last active slave in that power island or power domain.

**Example:**

```c
XPmStatus
XPm_ReleaseNode(const enum XpmNodeId node);
```

**Arguments:**

- **node:** Node ID of the PM slave.

**Returns:**

This API returns a status:

- On success: XST_SUCCESS.
- On failure: The appropriate error code from Appendix B, Error Codes.
Chapter 2: Xilpm API Functions

XPm_SetRequirement

**Description:** Change the requirements for a node in use. This function is used by a PU to announce a change in requirements for a specific slave node which is currently in use.

If this function is called after the last awake CPU within the PU calls XPm_SelfSuspend, the requirement change is performed after the CPU signals the end of suspend to the power management controller (for example: wait for interrupt (WFI)).

**Example:**

```c
XPmStatus
XPm_SetRequirement(const enum XpmNodeId node,
                    const u32 capabilities,
                    const u32 qos,
                    const enum XpmRequestAck ack);
```

**Arguments:**

- **node:** Node ID of the PM slave.
- **capabilities:** Slave-specific capabilities required.
- **qos:** Quality of Service (0-100) required.
- **ack:** Requested acknowledge type (See Acknowledge Mechanism).

**Returns:**

Based on the settings described in Acknowledge Mechanism, this API returns a status:

- **On success:** XST_SUCCESS.
- **On failure:** The appropriate error code from Appendix B, Error Codes.

XPm_SetMaxLatency

**Description:** A PU uses this function to announce a change in the maximum wake-up latency requirements for a specific slave node on that PU. Setting maximum wake-up latency can constrain the set of possible power states into which to put a resource.

**Example:**

```c
XPmStatus
XPm_SetMaxLatency(const enum XpmNodeId node,
                  const u32 latency);
```
Arguments:

- **node**: Node ID of the PM slave.
- **latency**: Maximum wake-up latency required.

Returns:

This API returns a status:

- **On success**: XST_SUCCESS.
- **On failure**: The appropriate error code from Appendix B, Error Codes.
Miscellaneous System-level API Functions

The miscellaneous system-level APIs are as follows:

**XPM_InitXilpm**

**Description:** This function initializes the Xilpm library by passing a valid pointer to the inter-processor interrupt (IPI) driver.

**IMPORTANT:** Call this function once prior to any other API call.

See this link to the “Interrupts” chapter in the Zynq UltraScale+ MPSoC Technical Reference Guide (UG1085) [Ref 1] for more information.

**Example:**

```c
XPMStatus
XPM_InitXilpm(XIpiPsu *IpiInst));
```

**Arguments:**

- *ipiInst*: Pointer to IPI driver instance.

**Returns:**

This API returns a status:

- **On success:** XST_SUCCESS.
- **On failure:** The appropriate error code from Appendix B, Error Codes.
XPM_GetApiVersion

**Description:** Request version information. This function is used to request the version number of the API running on the power management controller.

**Example:**

```c
XPmStatus
XPM_GetApiVersion(u32 *version);
```

**Arguments:**

- `version`: Returns the API version number, which is an unsigned integer value. Returns 0 if no power management unit firmware (PMUFW) present.

**Returns:**

This API returns a status:

- On success: XST_SUCCESS.
- On failure: The appropriate error code from Appendix B, Error Codes.
Xpm_ResetAssert

Description: This function asserts, releases, or pulses a reset.

Example:

XPmStatus
Xpm_ResetAssert(const u32 reset,
const u32 action);

Arguments:

• reset: ID of the reset line in question. The following table lists the reset values.

Table 2-1: Xpm_ResetAssert Reset Values

<table>
<thead>
<tr>
<th>Reset Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>PM_RESET_PCIE_CFG</td>
</tr>
<tr>
<td>PM_RESET_PCIE_CFG</td>
</tr>
<tr>
<td>PM_RESET_PCIE_CFG</td>
</tr>
<tr>
<td>PM_RESET_PCIE_CFG</td>
</tr>
<tr>
<td>PM_RESET_PCIE_CFG</td>
</tr>
<tr>
<td>PM_RESET_PCIE_CFG</td>
</tr>
<tr>
<td>PM_RESET_PCIE_CFG</td>
</tr>
<tr>
<td>PM_RESET_PCIE_CFG</td>
</tr>
<tr>
<td>PM_RESET_PCIE_CFG</td>
</tr>
<tr>
<td>PM_RESET_PCIE_CFG</td>
</tr>
<tr>
<td>PM_RESET_PCIE_CFG</td>
</tr>
<tr>
<td>PM_RESET_PCIE_CFG</td>
</tr>
<tr>
<td>PM_RESET_PCIE_CFG</td>
</tr>
<tr>
<td>PM_RESET_PCIE_CFG</td>
</tr>
<tr>
<td>PM_RESET_PCIE_CFG</td>
</tr>
<tr>
<td>PM_RESET_PCIE_CFG</td>
</tr>
<tr>
<td>PM_RESET_PCIE_CFG</td>
</tr>
<tr>
<td>PM_RESET_PCIE_CFG</td>
</tr>
<tr>
<td>PM_RESET_PCIE_CFG</td>
</tr>
<tr>
<td>PM_RESET_PCIE_CFG</td>
</tr>
<tr>
<td>PM_RESET_PCIE_CFG</td>
</tr>
<tr>
<td>PM_RESET_PCIE_CFG</td>
</tr>
<tr>
<td>PM_RESET_PCIE_CFG</td>
</tr>
<tr>
<td>PM_RESET_PCIE_CFG</td>
</tr>
<tr>
<td>PM_RESET_PCIE_CFG</td>
</tr>
<tr>
<td>PM_RESET_PCIE_CFG</td>
</tr>
<tr>
<td>PM_RESET_PCIE_CFG</td>
</tr>
<tr>
<td>PM_RESET_PCIE_CFG</td>
</tr>
<tr>
<td>PM_RESET_PCIE_CFG</td>
</tr>
<tr>
<td>PM_RESET_PCIE_CFG</td>
</tr>
<tr>
<td>PM_RESET_PCIE_CFG</td>
</tr>
<tr>
<td>PM_RESET_PCIE_CFG</td>
</tr>
<tr>
<td>PM_RESET_PCIE_CFG</td>
</tr>
<tr>
<td>PM_RESET_PCIE_CFG</td>
</tr>
<tr>
<td>PM_RESET_PCIE_CFG</td>
</tr>
<tr>
<td>PM_RESET_PCIE_CFG</td>
</tr>
<tr>
<td>PM_RESET_PCIE_CFG</td>
</tr>
<tr>
<td>PM_RESET_PCIE_CFG</td>
</tr>
<tr>
<td>PM_RESET_PCIE_CFG</td>
</tr>
<tr>
<td>PM_RESET_PCIE_CFG</td>
</tr>
<tr>
<td>PM_RESET_PCIE_CFG</td>
</tr>
</tbody>
</table>
| PM_RESET_PCIE_CFG | PM_RESET_PCIE_CTRL | PM_RESET_PCIE_BRIDGE | PM_RESET_PCIE(Clone)
Table 2-1: XPAR_ResetAssert Reset Values

<table>
<thead>
<tr>
<th>Reset Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>PM_RESET_GPU</td>
</tr>
<tr>
<td>PM_RESET_UART0</td>
</tr>
<tr>
<td>PM_RESET_RPU_R50</td>
</tr>
<tr>
<td>PM_RESET_DBG_LP</td>
</tr>
<tr>
<td>PM_RESET_GPO3_PL_13</td>
</tr>
<tr>
<td>PM_RESET_PS_ONLY</td>
</tr>
<tr>
<td>PM_RESET_GT</td>
</tr>
<tr>
<td>PM_RESET_UART1</td>
</tr>
<tr>
<td>PM_RESET_RPU_R51</td>
</tr>
<tr>
<td>PM_RESET_DBG_FP</td>
</tr>
<tr>
<td>PM_RESET_GPO3_PL_14</td>
</tr>
<tr>
<td>PM_RESET_PL</td>
</tr>
<tr>
<td>PM_RESET_SATA</td>
</tr>
<tr>
<td>PM_RESET_SPI0</td>
</tr>
<tr>
<td>PM_RESET_RPU_AMBA</td>
</tr>
<tr>
<td>PM_RESET_APLL</td>
</tr>
<tr>
<td>PM_RESET_GPO3_PL_15</td>
</tr>
<tr>
<td>PM_RESET_ACPU3_PWRON</td>
</tr>
<tr>
<td>PM_RESET_SPI1</td>
</tr>
<tr>
<td>PM_RESET_OCM</td>
</tr>
<tr>
<td>PM_RESET_DPLL</td>
</tr>
<tr>
<td>PM_RESET_GPO3_PL_16</td>
</tr>
<tr>
<td>PM_RESET_ACPU2_PWRON</td>
</tr>
<tr>
<td>PM_RESET_SDIO0</td>
</tr>
<tr>
<td>PM_RESET_RPU_PGE</td>
</tr>
<tr>
<td>PM_RESET_VPLL</td>
</tr>
<tr>
<td>PM_RESET_GPO3_PL_17</td>
</tr>
<tr>
<td>PM_RESET_ACPU1_PWRON</td>
</tr>
<tr>
<td>PM_RESET_SDIO1</td>
</tr>
<tr>
<td>PM_RESET_USB0_CORERESET</td>
</tr>
<tr>
<td>PM_RESET_IOPLL</td>
</tr>
<tr>
<td>PM_RESET_GPO3_PL_18</td>
</tr>
</tbody>
</table>

- action:
  - PM_RESET_ACTION_RELEASE: Release reset.
  - PM_RESET_ACTION_ASSERT: Assert reset.
  - PM_RESET_ACTION_PULSE: Pulse reset.

Returns:

This API returns a status:

- On success: XST_SUCCESS.
- On failure: The appropriate error code from Appendix B, Error Codes.
**Xpm_ResetGetStatus**

**Description:** This function determines the status of the selected reset line.

**Example:**

```c
XpmStatus
XPM_ResetGetStatus(const u32 reset,
                    u32 *const status);
```

**Arguments:**

- `reset`: ID of the reset line in question.
- `status`: Returns the status of the selected reset line:
  - 0: Released.
  - 1: Asserted.

**Returns:**

This API returns a status:

- On success: `XST_SUCCESS`. If the API status is `XST_SUCCESS`, this API also returns the status of the selected reset line.
- On failure: The appropriate error code from Appendix B, Error Codes.

**Xpm_GetNodeStatus**

**Description:** This function obtains information about the current state of a node. The caller must pass a pointer to an `XPm_NodeStatus()` structure, which must be pre-allocated by the caller.

**Example:**

```c
XpmGetNodeStatus(const enum XPmNodeId node,
                  XPm_NodeStatus* const nodestatus);
```

**Arguments:**

- `node`: ID of the component or sub-system in question.
- `nodestatus`: Used to return the complete status of the node.
- `status`: The current power state of the requested node, along with the requirement and usage settings?
  - For CPU nodes:
    - 0: CPU is powered down.
    - 1: CPU is active (powered up).
- 2: CPU is suspending (powered up).
- For power islands and power domains:
  - 0: Island is powered down.
  - 1: Island is powered up.
- For PM slaves:
  - 0: Slave is powered down.
  - 1: Slave is powered up.
  - 2: Slave is in retention.

- **requirement:** Slave nodes only: Returns the current requirements of the requesting PU.
- **usage:** Slave nodes only: Returns current usage status of the node:
  - 0: Node is not used by any PU.
  - 1: Node is used by caller exclusively.
  - 2: Node is used by other PU(s) only.
  - 3: Node is used by caller and by other PU(s).

**Returns:**

This API returns a status:

- On success: `XST_SUCCESS`.
- On failure: The appropriate error code from Appendix B, Error Codes.

**Xpm_RegisterNotifier**

**Description:** A PU can call this function to request notification when a qualified event occurs. One can request notification for a specific or any event related to a specific node. The `Xpm.NotifyCb Callback` callback function performs the notification.

The caller initializes the notifier object before invoking the `Xpm.RegisteredNotifier()` function. While the notifier is registered, the notifier object is not modified. The notifier object contains the following data related to the notification:
Chapter 2: Xilpm API Functions

Example:

```c
XStatus
XPM_RegisterNotifier(XPM_Notifier* const notifier);
```

Arguments:

- **notifier**: Pointer to the notifier object to be associated with the requested notification.

Returns:

This API returns a status:

- On success: `XST_SUCCESS`
- On failure: The appropriate error code from Appendix B, Error Codes.
**Xpm_GetOpCharacteristic**

**Description:** This function requests the PM controller to return information about an operating characteristic of a component.

---

**CAUTION!** *Power and latency are the only values returned; those values are not accurate or reliable.*

---

**Example:**

```c
XpmStatus
Xpm_GetOpCharacteristic(const enum XpmNodeId node,
const u32 type,
const u32 result);
```

**Arguments:**

- nodeID: ID of the component or sub-system in question.
- type: Type of operating characteristic requested:
  - power: Current power consumption.
  - latency: Current latency in $\mu$s to return to active state.
  - temperature: Current temperature. *(Unsupported)*
- result: A pointer to return the requested operating characteristic.

**Returns:**

This API returns a status:

- On success: XST_SUCCESS.
- On failure: The appropriate error code from Appendix B, Error Codes.

**Xpm_UnregisterNotifier**

**Description:** Unregister the notifier for events related to a node. A PU calls this function to unregister for the previously requested notifications.

**Example:**

```c
XStatus
Xpm_UnregisterNotifier(Xpm_Notifier* const notifier);
```

**Arguments:**

- notifier: Pointer to the notifier object associated with the previously requested notification.
**Callback: XPm_AcknowledgeCb**

**Description:** This callback is invoked by the PM controller in response to any request where an acknowledge callback was requested, such as where the `ack` argument passed by the PU was `REQUEST_ACK_NON_BLOCKING`.

**Example:**

```c
void XPm_AcknowledgeCb(const enum XPmNodeId node,
                        const XPmStatus status,
                        const u32 node_state);
```

**Arguments:**

- `node`: ID of the component or sub-system in question.
- `status`: Status of the operation:
  - `OK`: The operation completed successfully.
  - `ERR`: The requested operation failed.
- `node_state`: The current operating state of the component or sub-system in question.

---

**XPm_NotifyCb Callback**

**Description:** The PM controller callback invokes this callback if an event that the PU is registered for has occurred. The `XPm_NotifyCb()` callback invokes the user-defined callback function specified in the notifier data structure passed when `XPm_RegisterNotifier` was called.

**Example:**

```c
void XPm_NotifyCb(const enum XPmNodeId node,
                   const u32 event,
                   const u32 node_state);
```

**Arguments:**

- `node`: ID of the node to which the event notification is related.
- `event`: ID of the event.
- `node_state`: Current operating state of the node.

**Returns:**

- Nothing.
Direct Control API Functions

The following set of API functions can be used to allow access to resources that are not directly accessible due to bus access restrictions. Access authorization is verified by the power management controller.

**XPm_MmioWrite**

**Description:** This function writes a value to an address that is not accessible directly, such as registers in the clock control unit. If the PM controller does not grant access, the function returns an error code. For more information, see the Zynq UltraScale+ MPSoC Technical Reference Manual [Ref 1] section on “XPPU.”

**Example:**

```c
XPmStatus
    XPm_MmioWrite(const u32 address,
                   const u32 mask,
                   const u32 value);
```

**Arguments:**

- **address:** Physical 32-bit address of memory-mapped register to which to write.
- **mask:** 32-bit bitmask value used to limit write to specific bits in the register.
- **value:** Values to write to the register bits specified by the mask.

**Returns:**

This API returns a status:

- On success: **XST_SUCCESS**.
- On failure: The appropriate error code from Appendix B, Error Codes.
XPm_MmioRead

Description: This function reads a value from an address that is not directly accessible. If the PM controller does not grant access, this function returns an error code.

Example:

```c
XPmStatus
XPm_MmioRead(const u32 address,
              u32 *const value);
```

Arguments:

- **address**: Physical 32-bit address of memory mapped register to read from.
- **value**: Returns the 32-bit value read from the register.

Returns:

This API returns a status:

- On success: **XST_SUCCESS**.
- On failure: The appropriate error code from Appendix B, Error Codes.
Using the API for Power Management

Introduction

This chapter contains detailed instructions on how to use the Xilinx® power management framework (PMF) APIs to carry out common power management tasks.

Implementing Power Management on a Processor Unit

The Xilpm library provides the interface that the software executing on a PU can use to initiate the power management API calls. To make the API calls, the software executing on a processor unit (PU) needs to use the Xilpm library.

See the Xilinx Software Developer Kit Help (UG782) [Ref 7] for information on how to include the Xilpm library in a project.

Initializing the Xilpm Library

Before initiating any power management API calls, you must initialize the Xilpm library by calling XPm_InitXilpm, and passing a pointer to a properly initialized inter-processor interrupt (IPI) driver instance.

See this link to the “Interrupts” chapter of the Zynq UltraScale+ MPSoC Technical Reference Manual (UG1085) [Ref 1]. for more information regarding IPIs.

Working with Slave Devices

The Zynq® UltraScale+™ MPSoC device power management framework (PMF) contains functions dedicated to managing slave devices (also referred to as PM slaves), such as memories and peripherals. Processor units (PUs) use these functions to inform the power management controller about the requirements (such as capabilities and wake-up latencies) for those devices. The power management controller ensures that at all times each device resides in the lowest possible power state meeting the requirements from all eligible PUs.
Requesting and Releasing a Node

A PU uses the `XPm_RequestNode` API to request access to a slave device and assert its requirements on that device. Provided the PU is allowed to access the slave (see Access Rights), the power management controller ensures the requested device is powered on and placed into its active state. For devices that can only be serving a single PU, any other PU is then blocked from requesting this device.

The power management controller assigns access permissions to one or multiple PUs for each PM slave. When a PU requests a PM slave, the power management controller checks the privilege configuration to determine if the PU is allowed to use the requested PM slave. You must reconfigure the privilege assignment for the power management controller in source code.

After a device is no longer used, the PU typically calls the `XPm_ReleaseNode` function to allow the PM controller to re-evaluate the power state of that device, and potentially place it into a low-power state. It also then allows other PUs to request that device.

Changing Requirements

During use, a PU can change the requirements it asserts on the capability of a PM slave by using the `XPm_SetRequirement` API. Typically, a request for node use occurs when a device is no longer actively used; however, the appropriate wake-interrupts or preservation of the context of the node must be issued.

The following example call changes the requirement for the `node` argument to require wake-interrupts only:

```c
XPm_SetRequirement(node, PM_CAP_WAKEUP, 0, REQUEST_ACK_NO);
```

At some point, the PU could have no requirements, and then it has the option to release the PM slave or to set its requirements to zero.

**IMPORTANT:** Setting requirements of a node to zero is not equivalent to releasing the PM slave, because by releasing the PM slave, a PU releases its access rights, potentially allowing other PUs to use this device.

If the requirements are set to zero, the power management controller can place the device into a lower-power state while still keeping the access rights of the PU in place.

When multiple PUs share a PM slave (this applies mostly to memories), the power management controller configures a power state of the PM slave that satisfies all requirements of the requesting PUs.
Self-Suspending a CPU/PU

A PU can be a cluster of CPUs. The APU is a PU, that has four CPUs. An RPU has two CPUs.

To suspend itself, a CPU must inform the power management controller about its intent by calling the XPm_SelfSuspend function. The following actions then occur:

- After the XPm_SelfSuspend() call is processed, none of the future interrupts can prevent the CPU from entering a sleep state. To ensure such behavior in the case of the APU and RPU, after the XPm_SelfSuspend() call has completed, all of the interrupts to a CPU which redirects the interrupts to the power management controller as GIC wake interrupts.

- The power management controller then waits for the CPU to finalize the suspend procedure. The PU informs the power management controller that it is ready to enter a sleep state by calling XPm_SuspendFinalize.

- The XPm_SuspendFinalize() function is architecture-dependent. It ensures that any outstanding power management API call is processed, then executes the architecture-specific suspend sequence, which also signals the suspend completion to the power management controller.

- For ARM processors such as the APU and RPU, the XPm_SuspendFinalize() function uses the wait for interrupt (WFI) instruction, which suspends the CPU and triggers an interrupt to the power management controller.

- When the suspend completion is signaled to the power management controller, the power management controller places the CPU into reset, and can power down the power island of the CPU, provided that no other component within the island is currently active.

- Interrupts enabled through the GIC interface of the CPU redirect to the power management controller (PMC) as a GIC wake interrupt assigned to that particular CPU. Because the interrupts are redirected, the CPU can only be woken up using the power management controller.

- Suspending a PU requires suspending all of its CPUs individually.

Resuming Execution

A CPU can be woken up either by a wake interrupt triggered by a hardware resource or by an explicit wake request using the XPm_RequestWakeup API.

The CPU starts executing from the resume address provided with the XPm_RequestSuspend call.
Chapter 3: Using the API for Power Management

Setting up a Wake-up Source

The PM controller can power down the entire FPD if none of the FPD devices are in use and existing latency requirements allow this action. If the FPD is powered off and the APU is to be woken up by an interrupt triggered by a device in the LPD, the GIC Proxy must be configured to allow propagation of FPD wake events. The APU can ensure this by calling XPm_SetWakeUpSource for all devices that might need to issue wake interrupts.

Hence, prior to suspending, the APU must call XPm_SetWakeUpSource(NODE_APU, node, 1) to add the required slaves as a wake-up source. The APU can then set the requirements to zero for all slaves it is using. After the APU finalizes its suspend procedure, and provided that no other PU is using any resource in the FPD, the PM controller powers off the entire FPD and configures the GIC proxy to enable propagation of the wake event of the LPD slaves.

Aborting a Suspend Procedure

If a PU decides to abort the suspend procedure after calling the XPm_SelfSuspend function, it must inform the power management controller about the aborted suspend by calling the XPm_AbortSuspend function.

Handling PM Slaves During the Suspend Procedure

A PU that suspends itself must inform the power management controller about its changed requirements on the peripherals and memories in use. If a PU fails inform the power management controller, all of the used devices remain powered on. Typically, for memories you must ensure that their context is preserved by using the following function:

XPm_SetRequirement(node, PM_CAP_CONTEXT, 0, REQUEST_ACK_NO);

When setting requirements for a PM slave during the suspend procedure; such as after calling XPm_SelfSuspend, the setting is deferred until the CPU finishes the suspend. This deference ensures that devices that are needed for completing the suspend procedure can enter a low power state after the calling CPU finishes suspend.

A common example is instruction memory, which a CPU can access until the end of a suspend. After the CPU suspends a memory, that memory can be placed into retention. All deferred requirements reverse automatically before the respective CPU is woken up.

When an entire PU suspends, the last awake CPU within the PU must manage the changes to the devices.
Chapter 3: Using the API for Power Management

Example Code for Suspending an APU/RPU

There the following is an example of source code for suspending the APU or RPU:

```c
/* Base address of vector table (reset-vector) */
extern void *_vector_table;
/* Inform PM controller that APU_0 intends to suspend */
XPm_SelfSuspend(NODE_APU_0, MAX_LATENCY, 0,
                (u64)&_vector_table);
/**
 * Set requirements for OCM banks to preserve their context.
 * The PM controller will defer putting OCMs into retention
 * until the suspend is finalized
 */
XPm_SetRequirement(NODE_OCM_BANK_0, PM_CAP_CONTEXT, 0,
                   REQUEST_ACK_NO);
XPm_SetRequirement(NODE_OCM_BANK_1, PM_CAP_CONTEXT, 0,
                   REQUEST_ACK_NO);
XPm_SetRequirement(NODE_OCM_BANK_2, PM_CAP_CONTEXT, 0,
                   REQUEST_ACK_NO);
XPm_SetRequirement(NODE_OCM_BANK_3, PM_CAP_CONTEXT, 0,
                   REQUEST_ACK_NO);

/* Flush data cache */
Xil_DCacheFlush();
/* Inform PM controller that suspend procedure is completed */
XPm_SuspendFinalize();
```

Suspending the Entire FPD Domain

To power-down the entire full power domain, the power management controller must suspend the APU at a time when none of the FPD devices is in use. After this condition is met, the power management controller can power-down the FPD automatically. The power management controller powers down the FPD if no latency requirements constrain this action, otherwise the FPD remains powered on.

Forcefully Powering Down the FPD

There is the option to force the FPD to power-down by calling the function XPm_ForcePowerdown. This requires that the requesting PU has proper privileges configured in the power management controller. The power management controller releases all PM Slaves used by the APU automatically.

**Note:** This force method is typically not recommended, especially when running complex operating systems on the APU because it could result in loss of data or system corruption, due to the OS not suspending itself gracefully.

**IMPORTANT:** Use the XPm_RequestSuspend API.
Interacting With Other Processing Units

Suspending a PU

A PU can request that another PU be suspended by calling `Xpm_RequestSuspend`, and passing the targeted node name as an argument.

This causes the power management controller to call `Callback: Xpm_InitSuspendCb()`, which is a callback function implemented in the target PU. The target PU then initiates its own suspend procedure, or call `Xpm_AbortSuspend` and specify the abort reason. For example, you can request an APU to suspend with the following command:

```c
Xpm_RequestSuspend(NODE_APU, REQUEST_ACK_CB_STANDARD, MAX_LATENCY, 0);
```

The following diagram shows the general sequence triggered by a call to the `Xpm_RequestSuspend`.

![Diagram showing the sequence of events triggered by `Xpm_RequestSuspend`]

*Figure 3-1: APU initiating suspend for the RPU by calling Xpm_RequestSuspend*
Waking a PU

Additionally, a PU can request the wake-up of one of its CPUs or of another PU by calling `XPm_RequestWakeup`.

- When processing the call, the power management controller causes a target CPU or PU to be awakened.
- If a PU is the target, only one of its CPUs is woken-up by this request.
- The CPU chosen by the power management controller is considered the primary CPU within the PU.

The following is an example of a wake-up request:

```c
XPm_RequestWakeup(NODE_APU_1, REQUEST_ACK_NO);
```
Argument Value Definitions

Introduction
The following are value definitions for the various arguments used in the power management APIs, as defined in the file pm_defs.h.

Node IDs: XPmNodeId

The following table lists the defined Node IDs in the Zynq® UltraScale™+ MPSoC device.

<table>
<thead>
<tr>
<th>Node ID</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>NODE_UNKNOWN</td>
<td></td>
<td>0U</td>
</tr>
<tr>
<td>NODE_APU</td>
<td>APU Controller</td>
<td>1U</td>
</tr>
<tr>
<td>NODE_APU_0</td>
<td>APU Controller 0</td>
<td>2U</td>
</tr>
<tr>
<td>NODE_APU_1</td>
<td>APU Controller 1</td>
<td>3U</td>
</tr>
<tr>
<td>NODE_APU_2</td>
<td>APU Controller 2</td>
<td>4U</td>
</tr>
<tr>
<td>NODE_APU_3</td>
<td>APU Controller 3</td>
<td>5U</td>
</tr>
<tr>
<td>NODE_RPU</td>
<td>RPU Controller</td>
<td>6U</td>
</tr>
<tr>
<td>NODE_RPU_0</td>
<td>RPU Controller 0</td>
<td>7U</td>
</tr>
<tr>
<td>NODE_RPU_1</td>
<td>RPU Controller 1</td>
<td>8U</td>
</tr>
<tr>
<td>NODE_PL</td>
<td>PL Controller</td>
<td>9U</td>
</tr>
<tr>
<td>NODE_FPD</td>
<td>FPD Controller</td>
<td>10U</td>
</tr>
<tr>
<td>NODE_OCM_BANK_0</td>
<td>OCM Memory Tile 0</td>
<td>11U</td>
</tr>
<tr>
<td>NODE_OCM_BANK_1</td>
<td>OCM Memory Tile 1</td>
<td>12U</td>
</tr>
<tr>
<td>NODE_OCM_BANK_2</td>
<td>OCM Memory Tile 2</td>
<td>13U</td>
</tr>
<tr>
<td>NODE_OCM_BANK_3</td>
<td>OCM Memory Tile 3</td>
<td>14U</td>
</tr>
<tr>
<td>NODE_TCM_0_A</td>
<td>Tightly coupled memory (0A)</td>
<td>15U</td>
</tr>
<tr>
<td>NODE_TCM_0_B</td>
<td>Tightly coupled memory (0B)</td>
<td>16U</td>
</tr>
<tr>
<td>Node ID</td>
<td>Description</td>
<td>Node Value</td>
</tr>
<tr>
<td>------------------</td>
<td>--------------------------------------------------</td>
<td>------------</td>
</tr>
<tr>
<td>NODE_TCM_1_A</td>
<td>Tightly coupled memory (1A)</td>
<td>17U</td>
</tr>
<tr>
<td>NODE_TCM_1_B</td>
<td>Tightly coupled memory (1B)</td>
<td>18U</td>
</tr>
<tr>
<td>NODE_L2</td>
<td>L2 Cache system</td>
<td>19U</td>
</tr>
<tr>
<td>NODE_GPU_PP_0</td>
<td>Graphics Processing Unit 0</td>
<td>20U</td>
</tr>
<tr>
<td>NODE_GPU_PP_1</td>
<td>Graphics Processing Unit 1</td>
<td>21U</td>
</tr>
<tr>
<td>NODE_USB_0</td>
<td>USB Controller 0</td>
<td>22U</td>
</tr>
<tr>
<td>NODE_USB_1</td>
<td>USB Controller 1</td>
<td>23U</td>
</tr>
<tr>
<td>NODE_TTC_0</td>
<td>Triple-timer Counter 0</td>
<td>24U</td>
</tr>
<tr>
<td>NODE_TTC_1</td>
<td>Triple-timer Counter 1</td>
<td>25U</td>
</tr>
<tr>
<td>NODE_TTC_2</td>
<td>Triple-timer Counter 2</td>
<td>26U</td>
</tr>
<tr>
<td>NODE_TTC_3</td>
<td>Triple-timer Counter 3</td>
<td>27U</td>
</tr>
<tr>
<td>NODE_SATA</td>
<td>SATA Controller</td>
<td>28U</td>
</tr>
<tr>
<td>NODE_ETH_0</td>
<td>Gigabit Ethernet Controller 0</td>
<td>29U</td>
</tr>
<tr>
<td>NODE_ETH_1</td>
<td>Gigabit Ethernet Controller 1</td>
<td>30U</td>
</tr>
<tr>
<td>NODE_ETH_2</td>
<td>Gigabit Ethernet Controller 2</td>
<td>31U</td>
</tr>
<tr>
<td>NODE_ETH_3</td>
<td>Gigabit Ethernet Controller 3</td>
<td>32U</td>
</tr>
<tr>
<td>NODE_UART_0</td>
<td>UART Controller 0</td>
<td>33U</td>
</tr>
<tr>
<td>NODE_UART_1</td>
<td>UART Controller 1</td>
<td>34U</td>
</tr>
<tr>
<td>NODE_SPI_0</td>
<td>SPI Controller 0</td>
<td>35U</td>
</tr>
<tr>
<td>NODE_SPI_1</td>
<td>SPI Controller 1</td>
<td>36U</td>
</tr>
<tr>
<td>NODE_I2C_0</td>
<td>SPI Controller 2</td>
<td>37U</td>
</tr>
<tr>
<td>NODE_I2C_1</td>
<td>SPI Controller 3</td>
<td>38U</td>
</tr>
<tr>
<td>NODE_SD_0</td>
<td>SD/SDIO Controller 0</td>
<td>39U</td>
</tr>
<tr>
<td>NODE_SD_1</td>
<td>SD/SDIO Controller 1</td>
<td>40U</td>
</tr>
<tr>
<td>NODE_DP</td>
<td>DisplayPort Controller</td>
<td>41U</td>
</tr>
<tr>
<td>NODE_GDMA</td>
<td>FPD DMA Controller</td>
<td>42U</td>
</tr>
<tr>
<td>NODE_ADMA</td>
<td>APU DMA</td>
<td>43U</td>
</tr>
<tr>
<td>NODE_NAND</td>
<td>NAND Controller</td>
<td>44U</td>
</tr>
<tr>
<td>NODE_QSPI</td>
<td>QSPI Controller</td>
<td>45U</td>
</tr>
<tr>
<td>NODE_GPIO</td>
<td>GPIO Controller</td>
<td>46U</td>
</tr>
<tr>
<td>NODE_CAN_0</td>
<td>CAN Controller 0</td>
<td>47U</td>
</tr>
<tr>
<td>NODE_CAN_1</td>
<td>CAN Controller 1</td>
<td>48U</td>
</tr>
<tr>
<td>NODE_AFI</td>
<td>AFI Block</td>
<td>49U</td>
</tr>
<tr>
<td>NODE_APLL</td>
<td>APU PLL</td>
<td>50U</td>
</tr>
<tr>
<td>NODE_VPLL</td>
<td>Video PLL</td>
<td>51U</td>
</tr>
</tbody>
</table>
Appendix A: Argument Value Definitions

### Table A-1: XPmNodeIds (Cont’d)

<table>
<thead>
<tr>
<th>Node IDs for Zynq UltraScale+ MPSoC Devices</th>
</tr>
</thead>
<tbody>
<tr>
<td>NODE_DPLL</td>
</tr>
<tr>
<td>NODE_RPLL</td>
</tr>
<tr>
<td>NODE_IOPLL</td>
</tr>
<tr>
<td>NODE_DDR</td>
</tr>
<tr>
<td>NODE_IPI_APU</td>
</tr>
<tr>
<td>NODE_IPI_RPU_0</td>
</tr>
<tr>
<td>NODE_GPU</td>
</tr>
<tr>
<td>NODE_PCIE</td>
</tr>
</tbody>
</table>

---

### Acknowledge Request Types: XPmRequestAck

```c
enum XPmRequestAck {
    REQUEST_ACK_NO = 1,
    REQUEST_ACK_BLOCKING,
    REQUEST_ACK_NON_BLOCKING,
};
```

---

### Abort Reasons: XPmAbortReason

```c
enum XPmAbortReason {
    ABORT_REASON_WKUP_EVENT = 100,
    ABORT_REASON_PU_BUSY,
    ABORT_REASON_NO_PWRDN,
    ABORT_REASON_UNKNOWN,
};
```

---

### Suspend Reasons

```c
enum XPmSuspendReason {
    SUSPEND_REASON_PU_REQ = 201,
    SUSPEND_REASON_ALERT,
    SUSPEND_REASON_SYS_SHUTDOWN,
};
```
Operating Characteristic Types: XPmOpCharType

```c
enum XPmOpCharType {
    PM_OPCHAR_TYPE_POWER = 1,
    PM_OPCHAR_TYPE_ENERGY,
    PM_OPCHAR_TYPE_TEMP,
};
```

Notify Event Types: XPmNotifyEvent

```c
enum XPmNotifyEvent {
    EVENT_STATE_CHANGE = 1,
    EVENT_ZERO_USERS = 2,
    EVENT_ERROR_CONDITION = 4,
};
```

Reset Line IDs

The following table lists the reset line IDs for the Zynq UltraScale+ MPSoC device.

```c
enum XPmReset {
};
```

<table>
<thead>
<tr>
<th>Table A-2: Reset Line IDs</th>
<th>Reset Line IDs</th>
</tr>
</thead>
<tbody>
<tr>
<td>XILPM_RESET_PCIE_CFG = 1000</td>
<td>XILPM_RESET_DDR</td>
</tr>
<tr>
<td>XILPM_RESET_PCIE_BRIDGE</td>
<td>XILPM_RESET_APM_FPD</td>
</tr>
<tr>
<td>XILPM_RESET_PCIE_CTRL</td>
<td>XILPM_RESET_SOFT</td>
</tr>
<tr>
<td>XILPM_RESET_DP</td>
<td>XILPM_RESET_GEM0</td>
</tr>
<tr>
<td>XILPM_RESET_SWDT_CRF</td>
<td>XILPM_RESET_GEM1</td>
</tr>
<tr>
<td>XILPM_RESET_AFI_FM5</td>
<td>XILPM_RESET_GEM2</td>
</tr>
<tr>
<td>XILPM_RESET_AFI_FM4</td>
<td>XILPM_RESET_GEM3</td>
</tr>
</tbody>
</table>
Table A-2: Reset Line IDs (Cont’d)

<table>
<thead>
<tr>
<th>Reset Line IDs</th>
</tr>
</thead>
<tbody>
<tr>
<td>XILPM_RESET_AFI_FM3</td>
</tr>
<tr>
<td>XILPM_RESET_AFI_FM2</td>
</tr>
<tr>
<td>XILPM_RESET_AFI_FM1</td>
</tr>
<tr>
<td>XILPM_RESET_AFI_FM0</td>
</tr>
<tr>
<td>XILPM_RESET_GDMA</td>
</tr>
<tr>
<td>XILPM_RESET_GPU_PP1</td>
</tr>
<tr>
<td>XILPM_RESET_GPU_PP0</td>
</tr>
<tr>
<td>XILPM_RESET_GPU</td>
</tr>
<tr>
<td>XILPM_RESET_GT</td>
</tr>
<tr>
<td>XILPM_RESET_SATA</td>
</tr>
<tr>
<td>XILPM_RESET_ACPU3_PWRON</td>
</tr>
<tr>
<td>XILPM_RESET_ACPU2_PWRON</td>
</tr>
<tr>
<td>XILPM_RESET_ACPU1_PWRON</td>
</tr>
<tr>
<td>XILPM_RESET_ACPU0_PWRON</td>
</tr>
<tr>
<td>XILPM_RESET_APU_L2</td>
</tr>
<tr>
<td>XILPM_RESET_ACPU3</td>
</tr>
</tbody>
</table>

Notify Event Types: XPmNotifyEvent

```
enum XPmNotifyEvent {
    EVENT_STATE_CHANGE = 1,
    EVENT_ZERO_USERS = 2,
    EVENT_ERROR_CONDITION = 4,
};
```
Appendix A: Argument Value Definitions

Reset Line IDs

class XpmReset {
XILPM_RESET_PCIE_CFG = 1000,
XILPM_RESET_PCIE_BRIDGE,
XILPM_RESET_PCIE_CTRL,
XILPM_RESET_DP,
XILPM_RESET_SWDT_CRF,
XILPM_RESET_API_FM5,
XILPM_RESET_API_FM4,
XILPM_RESET_API_FM3,
XILPM_RESET_API_FM2,
XILPM_RESET_API_FM1,
XILPM_RESET_API_FM0,
XILPM_RESET_GDMA,
XILPM_RESET_GPU_PP1,
XILPM_RESET_GPU_PP0,
XILPM_RESET_GPU,
XILPM_RESET_GT,
XILPM_RESET_SATA,
XILPM_RESET_ACPU3_PWRON,
XILPM_RESET_ACPU2_PWRON,
XILPM_RESET_ACPU1_PWRON,
XILPM_RESET_ACPU0_PWRON,
XILPM_RESET_APU_L2,
XILPM_RESET_ACPU3,
XILPM_RESET_ACPU2,
XILPM_RESET_ACPU1,
XILPM_RESET_ACPU0,
XILPM_RESET_DDR,
XILPM_RESET_APM_FPD,
XILPM_RESET_SOFT,
XILPM_RESET_GEM0,
XILPM_RESET_GEM1,
XILPM_RESET_GEM2,
XILPM_RESET_GEM3,
XILPM_RESET_QSPI,
XILPM_RESET_UART0,
XILPM_RESET_UART1,
XILPM_RESET_SPI0,
XILPM_RESET_SPI1,
Appendix A: Argument Value Definitions

XILPM_RESET_SDIO0,
XILPM_RESET_SDIO1,
XILPM_RESET_CAN0,
XILPM_RESET_CAN1,
XILPM_RESET_I2C0,
XILPM_RESET_I2C1,
XILPM_RESET_TTC0,
XILPM_RESET_TTC1,
XILPM_RESET_TTC2,
XILPM_RESET_TTC3,
XILPM_RESET_SWDT_CRL,
XILPM_RESET_NAND,
XILPM_RESET_ADMA,
XILPM_RESET_GPIO,
XILPM_RESET_IOU_CC,
XILPM_RESET_TIMESTAMP,
XILPM_RESET_RPU_R50,
XILPM_RESET_RPU_R51,
XILPM_RESET_RPU_AMBA,
XILPM_RESET_OCM,
XILPM_RESET_RPU_PGE,
XILPM_RESET_USB0_CORE_RESET,
XILPM_RESET_USB0_CORE_RESET,
XILPM_RESET_USB0_HIBER_RESET,
XILPM_RESET_USB1_HIBER_RESET,
XILPM_RESET_USB0_APB,
XILPM_RESET_USB1_APB,
XILPM_RESET_IPI,
XILPM_RESET_APM_LPD,
XILPM_RESET_RTC,
XILPM_RESET_SYSMON,
XILPM_RESET_API_FM6,
XILPM_RESET_LP_D_SDWT,
XILPM_RESET_FPD,
XILPM_RESET_RPU_DBG1,
XILPM_RESET_RPU_DBG0,
XILPM_RESET_DBG_LP_D,
XILPM_RESET_DBG_FPD,
XILPM_RESET_APLL,
XILPM_RESET_DPLL,
XILPM_RESET_VPLL,
XILPM_RESET_IOPLL,
Appendix A: Argument Value Definitions

XILPM_RESET_RPLL,
XILPM_RESET_GPO3_PL_0,
XILPM_RESET_GPO3_PL_1,
XILPM_RESET_GPO3_PL_2,
XILPM_RESET_GPO3_PL_3,
XILPM_RESET_GPO3_PL_4,
XILPM_RESET_GPO3_PL_5,
XILPM_RESET_GPO3_PL_6,
XILPM_RESET_GPO3_PL_7,
XILPM_RESET_GPO3_PL_8,
XILPM_RESET_GPO3_PL_9,
XILPM_RESET_GPO3_PL_10,
XILPM_RESET_GPO3_PL_11,
XILPM_RESET_GPO3_PL_12,
XILPM_RESET_GPO3_PL_13,
XILPM_RESET_GPO3_PL_14,
XILPM_RESET_GPO3_PL_15,
XILPM_RESET_GPO3_PL_16,
XILPM_RESET_GPO3_PL_17,
XILPM_RESET_GPO3_PL_18,
XILPM_RESET_GPO3_PL_19,
XILPM_RESET_GPO3_PL_20,
XILPM_RESET_GPO3_PL_21,
XILPM_RESET_GPO3_PL_22,
XILPM_RESET_GPO3_PL_23,
XILPM_RESET_GPO3_PL_24,
XILPM_RESET_GPO3_PL_25,
XILPM_RESET_GPO3_PL_26,
XILPM_RESET_GPO3_PL_27,
XILPM_RESET_GPO3_PL_28,
XILPM_RESET_GPO3_PL_29,
XILPM_RESET_GPO3_PL_30,
XILPM_RESET_GPO3_PL_31,
Appendix A: Argument Value Definitions

**XPm_Notifier struct**

The XPm_Notifier struct is the structure to be passed in XPm_RegisterNotifier.

```c
typedef struct XPm_Notifier {
    void (*const callback)(XPm_Notifier* const notifier);
    enum XpmNodeId node;
    enum XpmNotifyEvent event;
    u32 flags;
    volatile u32 oppoint;
    volatile u32 received;
    XPm_Notifier* next;
} XPm_Notifier;
```

**Struct Members**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>callback</td>
<td>Custom callback handler to be called when the notification is received. The custom handler executes from the interrupt context; hence, it shall return quickly and must not block! (enables event-driven notifications),</td>
</tr>
<tr>
<td>node</td>
<td>Node argument (the node to receive notifications about).</td>
</tr>
<tr>
<td>event</td>
<td>Event argument (the event type to receive notifications about).</td>
</tr>
<tr>
<td>flags</td>
<td>Flags: Currently the flags only contain the wake option in bit0.</td>
</tr>
<tr>
<td></td>
<td>• flags = 1: wake up on event</td>
</tr>
<tr>
<td></td>
<td>• flags = 0: do not wake up (only notify if awake), no buffering or queueing will take place</td>
</tr>
<tr>
<td>oppoint</td>
<td>Operating point of node in question. Contains the value updated when the last event notification is received. User shall not modify this value while the notifier is registered.</td>
</tr>
<tr>
<td>received</td>
<td>How many times the notification has been received - to be used by application (enables polling). User shall not modify this value while the notifier is registered.</td>
</tr>
<tr>
<td>next</td>
<td>Pointer to next notifier in linked list. Must not be modified while the notifier is registered. User shall not ever modify this value.</td>
</tr>
</tbody>
</table>
Appendix A:  Argument Value Definitions

Xpm_NodeStatus struct

The Xpm_NodeStatus struct is used to pass node status information.

```c
typedef struct Xpm_NodeStatus {
  u32 status;
  u32 requirements;
  u32 usage;
} Xpm_NodeStatus;
```

Struct Members:

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>status</td>
<td>Node power state.</td>
</tr>
<tr>
<td>requirements</td>
<td>Current requirements asserted on the node (slaves only).</td>
</tr>
<tr>
<td>usage</td>
<td>Usage information (which master is currently using the slave).</td>
</tr>
<tr>
<td></td>
<td>This information is used for slave nodes only. It is encoded based on the</td>
</tr>
<tr>
<td></td>
<td>IPI bits for the masters.</td>
</tr>
<tr>
<td></td>
<td>If the respective bit is set, the corresponding master is currently using</td>
</tr>
<tr>
<td></td>
<td>the node.</td>
</tr>
</tbody>
</table>

Table A-4:  Struct Members
## Error Codes

The following table lists the error codes used in the Power Management Frameworks API.

### Table B-1: Power Management Error Codes

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>EEMI_SUCCESS</td>
<td>The API call was processed successfully.</td>
</tr>
<tr>
<td>EEMI_INVALID_PARAM</td>
<td>An argument is either out-of-range or its value is not admissible in the respective API call.</td>
</tr>
<tr>
<td>EEMI_NO_FEATURE</td>
<td>The requested feature is not available for the selected PM slave.</td>
</tr>
<tr>
<td>EEMI_CONFLICT</td>
<td>Conflicting requirements have been asserted when more than one processing cluster is using the same PM slave.</td>
</tr>
<tr>
<td>EEMI_DOUBLE_REQUEST</td>
<td>request_node: A processing cluster has already been assigned access to a PM slave and has issued a duplicate request for that PM slave.</td>
</tr>
<tr>
<td>EEMI_INVALID_NODE</td>
<td>The API function does not apply to the node passed as argument.</td>
</tr>
<tr>
<td>EEMI_NO_ACCESS</td>
<td>The processing cluster does not have access to the requested node or operation.</td>
</tr>
<tr>
<td>EEMI_ABORT_SUSPEND</td>
<td>The target processing cluster has aborted suspend.</td>
</tr>
</tbody>
</table>
Appendix C

Additional Resources and Legal Notices

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see Xilinx Support.

Solution Centers

See the Xilinx Solution Centers for support on devices, software tools, and intellectual property at all stages of the design cycle. Topics include design assistance, advisories, and troubleshooting tips.

Documentation Navigator and Design Hubs

Xilinx Documentation Navigator provides access to Xilinx documents, videos, and support resources, which you can filter and search to find information. To open the Xilinx Documentation Navigator (DocNav):

- From the Vivado IDE, select Help > Documentation and Tutorials.
- On Windows, select Start > All Programs > Xilinx Design Tools > DocNav.
- At the Linux command prompt, enter docnav.

Xilinx Design Hubs provide links to documentation organized by design tasks and other topics, which you can use to learn key concepts and address frequently asked questions. To access the Design Hubs:

- In the Xilinx Documentation Navigator, click the Design Hubs View tab.
- On the Xilinx website, see the Design Hubs page.

*Note:* For more information on Documentation Navigator, see the Documentation Navigator page on the Xilinx website.
References

2. Embedded Energy Management APIs (UG1200)
3. SMC Calling Convention – ARM DEN 0028A, 6/2013
5. Zynq UltraScale+ MPSoC Software Developer Guide (UG1137)
6. Zynq UltraScale+ MPSoC Register Reference (UG1087)
7. Xilinx Software Developer Kit Help (UG782)

Please Read: Important Legal Notices

The information disclosed to you hereunder (the “Materials”) is provided solely for the selection and use of Xilinx products. To the maximum extent permitted by applicable law: (1) Materials are made available “AS IS” and with all faults, Xilinx hereby DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third-party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of Xilinx’s limited warranty, please refer to Xilinx’s Terms of Sale which can be viewed at http://www.xilinx.com/legal.htm#tos; IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in such critical applications, please refer to Xilinx’s Terms of Sale which can be viewed at http://www.xilinx.com/legal.htm#tos.

AUTOMOTIVE APPLICATIONS DISCLAIMER

AUTOMOTIVE PRODUCTS (IDENTIFIED AS “XA” IN THE PART NUMBER) ARE NOT WARRANTED FOR USE IN THE DEPLOYMENT OF AIRBAGS OR FOR USE IN APPLICATIONS THAT AFFECT CONTROL OF A VEHICLE (“SAFETY APPLICATION”) UNLESS THERE IS A SAFETY CONCEPT OR REDUNDANCY FEATURE CONSISTENT WITH THE ISO 26262 AUTOMOTIVE SAFETY STANDARD (“SAFETY DESIGN”). CUSTOMER SHALL, PRIOR TO USING OR DISTRIBUTING ANY SYSTEMS THAT INCORPORATE PRODUCTS, THOROUGHLY TEST SUCH SYSTEMS FOR SAFETY PURPOSES. USE OF PRODUCTS IN A SAFETY APPLICATION WITHOUT A SAFETY DESIGN IS FULLY AT THE RISK OF CUSTOMER, SUBJECT ONLY TO APPLICABLE LAWS AND REGULATIONS GOVERNING LIMITATIONS ON PRODUCT LIABILITY.

© Copyright 2016 Xilinx, Inc. Xilinx, the Xilinx logo, Artix, JSE, Kintex, Spartan, Virtex, Vivado, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. All other trademarks are the property of their respective owners. AMBA, AMBA Designer, ARM, ARM1176JZ-S, CoreSight, Cortex, and PrimeCell are trademarks of ARM in the EU and other countries. MATLAB and Simulink are registered trademarks of The MathWorks, Inc. PCI, PCI Express, PCIe, and PCI-X are trademarks of PCI-SIG.