Zynq Power Management Framework User Guide

For Zynq UltraScale+ MPSoC Devices

UG1199 (v2.0) November 30, 2016

XILINX
ALL PROGRAMMABLE™

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
12/15/2016	2.0	Modified XPm_SystemShutdown in Chapter 2.
09/21/2016	1.0	Initial Xilinx release.

Table of Contents

Appendix B: Error Codes

Appendix C: Additional Resources and Legal Notices

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**

Chapter	Description	Target Audience
1	Introduction to the Power Management Framework.	All users.
2	Chapter 2, Xilpm API Functions.	Bare-metal, RTOS, and other software developers.
3	Chapter 3, Using the API for Power Management	Bare-metal RTOS and other software developers.
Appendix A	Appendix A, Argument Value Definitions lists the defined argument values for the power management controller, which is described	All users.
Appendix B	Appendix B, Error Codes, describes the PMF error codes.	All users.
Appendix C	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	All users.

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.

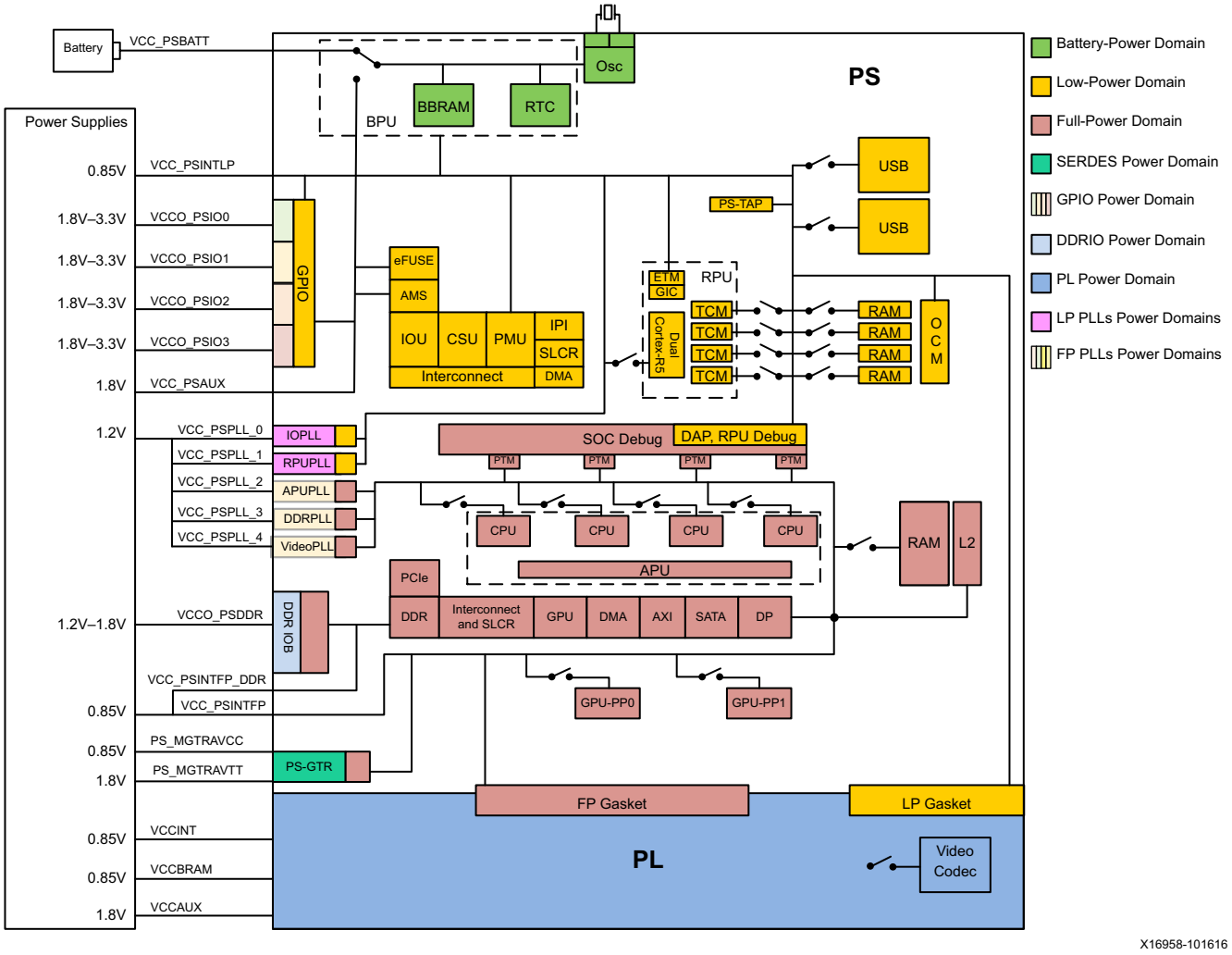

Figure 1-1: **Zynq UltraScale+ MPSoC Power Domain and Islands**

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.

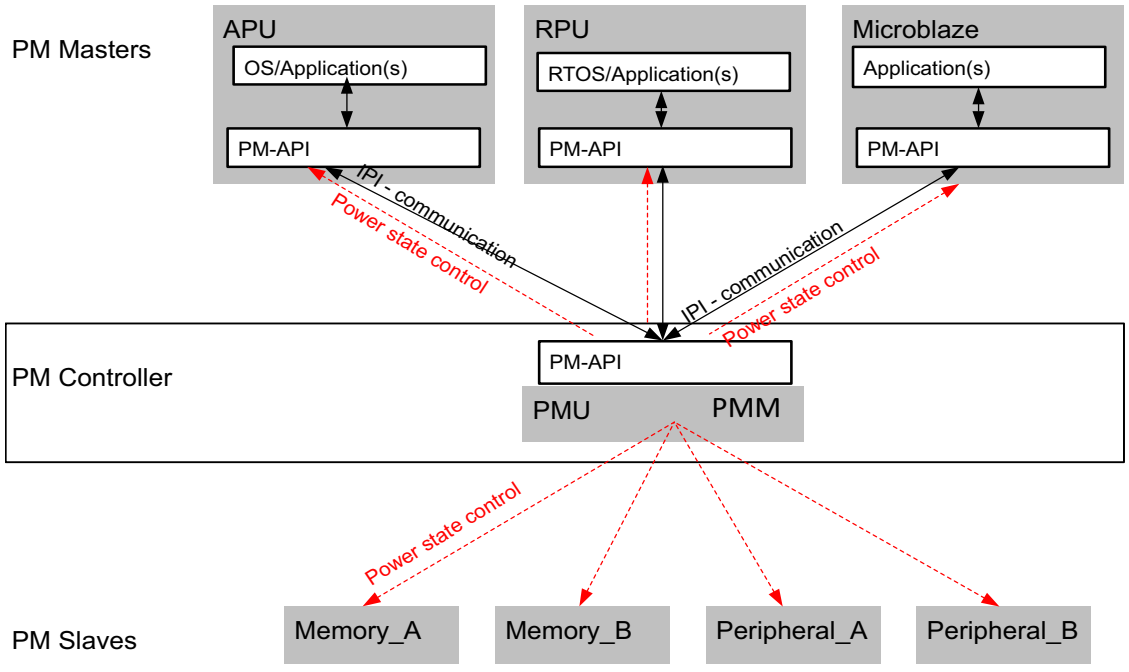

Figure 1-2: **API-Based Power Management Software Architecture -**

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.

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.

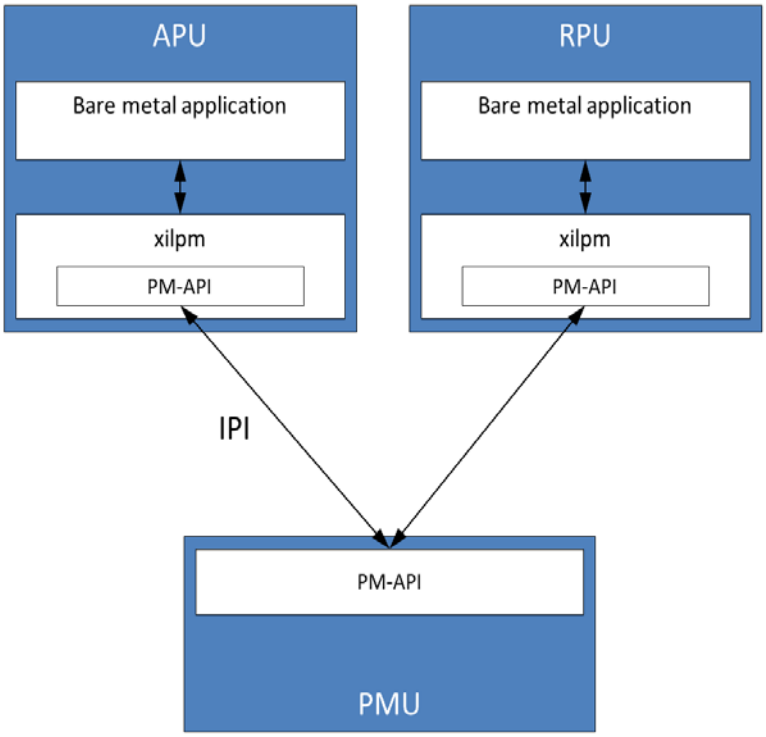

Figure 1-3: **API Layers Used with Bare-Metal Applications Only**

If the APU is running a complete software stack with an operating system, the `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.

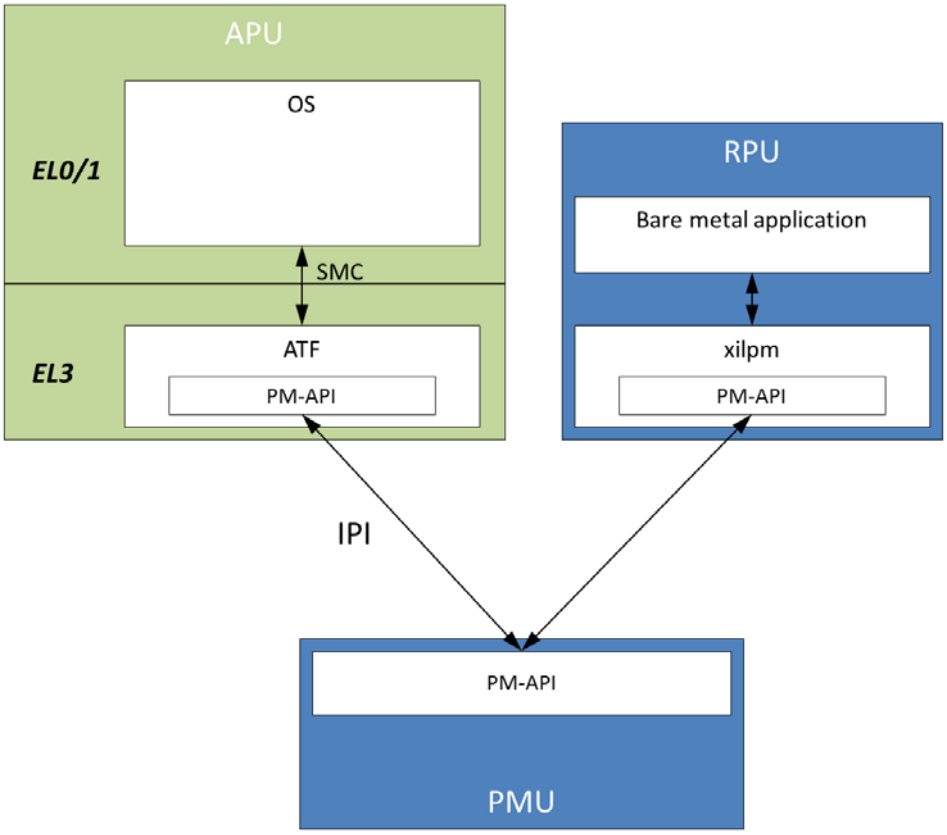

Figure 1-4: **PM Framework Layers Involved When Running a Full Software Stack on the APU**

Typical Power Management API Call Flow

Any entity involved in power management is referred to as a *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`.

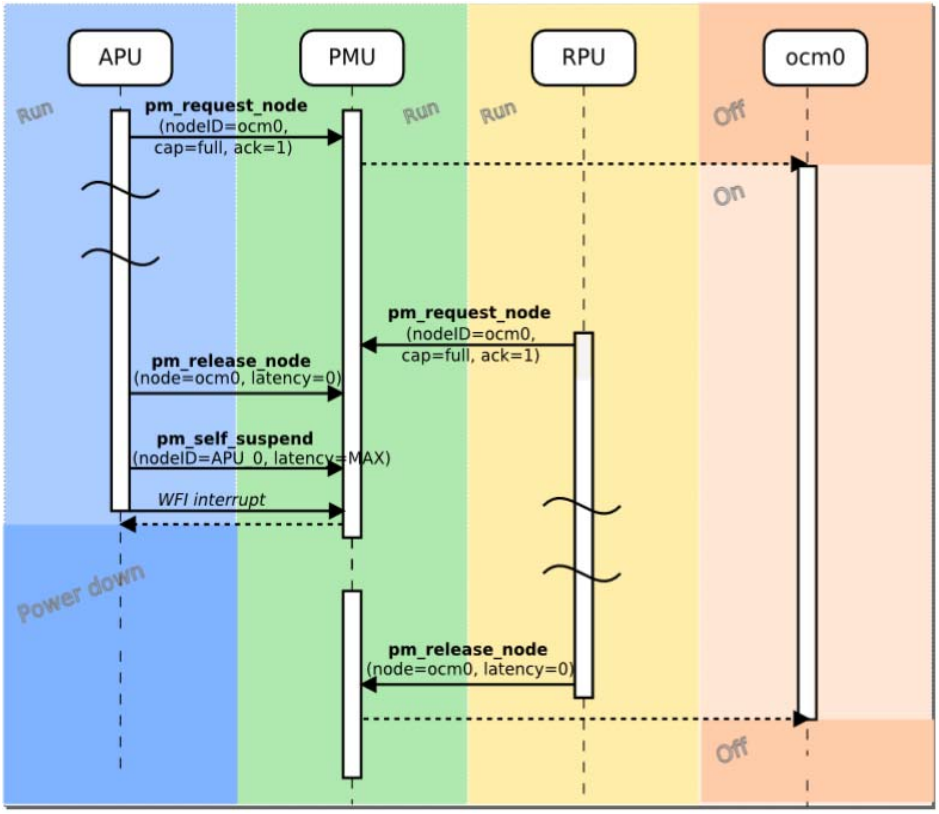

Figure 1-5: **PM Framework Call Sequence for APU and RPU Sharing an OCM Memory Bank**

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.

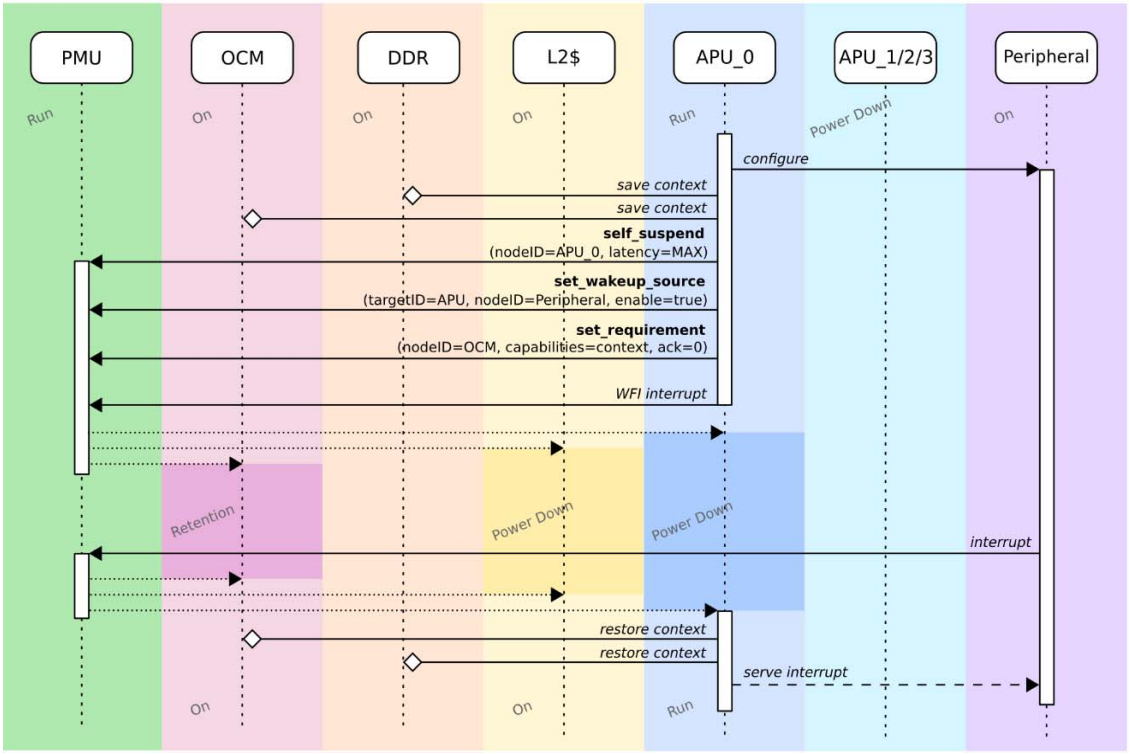

Figure 1-6: **APU Suspend and Resume Procedure**

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.

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**

Slave Nodes for the APU				
NODE_ADMA	NODE_DP	NODE_GPU_PP_0..1	NODE_OCM_0..3	NODE_SPI_0..1
NODE_AFI	NODE_DPLL	NODE_I2C_0..1	NODE_QSPI	NODE_TTC_0..3
NODE_APLL	NODE_ETH_0..3	NODE_IOPLL	NODE_RPLL	NODE_UART_0..1
NODE_CAN_0..1	NODE_GDMA	NODE_L2	NODE_SATA	NODE_USB_0..1
NODE_DDR	NODE_GPI0	NODE_NAND	NODE_SD_0..1	NODE_VPLL

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**

RPU Controlled Slave Nodes	
NODE_APLL	NODE_SATA
NODE_DDR	NODE_TCM_0_A/B
NODE_DPLL	NODE_TCM_1_A/B
NODE_IOPLL	NODE_TTC_0
NODE_OCM_0..3	NODE_USB_0..1
NODE_RPLL	NODE_VPLL

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**

Nodes Initialized by PMU					
NODE_ADMA	NODE_AFI	NODE_CAN_0	NODE_CAN_1	NODE_DP	NODE_GDMA
NODE_I2C_0	NODE_I2C_1	NODE_NAND	NODE_QSPI	NODE_SATA	NODE_SD_0
NODE_SPI_0	NODE_SPI_1	NODE_TTC_0	ODE_TTC_1	NODE_TTC_2	NODE_TTC_3
NODE_GPIO	NODE_SD_1	NODE_UART_0	NODE_UART_1	NODE_USB_0	NODE_USB_1

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 (PU)s.

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:

```
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.

XPm_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:

```
XPmStatus
XPm_SelfSuspend(const enum XPmNodeId 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:

```
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:

```
XPmStatus
XPm_RequestSuspend(const enum XPmNodeId node,
        const enum XPmRequestAck ack,
        const u32 latency,
        const u8 state);
```

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:

```
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:

```
XPmStatus
XPm_AbortSuspend(const enum XPmAbortReason reason);
```

Arguments:

- `reason`: Reason code why the suspend can not 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:

```
XPmStatus
XPm_RequestWakeUp(const enum XPmNodeId node,
      const u8 setAddress,
      const u64 address,
      const enum XPmRequestAck ack);
```

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:

```
XPmStatus
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:

```
void
XPm_InitSuspendCb(const enum XPmSuspendReason 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:

```
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:

```
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.

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:

```
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.

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:

```
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:

```
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:

```
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:

```
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**

Reset Values					
PM_RESET_PCIE_CFG	PM_RESET_ACPU0_PWRON	PM_RESET_CAN0	PM_RESET_USB1_CORERESET	PM_RESET_RPLL	PM_RESET_GPO3_PL_19
PM_RESET_PCIE_BRIDGE	PM_RESET_APU_L2	PM_RESET_CAN1	PM_RESET_USB0_HIBERRESET	PM_RESET_GPO3_PL_0	PM_RESET_GPO3_PL_20
PM_RESET_PCIE_CTRL	PM_RESET_ACPU3	PM_RESET_I2C0	PM_RESET_USB1_HIBERRESET	PM_RESET_GPO3_PL_1	PM_RESET_GPO3_PL_21
PM_RESET_DP	PM_RESET_ACPU2	PM_RESET_I2C1	PM_RESET_USB0_APB	PM_RESET_GPO3_PL_2	PM_RESET_GPO3_PL_22
PM_RESET_SWDT_CRF	PM_RESET_ACPU1	PM_RESET_TTC0	PM_RESET_USB1_APB	PM_RESET_GPO3_PL_3	PM_RESET_GPO3_PL_23
PM_RESET_AFI_FM5	PM_RESET_ACPU0	PM_RESET_TTC1	PM_RESET_IPI	PM_RESET_GPO3_PL_4	PM_RESET_GPO3_PL_24
PM_RESET_AFI_FM4	PM_RESET_DDR	PM_RESET_TTC2	PM_RESET_APM_LPD	PM_RESET_GPO3_PL_5	PM_RESET_GPO3_PL_25
PM_RESET_AFI_FM3	PM_RESET_APM_FPD	PM_RESET_TTC3	PM_RESET_RTC	PM_RESET_GPO3_PL_6	PM_RESET_GPO3_PL_26
PM_RESET_AFI_FM2	PM_RESET_SOFT	PM_RESET_SWDT_CRL	PM_RESET_SYSMON	PM_RESET_GPO3_PL_7	PM_RESET_GPO3_PL_27
PM_RESET_AFI_FM1	PM_RESET_GEM0	PM_RESET_NAND	PM_RESET_AFI_FM6	PM_RESET_GPO3_PL_8	PM_RESET_GPO3_PL_28
PM_RESET_AFI_FM0	PM_RESET_GEM1	PM_RESET_ADMA	PM_RESET_LPD_SWDT	PM_RESET_GPO3_PL_9	PM_RESET_GPO3_PL_29
PM_RESET_GDMA	PM_RESET_GEM2	PM_RESET_GPIO	PM_RESET_FPD	PM_RESET_GPO3_PL_10	PM_RESET_GPO3_PL_30
PM_RESET_GPU_PP1	PM_RESET_GEM3	PM_RESET_IOU_CC	PM_RESET_RPU_DBG1	PM_RESET_GPO3_PL_11	PM_RESET_GPO3_PL_31
PM_RESET_GPU_PP0	PM_RESET_QSPI	PM_RESET_TIMESTAMP	PM_RESET_RPU_DBG0	PM_RESET_GPO3_PL_12	PM_RESET_RPU_LS

Table 2-1: **XPm_ResetAssert Reset Values**

Reset Values					
PM_RESET_GPU	PM_RESET_UART0	PM_RESET_RPU_R50	PM_RESET_DBG_LPD	PM_RESET_GPO3_PL_13	PM_RESET_PS_ONLY
PM_RESET_GT	PM_RESET_UART1	PM_RESET_RPU_R51	PM_RESET_DBG_FPD	PM_RESET_GPO3_PL_14	PM_RESET_PL
PM_RESET_SATA	PM_RESET_SPI0	PM_RESET_RPU_AMBA	PM_RESET_APLL	PM_RESET_GPO3_PL_15	
PM_RESET_ACPU3_PWRON	PM_RESET_SPI1	PM_RESET_OCM	PM_RESET_DPLL	PM_RESET_GPO3_PL_16	
PM_RESET_ACPU2_PWRON	PM_RESET_SDIO0	PM_RESET_RPU_PGE	PM_RESET_VPLL	PM_RESET_GPO3_PL_17	
PM_RESET_ACPU1_PWRON	PM_RESET_SDIO1	PM_RESET_USB0_CORERESET	PM_RESET_IOPLL	PM_RESET_GPO3_PL_18	

- `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:

```
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:

```
XPm_GetNodeStatus(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:

Table 2-2: **Notifier Data**

Argument	Notifier Data
node_id	ID of the node.
event_id	ID of the event in question, "-1" denotes all events: • EVENT_STATE_CHANGE: This event is triggered when a component changes its power state. • EVENT_ZERO_USERS: This event is triggered when a slave node's last user calls XPm_ReleaseNode. • EVENT_ERROR_CONDITION: This event is triggered when an error occurs (currently never triggered).
wake	• true: Wake up on event. • false: do not wake up (only notify if awake), no buffering/queuing.
callback	Pointer to the custom callback function to call when the notification is available. The callback executes from interrupt context, so the user must take special care when implementing the callback. Callback is optional, and can be set to `NULL`.
received	Variable indicating how many times the notification was received because the notifier is registered.

Example:

```
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:

```
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 µs 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:

```
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:

```
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:

```
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:

```
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:

```
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:

```
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.

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.

Example Code for Suspending an APU/RPU

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

```
/* 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:

```
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.

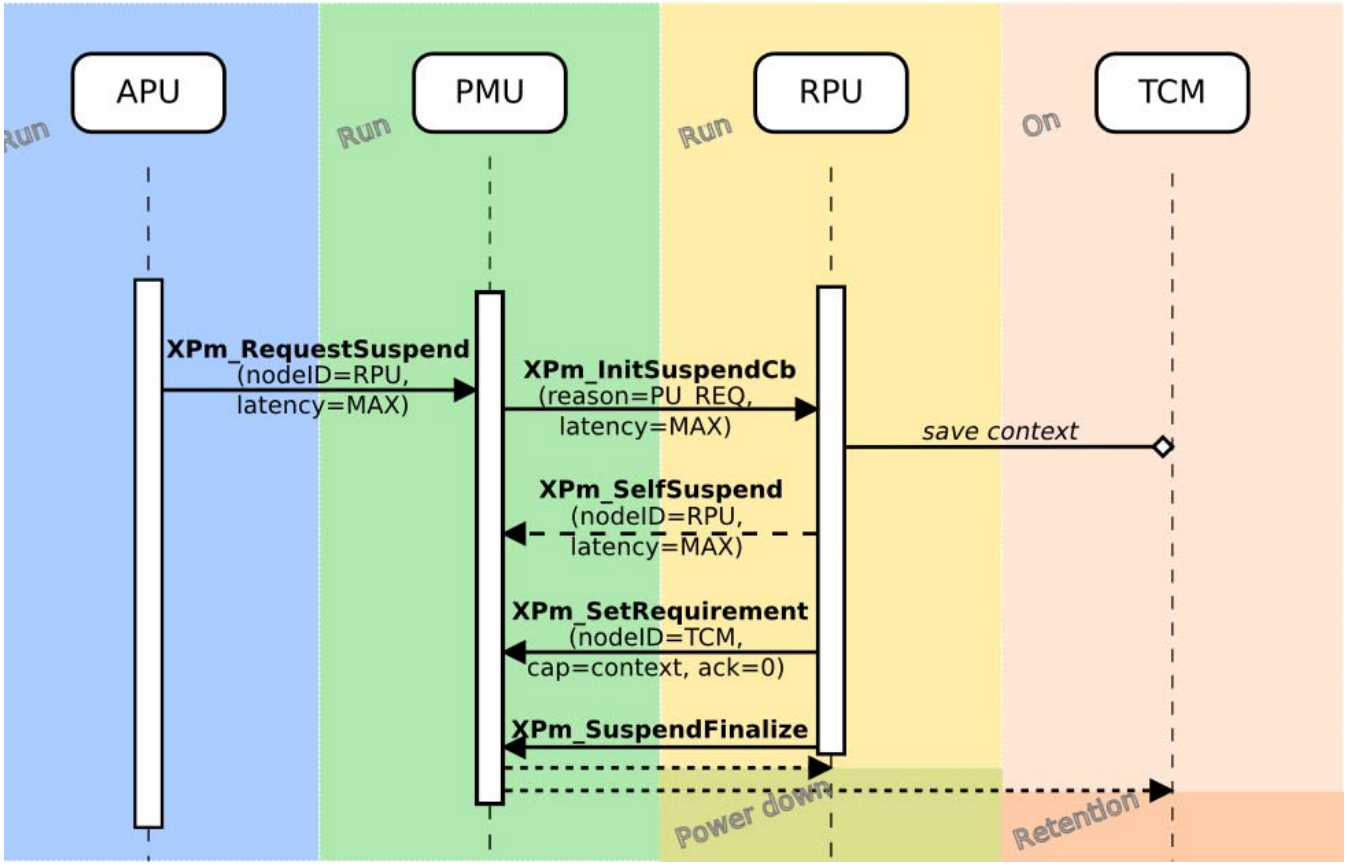

Figure 3-1: **APU initiating suspend for the RPU by calling XPm_RequestSuspend**

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:

```
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.

```
enum XPmNodeId {
```

Table A-1: **XPmNodeIds**

Node IDs for Zynq UltraScale+ MPSoC Devices		
NODE_UNKNOWN		0U
NODE_APU	APU Controller	1U
NODE_APU_0	APU Controller 0	2U
NODE_APU_1	APU Controller 1	3U
NODE_APU_2	APU Controller 2	4U
NODE_APU_3	APU Controller 3	5U
NODE_RPU	RPU Controller	6U
NODE_RPU_0	RPU Controller 0	7U
NODE_RPU_1	RPU Controller 1	8U
NODE_PL	PL Controller	9U
NODE_FPD	FPD Controller	10U
NODE_OCM_BANK_0	OCM Memory Tile 0	11U
NODE_OCM_BANK_1	OCM Memory Tile 1	12U
NODE_OCM_BANK_2	OCM Memory Tile 2	13U
NODE_OCM_BANK_3	OCM Memory Tile 3	14U
NODE_TCM_0_A	Tightly coupled memory (0A)	15U
NODE_TCM_0_B	Tightly coupled memory (0B)	16U

Table A-1: **XPmNodeIds *(Cont'd)***

Node IDs for Zynq UltraScale+ MPSoC Devices		
NODE_TCM_1_A	Tightly coupled memory (1A)	17U
NODE_TCM_1_B	Tightly coupled memory (1B)	18U
NODE_L2	L2 Cache system	19U
NODE_GPU_PP_0	Graphics Processing Unit 0	20U
NODE_GPU_PP_1	Graphics Processing Unit 1	21U
NODE_USB_0	USB Controller 0	22U
NODE_USB_1	USB Controller 1	23U
NODE_TTC_0	Triple-timer Counter 0	24U
NODE_TTC_1	Triple-timer Counter 1	25U
NODE_TTC_2	Triple-timer Counter 2	26U
NODE_TTC_3	Triple-timer Counter 3	27U
NODE_SATA	SATA Controller	28U
NODE_ETH_0	Gigabit Ethernet Controller 0	29U
NODE_ETH_1	Gigabit Ethernet Controller 1	30U
NODE_ETH_2	Gigabit Ethernet Controller 2	31U
NODE_ETH_3	Gigabit Ethernet Controller 3	32U
NODE_UART_0	UART Controller 0	33U
NODE_UART_1	UART Controller 1	34U
NODE_SPI_0	SPI Controller 0	35U
NODE_SPI_1	SPI Controller 1	36U
NODE_I2C_0	SPI Controller 2	37U
NODE_I2C_1	SPI Controller 3	38U
NODE_SD_0	SD/SDIO Controller 0	39U
NODE_SD_1	SD/SDIO Controller 1	40U
NODE_DP	DisplayPort Controller	41U
NODE_GDMA	FPD DMA Controller	42U
NODE_ADMA	APU DMA	43U
NODE_NAND	NAND Controller	44U
NODE_QSPI	QSPI Controller	45U
NODE_GPIO	GPIO Controller	46U
NODE_CAN_0	CAN Controller 0	47U
NODE_CAN_1	CAN Controller 1	48U
NODE_AFI	AFI Block	49U
NODE_APLL	APU PLL	50U
NODE_VPLL	Video PLL	51U

Table A-1: **XPmNodeIds *(Cont'd)***

Node IDs for Zynq UltraScale+ MPSoC Devices		
NODE_DPLL	DDR Controller PLL	52U
NODE_RPLL	RPU PLL	53U
NODE_IOPLL	Peripheral I/O PLL	54U
NODE_DDR	DDR Controller	55U
NODE_IPI_APU	IPI APU Controller	56U
NODE_IPI_RPU_0	IPI APU Controller 0	57U
NODE_GPU	Graphics Processing Unit Controller	58U
NODE_PCIE	PCIE Controller	59U

Acknowledge Request Types: XPmRequestAck

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

Abort Reasons: XPmAbortReason

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

Suspend Reasons

```
enum XPmSuspendReason {
  SUSPEND_REASON_PU_REQ = 201,
  SUSPEND_REASON_ALERT,
  SUSPEND_REASON_SYS_SHUTDOWN,
};
```

Operating Characteristic Types: XPmOpCharType

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

Notify Event Types: XPmNotifyEvent

```
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.

```
enum XPmReset {
```

Table A-2: **Reset Line IDs**

Reset Line IDs				
XILPM_RESET_PCIE_CFG = 1000	XILPM_RESET_DDR	XILPM_RESET_IOU_CC	XILPM_RESET_VPLL	XILPM_RESET_GPO3_PL_22
XILPM_RESET_PCIE_BRIDGE	XILPM_RESET_APM_FPD	XILPM_RESET_TIMESTAMP	XILPM_RESET_IOPLL	XILPM_RESET_GPO3_PL_23
XILPM_RESET_PCIE_CTRL	XILPM_RESET_SOFT	XILPM_RESET_RPU_R50	XILPM_RESET_RPLL	XILPM_RESET_GPO3_PL_24
XILPM_RESET_DP	XILPM_RESET_GEM0	XILPM_RESET_RPU_R51	XILPM_RESET_RPLL	XILPM_RESET_GPO3_PL_25
XILPM_RESET_SWDT_CRF	XILPM_RESET_GEM1	XILPM_RESET_RPU_AMBA	XILPM_RESET_GPO3_PL_0	XILPM_RESET_GPO3_PL_26
XILPM_RESET_AFI_FM5	XILPM_RESET_GEM2	XILPM_RESET_OCM	XILPM_RESET_GPO3_PL_1	XILPM_RESET_GPO3_PL_27
XILPM_RESET_AFI_FM4	XILPM_RESET_GEM3	XILPM_RESET_RPU_PGE	XILPM_RESET_GPO3_PL_2	XILPM_RESET_GPO3_PL_28

Table A-2: **Reset Line IDs *(Cont'd)***

Reset Line IDs				
XILPM_RESET_AFI_FM3	XILPM_RESET_QSPI	XILPM_RESET_USB0_CORERESET	XILPM_RESET_GPO3_PL_3	XILPM_RESET_GPO3_PL_29
XILPM_RESET_AFI_FM2	XILPM_RESET_UART0	XILPM_RESET_USB1_CORERESET	XILPM_RESET_GPO3_PL_4	XILPM_RESET_GPO3_PL_30
XILPM_RESET_AFI_FM1	XILPM_RESET_UART1	XILPM_RESET_USB0_HIBERRESET	XILPM_RESET_GPO3_PL_5	XILPM_RESET_GPO3_PL_31
XILPM_RESET_AFI_FM0	XILPM_RESET_SPI0	XILPM_RESET_USB1_HIBERRESET	XILPM_RESET_GPO3_PL_6	XILPM_RESET_ACPU2
XILPM_RESET_GDMA	XILPM_RESET_SPI1	XILPM_RESET_USB0_APB	XILPM_RESET_GPO3_PL_7	XILPM_RESET_ACPU1
XILPM_RESET_GPU_PP1	XILPM_RESET_SDIO0	XILPM_RESET_USB1_APB	XILPM_RESET_GPO3_PL_8	XILPM_RESET_ACPU0
XILPM_RESET_GPU_PP0	XILPM_RESET_SDIO1	XILPM_RESET_IPI	XILPM_RESET_GPO3_PL_9	XILPM_RESET_NAND
XILPM_RESET_GPU	XILPM_RESET_CAN0	XILPM_RESET_APM_LPD	XILPM_RESET_GPO3_PL_10	XILPM_RESET_ADMA
XILPM_RESET_GT	XILPM_RESET_CAN1	XILPM_RESET_RTC	XILPM_RESET_GPO3_PL_11	XILPM_RESET_GPIO
XILPM_RESET_SATA	XILPM_RESET_I2C0	XILPM_RESET_SYSMON	XILPM_RESET_GPO3_PL_12	XILPM_RESET_DBG_FPD
XILPM_RESET_ACPU3_PWRON	XILPM_RESET_I2C1	XILPM_RESET_AFI_FM6	XILPM_RESET_GPO3_PL_13	XILPM_RESET_APLL
XILPM_RESET_ACPU2_PWRON	XILPM_RESET_TTC0	XILPM_RESET_LPD_SWDT	XILPM_RESET_GPO3_PL_14	XILPM_RESET_DPLL
XILPM_RESET_ACPU1_PWRON	XILPM_RESET_TTC1	XILPM_RESET_FPD	XILPM_RESET_GPO3_PL_15	XILPM_RESET_GPO3_PL_19
XILPM_RESET_ACPU0_PWRON	XILPM_RESET_TTC2	XILPM_RESET_RPU_DBG1	XILPM_RESET_GPO3_PL_16	XILPM_RESET_GPO3_PL_20
XILPM_RESET_APU_L2	XILPM_RESET_TTC3	XILPM_RESET_RPU_DBG0	XILPM_RESET_GPO3_PL_17	XILPM_RESET_GPO3_PL_21
XILPM_RESET_ACPU3	XILPM_RESET_SWDT_CRL	XILPM_RESET_DBG_LPD	XILPM_RESET_GPO3_PL_18	

Notify Event Types: XPmNotifyEvent

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

Reset Line IDs

```
enum XPmReset {
XILPM_RESET_PCIE_CFG = 1000,
XILPM_RESET_PCIE_BRIDGE,
XILPM_RESET_PCIE_CTRL,
XILPM_RESET_DP,
XILPM_RESET_SWDT_CRF,
XILPM_RESET_AFI_FM5,
XILPM_RESET_AFI_FM4,
XILPM_RESET_AFI_FM3,
XILPM_RESET_AFI_FM2,
XILPM_RESET_AFI_FM1,
XILPM_RESET_AFI_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,
```

```
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_CORERESET,
XILPM_RESET_USB1_CORERESET,
XILPM_RESET_USB0_HIBERRESET,
XILPM_RESET_USB1_HIBERRESET,
XILPM_RESET_USB0_APB,
XILPM_RESET_USB1_APB,
XILPM_RESET_IPI,
XILPM_RESET_APM_LPD,
XILPM_RESET_RTC,
XILPM_RESET_SYSMON,
XILPM_RESET_AFI_FM6,
XILPM_RESET_LPD_SWDT,
XILPM_RESET_FPD,
XILPM_RESET_RPU_DBG1,
XILPM_RESET_RPU_DBG0,
XILPM_RESET_DBG_LPD,
XILPM_RESET_DBG_FPD,
XILPM_RESET_APLL,
XILPM_RESET_DPLL,
XILPM_RESET_VPLL,
XILPM_RESET_IOPLL,
```

```
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,
};
```

XPm_Notifier struct

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

```
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 A-3: **Struct Members**

Name	Description
`callback`	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),
`node`	Node argument (the node to receive notifications about).
`event`	Event argument (the event type to receive notifications about).
`flags`	Flags: Currently the flags only contain the wake option in bit0. • flags = 1: wake up on event • flags = 0: do not wake up (only notify if awake), no buffering or queueing will take place
`oppoint`	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.
`received`	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.
`next`	Pointer to next notifier in linked list. Must not be modified while the notifier is registered. User shall not ever modify this value.

XPm_NodeStatus struct

The XPm_NodeStatus struct is used to pass node status information.

```
typedef struct XPm_NodeStatus {
  u32 status;
  u32 requirements;
  u32 usage;
} XPm_NodeStatus;
```

Struct Members:

Table A-4: **Struct Members**

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

Error Codes

Error Codes

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

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

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

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

1. *Zynq UltraScale+ MPSoC Technical Reference Manual* (UG1085)
2. *Embedded Energy Management APIs* (UG1200)
3. *SMC Calling Convention* – ARM DEN 0028A, 6/2013
4. *Power State Coordination Interface* – ARM DEN 0022B.b, 6/25/2013
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