Xilinx Standalone Library Documentation

XilPM Library v3.1

UG1225 (v2020.1) June 3, 2020

XILINX®

Table of Contents

XilPM Zynq UltraScale+ MPSoC APIs

Xilinx Power Management (XilPM) provides Embedded Energy Management Interface (EEMI) APIs for power management on Zynq UltraScale+ MPSoC. For more details about EEMI, see the Embedded Energy Management Interface (EEMI) API User Guide (UG1200).

Table 1: **Quick Function Reference**

Type	Name	Arguments
XStatus	XPm_InitXilpm	XIpiPsu * IpiInst
enum XPmBootStatus	XPm_GetBootStatus	void
void	XPm_SuspendFinalize	void
XStatus	pm_ipi_send	struct XPm_Master *const master u32 payload
XStatus	pm_ipi_buff_read32	struct XPm_Master *const master u32 * value1 u32 * value2 u32 * value3
XStatus	XPm_SelfSuspend	const enum XPmNodeId nid const u32 latency const u8 state const u64 address
XStatus	XPm_SetConfiguration	const u32 address
XStatus	XPm_InitFinalize	void
XStatus	XPm_RequestSuspend	const enum XPmNodeId target const enum XPmRequestAck ack const u32 latency const u8 state

Table 1: **Quick Function Reference** *(cont'd)*

Type	Name	Arguments
XStatus	XPm_RequestWakeUp	const enum XPmNodeId target const bool setAddress const u64 address const enum XPmRequestAck ack
XStatus	XPm_ForcePowerDown	const enum XPmNodeId target const enum XPmRequestAck ack
XStatus	XPm_AbortSuspend	const enum XPmAbortReason reason
XStatus	XPm_SetWakeUpSource	const enum XPmNodeId target const enum XPmNodeId wkup_node const u8 enable
XStatus	XPm_SystemShutdown	restart
XStatus	XPm_RequestNode	const enum XPmNodeId node const u32 capabilities const u32 qos const enum XPmRequestAck ack
XStatus	XPm_SetRequirement	const enum XPmNodeId nid const u32 capabilities const u32 qos const enum XPmRequestAck ack
XStatus	XPm_ReleaseNode	const enum XPmNodeId node
XStatus	XPm_SetMaxLatency	const enum XPmNodeId node const u32 latency
void	XPm_InitSuspendCb	const enum XPmSuspendReason reason const u32 latency const u32 state const u32 timeout
void	XPm_AcknowledgeCb	const enum XPmNodeId node const XStatus status const u32 oppoint

Table 1: **Quick Function Reference** *(cont'd)*

Type	Name	Arguments
void	XPm_NotifyCb	const enum XPmNodeId node const enum XPmNotifyEvent event const u32 oppoint
XStatus	XPm_GetApiVersion	u32 * version
XStatus	XPm_GetNodeStatus	const enum XPmNodeId node XPm_NodeStatus *const nodestatus
XStatus	XPm_GetOpCharacteristic	const enum XPmNodeId node const enum XPmOpCharType type u32 *const result
XStatus	XPm_ResetAssert	const enum XPmReset reset assert
XStatus	XPm_ResetGetStatus	const enum XPmReset reset u32 * status
XStatus	XPm_RegisterNotifier	XPm_Notifier *const notifier
XStatus	XPm_UnregisterNotifier	XPm_Notifier *const notifier
XStatus	XPm_MmioWrite	const u32 address const u32 mask const u32 value
XStatus	XPm_MmioRead	const u32 address u32 *const value
XStatus	XPm_ClockEnable	const enum XPmClock clock
XStatus	XPm_ClockDisable	const enum XPmClock clock
XStatus	XPm_ClockGetStatus	const enum XPmClock clock u32 *const status
XStatus	XPm_ClockSetOneDivider	const enum XPmClock clock const u32 divider const u32 divId

Table 1: **Quick Function Reference** *(cont'd)*

Type	Name	Arguments
XStatus	XPm_ClockSetDivider	const enum XPmClock clock const u32 divider
XStatus	XPm_ClockGetOneDivider	const enum XPmClock clock u32 *const divider
XStatus	XPm_ClockGetDivider	const enum XPmClock clock u32 *const divider
XStatus	XPm_ClockSetParent	const enum XPmClock clock const enum XPmClock parent
XStatus	XPm_ClockGetParent	const enum XPmClock clock enum XPmClock *const parent
XStatus	XPm_ClockSetRate	const enum XPmClock clock const u32 rate
XStatus	XPm_ClockGetRate	const enum XPmClock clock u32 *const rate
XStatus	XPm_PllSetParameter	const enum XPmNodeId node const enum XPmPllParam parameter const u32 value
XStatus	XPm_PllGetParameter	const enum XPmNodeId node const enum XPmPllParam parameter u32 *const value
XStatus	XPm_PllSetMode	const enum XPmNodeId node const enum XPmPllMode mode
XStatus	XPm_PllGetMode	const enum XPmNodeId node enum XPmPllMode *const mode
XStatus	XPm_PinCtrlAction	const u32 pin
XStatus	XPm_PinCtrlRequest	const u32 pin
XStatus	XPm_PinCtrlRelease	const u32 pin

Table 1: **Quick Function Reference** *(cont'd)*

Type	Name	Arguments
XStatus	XPm_PinCtrlSetFunction	const u32 pin const enum XPmPinFn fn
XStatus	XPm_PinCtrlGetFunction	const u32 pin enum XPmPinFn *const fn
XStatus	XPm_PinCtrlSetParameter	const u32 pin const enum XPmPinParam param const u32 value
XStatus	XPm_PinCtrlGetParameter	const u32 pin const enum XPmPinParam param u32 *const value

Functions

XPm_InitXilpm

Initialize xilpm library.

Note: None

Prototype

```
XStatus XPm_InitXilpm(XIpiPsu *IpiInst);
```

Parameters

The following table lists the `XPm_InitXilpm` function arguments.

Table 2: **XPm_InitXilpm Arguments**

Type	Name	Description
XIpiPsu *	IpiInst	Pointer to IPI driver instance

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

XPm_GetBootStatus

This Function returns information about the boot reason. If the boot is not a system startup but a resume, power down request bitfield for this processor will be cleared.

***Note*:** None

Prototype

```
enum
                XPmBootStatus
              XPm_GetBootStatus(void);
```

Returns

Returns processor boot status

- PM_RESUME : If the boot reason is because of system resume.
- PM_INITIAL_BOOT : If this boot is the initial system startup.

XPm_SuspendFinalize

This Function waits for PMU to finish all previous API requests sent by the PU and performs client specific actions to finish suspend procedure (e.g. execution of wfi instruction on A53 and R5 processors).

***Note*:** This function should not return if the suspend procedure is successful.

Prototype

```
void XPm_SuspendFinalize(void);
```

Returns

pm_ipi_send

Sends IPI request to the PMU.

***Note*:** None

Prototype

```
XStatus pm_ipi_send(struct XPm_Master *const master, u32
payload[PAYLOAD_ARG_CNT]);
```

Parameters

The following table lists the `pm_ipi_send` function arguments.

Table 3: **pm_ipi_send Arguments**

Type	Name	Description
struct XPm_Master *const	master	Pointer to the master who is initiating request
u32	payload	API id and call arguments to be written in IPI buffer

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

pm_ipi_buff_read32

Reads IPI response after PMU has handled interrupt.

Note: None

Prototype

```
XStatus pm_ipi_buff_read32(struct XPm_Master *const master, u32 *value1,
u32 *value2, u32 *value3);
```

Parameters

The following table lists the `pm_ipi_buff_read32` function arguments.

Table 4: **pm_ipi_buff_read32 Arguments**

Type	Name	Description
struct XPm_Master *const	master	Pointer to the master who is waiting and reading response
u32 *	value1	Used to return value from 2nd IPI buffer element (optional)
u32 *	value2	Used to return value from 3rd IPI buffer element (optional)
u32 *	value3	Used to return value from 4th IPI buffer element (optional)

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

XPm_SelfSuspend

This function is used by a CPU to declare that it is about to suspend itself. After the PMU processes this call it will wait for the requesting CPU to complete the suspend procedure and become ready to be put into a sleep state.

***Note*:** This is a blocking call, it will return only once PMU has responded

Prototype

```
XStatus XPm_SelfSuspend(const enum XPmNodeId nid, const u32 latency, const
u8 state, const u64 address);
```

Parameters

The following table lists the `XPm_SelfSuspend` function arguments.

Table 5: **XPm_SelfSuspend Arguments**

Type	Name	Description
const enum `XPmNodeId`	nid	Node ID of the CPU node to be suspended.
const u32	latency	Maximum wake-up latency requirement in us(microsecs)
const u8	state	Instead of specifying a maximum latency, a CPU can also explicitly request a certain power state.
const u64	address	Address from which to resume when woken up.

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

XPm_SetConfiguration

This function is called to configure the power management framework. The call triggers power management controller to load the configuration object and configure itself according to the content of the object.

***Note*:** The provided address must be in 32-bit address space which is accessible by the PMU.

Prototype

```
XStatus XPm_SetConfiguration(const u32 address);
```

Parameters

The following table lists the `XPm_SetConfiguration` function arguments.

Table 6: **XPm_SetConfiguration Arguments**

Type	Name	Description
const u32	address	Start address of the configuration object

Returns

XST_SUCCESS if successful, otherwise an error code

XPm_InitFinalize

This function is called to notify the power management controller about the completed power management initialization.

***Note*:** It is assumed that all used nodes are requested when this call is made. The power management controller may power down the nodes which are not requested after this call is processed.

Prototype

```
XStatus XPm_InitFinalize(void);
```

Returns

XST_SUCCESS if successful, otherwise an error code

XPm_RequestSuspend

This function is used by a PU to request suspend of another PU. This call triggers the power management controller to notify the PU identified by 'nodeID' that a suspend has been requested. This will allow said PU to gracefully suspend itself by calling XPm_SelfSuspend for each of its CPU nodes, or else call XPm_AbortSuspend with its PU node as argument and specify the reason.

***Note*:** If 'ack' is set to PM_ACK_NON_BLOCKING, the requesting PU will be notified upon completion of suspend or if an error occurred, such as an abort. REQUEST_ACK_BLOCKING is not supported for this command.

Prototype

```
XStatus XPm_RequestSuspend(const enum XPmNodeId target, const enum
XPmRequestAck ack, const u32 latency, const u8 state);
```

Parameters

The following table lists the `XPm_RequestSuspend` function arguments.

Table 7: **XPm_RequestSuspend Arguments**

Type	Name	Description
const enum XPmNodeId	target	Node ID of the PU node to be suspended
const enum XPmRequestAck	ack	Requested acknowledge type
const u32	latency	Maximum wake-up latency requirement in us(micro sec)
const u8	state	Instead of specifying a maximum latency, a PU can also explicitly request a certain power state.

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

XPm_RequestWakeUp

This function can be used to request power up of a CPU node within the same PU, or to power up another PU.

***Note*:** If acknowledge is requested, the calling PU will be notified by the power management controller once the wake-up is completed.

Prototype

```
XStatus XPm_RequestWakeUp(const enum XPmNodeId target, const bool
setAddress, const u64 address, const enum XPmRequestAck ack);
```

Parameters

The following table lists the `XPm_RequestWakeUp` function arguments.

Table 8: **XPm_RequestWakeUp Arguments**

Type	Name	Description
const enum XPmNodeId	target	Node ID of the CPU or PU to be powered/woken up.
const bool	setAddress	Specifies whether the start address argument is being passed. • 0 : do not set start address • 1 : set start address
const u64	address	Address from which to resume when woken up. Will only be used if set_address is 1.
const enum XPmRequestAck	ack	Requested acknowledge type

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

XPm_ForcePowerDown

One PU can request a forced poweroff 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 will be automatically released.

***Note*:** Force power down may not be requested by a PU for itself.

Prototype

```
XStatus XPm_ForcePowerDown(const enum XPmNodeId target, const enum
XPmRequestAck ack);
```

Parameters

The following table lists the `XPm_ForcePowerDown` function arguments.

Table 9: **XPm_ForcePowerDown Arguments**

Type	Name	Description
const enum `XPmNodeId`	target	Node ID of the PU node or power island/domain to be powered down.
const enum `XPmRequestAck`	ack	Requested acknowledge type

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

XPm_AbortSuspend

This function is called by a CPU after a XPm_SelfSuspend call to notify the power management controller that CPU has aborted suspend or in response to an init suspend request when the PU refuses to suspend.

***Note*:** Calling PU expects the PMU to abort the initiated suspend procedure. This is a non-blocking call without any acknowledge.

Prototype

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

Parameters

The following table lists the `XPm_AbortSuspend` function arguments.

Table 10: **XPm_AbortSuspend Arguments**

Type	Name	Description
const enum XPmAbortReason	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 powerdown supported • ABORT_REASON_UNKNOWN : unknown error during suspend procedure

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

XPm_SetWakeUpSource

This function is called by a PU to add or remove a wake-up source prior to going to suspend. The list of wake sources for a PU is automatically cleared whenever the PU is woken up or when one of its CPUs aborts the suspend procedure.

***Note*:** Declaring a node as a wakeup source will ensure that the node will not be powered off. It also will cause the PMU to configure the GIC Proxy accordingly if the FPD is powered off.

Prototype

```
XStatus XPm_SetWakeUpSource(const enum XPmNodeId target, const enum
XPmNodeId wkup_node, const u8 enable);
```

Parameters

The following table lists the `XPm_SetWakeUpSource` function arguments.

Table 11: **XPm_SetWakeUpSource Arguments**

Type	Name	Description
const enum XPmNodeId	target	Node ID of the target to be woken up.
const enum XPmNodeId	wkup_node	Node ID of the wakeup device.
const u8	enable	Enable flag: • 1 : the wakeup source is added to the list • 0 : the wakeup source is removed from the list

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

XPm_SystemShutdown

This function can be used by a privileged PU to shut down or restart the complete device.

***Note*:** In either case the PMU will call XPm_InitSuspendCb for each of the other PUs, allowing them to gracefully shut down. If a PU is asleep it will be woken up by the PMU. The PU making the XPm_SystemShutdown should perform its own suspend procedure after calling this API. It will not receive an init suspend callback.

Prototype

```
XStatus XPm_SystemShutdown(u32 type, u32 subtype);
```

Parameters

The following table lists the `XPm_SystemShutdown` function arguments.

Table 12: **XPm_SystemShutdown Arguments**

Type	Name	Description
Commented parameter restart does not exist in function XPm_SystemShutdown.	restart	Should the system be restarted automatically? • PM_SHUTDOWN : no restart requested, system will be powered off permanently • PM_RESTART : restart is requested, system will go through a full reset

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

XPm_RequestNode

Used to request the usage of a PM-slave. 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 PMU will enable access to the memory mapped region containing the control registers of that device. For devices that can only be serving a single PU, any other privileged PU will now be blocked from accessing this device until the node is released.

***Note*:** None

Prototype

```
XStatus XPm_RequestNode(const enum XPmNodeId node, const u32 capabilities,
const u32 qos, const enum XPmRequestAck ack);
```

Parameters

The following table lists the `XPm_RequestNode` function arguments.

Table 13: **XPm_RequestNode Arguments**

Type	Name	Description
const enum `XPmNodeId`	node	Node ID of the PM slave requested
const u32	capabilities	Slave-specific capabilities required, can be combined • PM_CAP_ACCESS : full access / functionality • PM_CAP_CONTEXT : preserve context • PM_CAP_WAKEUP : emit wake interrupts
const u32	qos	Quality of Service (0-100) required
const enum `XPmRequestAck`	ack	Requested acknowledge type

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

XPm_SetRequirement

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

Note: If this function is called after the last awake CPU within the PU calls SelfSuspend, the requirement change shall be performed after the CPU signals the end of suspend to the power management controller, (e.g. WFI interrupt).

Prototype

```
XStatus XPm_SetRequirement(const enum XPmNodeId nid, const u32
capabilities, const u32 qos, const enum XPmRequestAck ack);
```

Parameters

The following table lists the `XPm_SetRequirement` function arguments.

Table 14: **XPm_SetRequirement Arguments**

Type	Name	Description
const enum XPmNodeId	nid	Node ID of the PM slave.
const u32	capabilities	Slave-specific capabilities required.
const u32	qos	Quality of Service (0-100) required.
const enum XPmRequestAck	ack	Requested acknowledge type

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

XPm_ReleaseNode

This function is used by a PU to release the usage of a PM slave. This will tell the power management controller that the node is no longer needed by that PU, potentially allowing the node to be placed into an inactive state.

Note: None

Prototype

```
XStatus XPm_ReleaseNode(const enum XPmNodeId node);
```

Parameters

The following table lists the `XPm_ReleaseNode` function arguments.

Table 15: **XPm_ReleaseNode Arguments**

Type	Name	Description
const enum XPmNodeId	node	Node ID of the PM slave.

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

XPm_SetMaxLatency

This function is used by a PU to announce a change in the maximum wake-up latency requirements for a specific slave node currently used by that PU.

Note: Setting maximum wake-up latency can constrain the set of possible power states a resource can be put into.

Prototype

```
XStatus XPm_SetMaxLatency(const enum XPmNodeId node, const u32 latency);
```

Parameters

The following table lists the `XPm_SetMaxLatency` function arguments.

Table 16: **XPm_SetMaxLatency Arguments**

Type	Name	Description
const enum XPmNodeId	node	Node ID of the PM slave.
const u32	latency	Maximum wake-up latency required.

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

XPm_InitSuspendCb

Callback function to be implemented in each PU, allowing the power management controller to request that the PU suspend itself.

Note: If the PU fails to act on this request the power management controller or the requesting PU may choose to employ the forceful power down option.

Prototype

```
void XPm_InitSuspendCb(const enum XPmSuspendReason reason, const u32
latency, const u32 state, const u32 timeout);
```

Parameters

The following table lists the `XPm_InitSuspendCb` function arguments.

Table 17: **XPm_InitSuspendCb Arguments**

Type	Name	Description
const enum XPmSuspendReason	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

Table 17: **XPm_InitSuspendCb Arguments** *(cont'd)*

Type	Name	Description
const u32	latency	Maximum wake-up latency in us(micro secs). This information can be used by the PU to decide what level of context saving may be required.
const u32	state	Targeted sleep/suspend state.
const u32	timeout	Timeout in ms, specifying how much time a PU has to initiate its suspend procedure before it's being considered unresponsive.

Returns

None

XPm_AcknowledgeCb

This function is called by the power management controller in response to any request where an acknowledge callback was requested, i.e. where the 'ack' argument passed by the PU was REQUEST_ACK_NON_BLOCKING.

***Note*:** None

Prototype

```
void XPm_AcknowledgeCb(const enum XPmNodeId node, const XStatus status,
const u32 oppoint);
```

Parameters

The following table lists the `XPm_AcknowledgeCb` function arguments.

Table 18: **XPm_AcknowledgeCb Arguments**

Type	Name	Description
const enum `XPmNodeId`	node	ID of the component or sub-system in question.
const XStatus	status	Status of the operation: • OK: the operation completed successfully • ERR: the requested operation failed
const u32	oppoint	Operating point of the node in question

Returns

None

XPm_NotifyCb

This function is called by the power management controller if an event the PU was registered for has occurred. It will populate the notifier data structure passed when calling XPm_RegisterNotifier.

Note: None

Prototype

```
void XPm_NotifyCb(const enum XPmNodeId node, const enum XPmNotifyEvent
event, const u32 oppoint);
```

Parameters

The following table lists the `XPm_NotifyCb` function arguments.

Table 19: **XPm_NotifyCb Arguments**

Type	Name	Description
const enum `XPmNodeId`	node	ID of the node the event notification is related to.
const enum `XPmNotifyEvent`	event	ID of the event
const u32	oppoint	Current operating state of the node.

Returns

None

XPm_GetApiVersion

This function is used to request the version number of the API running on the power management controller.

Note: None

Prototype

```
XStatus XPm_GetApiVersion(u32 *version);
```

Parameters

The following table lists the `XPm_GetApiVersion` function arguments.

Table 20: **XPm_GetApiVersion Arguments**

Type	Name	Description
u32 *	version	Returns the API 32-bit version number. Returns 0 if no PM firmware present.

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

XPm_GetNodeStatus

This function is used to obtain information about the current state of a component. The caller must pass a pointer to an `XPm_NodeStatus` structure, which must be pre-allocated by the caller.

- status - The current power state of the requested node.
  - For CPU nodes:
    - 0 : if CPU is powered down,
    - 1 : if CPU is active (powered up),
    - 2 : if CPU is suspending (powered up)
  - For power islands and power domains:
    - 0 : if island is powered down,
    - 1 : if island is powered up
  - For PM slaves:
    - 0 : if slave is powered down,
    - 1 : if slave is powered up,
    - 2 : if slave is in retention
- requirement - Slave nodes only: Returns current requirements the requesting PU has requested of the node.
- 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)

Note: None

Prototype

```
XStatus XPm_GetNodeStatus(const enum XPmNodeId node, XPm_NodeStatus *const
nodestatus);
```

Parameters

The following table lists the `XPm_GetNodeStatus` function arguments.

Table 21: **XPm_GetNodeStatus Arguments**

Type	Name	Description
const enum `XPmNodeId`	node	ID of the component or sub-system in question.
`XPm_NodeStatus` *const	nodestatus	Used to return the complete status of the node.

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

XPm_GetOpCharacteristic

Call this function to request the power management controller to return information about an operating characteristic of a component.

Note: None

Prototype

```
XStatus XPm_GetOpCharacteristic(const enum XPmNodeId node, const enum
XPmOpCharType type, u32 *const result);
```

Parameters

The following table lists the `XPm_GetOpCharacteristic` function arguments.

Table 22: **XPm_GetOpCharacteristic Arguments**

Type	Name	Description
const enum `XPmNodeId`	node	ID of the component or sub-system in question.
const enum `XPmOpCharType`	type	Type of operating characteristic requested: • power (current power consumption), • latency (current latency in us to return to active state), • temperature (current temperature),
u32 *const	result	Used to return the requested operating characteristic.

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

XPm_ResetAssert

This function is used to assert or release reset for a particular reset line. Alternatively a reset pulse can be requested as well.

***Note*:** None

Prototype

```
XStatus XPm_ResetAssert(const enum XPmReset reset, const enum
XPmResetAction resetaction);
```

Parameters

The following table lists the `XPm_ResetAssert` function arguments.

Table 23: **XPm_ResetAssert Arguments**

Type	Name	Description
const enum `XPmReset`	reset	ID of the reset line
Commented parameter assert does not exist in function XPm_ResetAssert.	assert	Identifies action: • PM_RESET_ACTION_RELEASE : release reset, • PM_RESET_ACTION_ASSERT : assert reset, • PM_RESET_ACTION_PULSE : pulse reset,

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

XPm_ResetGetStatus

Call this function to get the current status of the selected reset line.

***Note*:** None

Prototype

```
XStatus XPm_ResetGetStatus(const enum XPmReset reset, u32 *status);
```

Parameters

The following table lists the `XPm_ResetGetStatus` function arguments.

Table 24: **XPm_ResetGetStatus Arguments**

Type	Name	Description
const enum `XPmReset`	reset	Reset line
u32 *	status	Status of specified reset (true - asserted, false - released)

Returns

Returns 1/XST_FAILURE for 'asserted' or 0/XST_SUCCESS for 'released'.

XPm_RegisterNotifier

A PU can call this function to request that the power management controller call its notify callback whenever a qualifying event occurs. One can request to be notified for a specific or any event related to a specific node.

- nodeID : ID of the node to be notified about,
- eventID : ID of the event in question, '-1' denotes all events (- EVENT_STATE_CHANGE, EVENT_ZERO_USERS),
- wake : true: wake up on event, false: do not wake up (only notify if awake), no buffering/ queueing
- callback : Pointer to the custom callback function to be called 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, may be set to NULL.
- received : Variable indicating how many times the notification has been received since the notifier is registered.

***Note*:** The caller shall initialize the notifier object before invoking the XPm_RegisteredNotifier function. While notifier is registered, the notifier object shall not be modified by the caller.

Prototype

```
XStatus XPm_RegisterNotifier(XPm_Notifier *const notifier);
```

Parameters

The following table lists the `XPm_RegisterNotifier` function arguments.

Table 25: **XPm_RegisterNotifier Arguments**

Type	Name	Description
XPm_Notifier *const	notifier	Pointer to the notifier object to be associated with the requested notification. The notifier object contains the following data related to the notification:

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

XPm_UnregisterNotifier

A PU calls this function to unregister for the previously requested notifications.

***Note*:** None

Prototype

```
XStatus XPm_UnregisterNotifier(XPm_Notifier *const notifier);
```

Parameters

The following table lists the `XPm_UnregisterNotifier` function arguments.

Table 26: **XPm_UnregisterNotifier Arguments**

Type	Name	Description
XPm_Notifier *const	notifier	Pointer to the notifier object associated with the previously requested notification

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

XPm_MmioWrite

Call this function to write a value directly into a register that isn't accessible directly, such as registers in the clock control unit. This call is bypassing the power management logic. The permitted addresses are subject to restrictions as defined in the PCW configuration.

***Note*:** If the access isn't permitted this function returns an error code.

Prototype

```
XStatus XPm_MmioWrite(const u32 address, const u32 mask, const u32 value);
```

Parameters

The following table lists the `XPm_MmioWrite` function arguments.

Table 27: **XPm_MmioWrite Arguments**

Type	Name	Description
const u32	address	Physical 32-bit address of memory mapped register to write to.
const u32	mask	32-bit value used to limit write to specific bits in the register.
const u32	value	Value to write to the register bits specified by the mask.

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

XPm_MmioRead

Call this function to read a value from a register that isn't accessible directly. The permitted addresses are subject to restrictions as defined in the PCW configuration.

Note: If the access isn't permitted this function returns an error code.

Prototype

```
XStatus XPm_MmioRead(const u32 address, u32 *const value);
```

Parameters

The following table lists the `XPm_MmioRead` function arguments.

Table 28: **XPm_MmioRead Arguments**

Type	Name	Description
const u32	address	Physical 32-bit address of memory mapped register to read from.
u32 *const	value	Returns the 32-bit value read from the register

Returns

XST_SUCCESS if successful else XST_FAILURE or an error code or a reason code

XPm_ClockEnable

Call this function to enable (activate) a clock.

Note: If the access isn't permitted this function returns an error code.

Prototype

```
XStatus XPm_ClockEnable(const enum XPmClock clock);
```

Parameters

The following table lists the `XPm_ClockEnable` function arguments.

Table 29: **XPm_ClockEnable Arguments**

Type	Name	Description
const enum `XPmClock`	clock	Identifier of the target clock to be enabled

Returns

Status of performing the operation as returned by the PMU-FW

XPm_ClockDisable

Call this function to disable (gate) a clock.

Note: If the access isn't permitted this function returns an error code.

Prototype

```
XStatus XPm_ClockDisable(const enum XPmClock clock);
```

Parameters

The following table lists the `XPm_ClockDisable` function arguments.

Table 30: **XPm_ClockDisable Arguments**

Type	Name	Description
const enum `XPmClock`	clock	Identifier of the target clock to be disabled

Returns

Status of performing the operation as returned by the PMU-FW

XPm_ClockGetStatus

Call this function to get status of a clock gate state.

Prototype

```
XStatus XPm_ClockGetStatus(const enum XPmClock clock, u32 *const status);
```

Parameters

The following table lists the `XPm_ClockGetStatus` function arguments.

Table 31: **XPm_ClockGetStatus Arguments**

Type	Name	Description
const enum `XPmClock`	clock	Identifier of the target clock
u32 *const	status	Location to store clock gate state (1=enabled, 0=disabled)

Returns

Status of performing the operation as returned by the PMU-FW

XPm_ClockSetOneDivider

Call this function to set divider for a clock.

***Note*:** If the access isn't permitted this function returns an error code.

Prototype

```
XStatus XPm_ClockSetOneDivider(const enum XPmClock clock, const u32
divider, const u32 divId);
```

Parameters

The following table lists the `XPm_ClockSetOneDivider` function arguments.

Table 32: **XPm_ClockSetOneDivider Arguments**

Type	Name	Description
const enum `XPmClock`	clock	Identifier of the target clock
const u32	divider	Divider value to be set
const u32	divId	ID of the divider to be set

Returns

Status of performing the operation as returned by the PMU-FW

XPm_ClockSetDivider

Call this function to set divider for a clock.

Note: If the access isn't permitted this function returns an error code.

Prototype

```
XStatus XPm_ClockSetDivider(const enum XPmClock clock, const u32 divider);
```

Parameters

The following table lists the `XPm_ClockSetDivider` function arguments.

Table 33: **XPm_ClockSetDivider Arguments**

Type	Name	Description
const enum `XPmClock`	clock	Identifier of the target clock
const u32	divider	Divider value to be set

Returns

XST_INVALID_PARAM or status of performing the operation as returned by the PMU-FW

XPm_ClockGetOneDivider

Local function to get one divider (DIV0 or DIV1) of a clock.

Prototype

```
XStatus XPm_ClockGetOneDivider(const enum XPmClock clock, u32 *const
divider, const u32 divId);
```

Parameters

The following table lists the `XPm_ClockGetOneDivider` function arguments.

Table 34: **XPm_ClockGetOneDivider Arguments**

Type	Name	Description
const enum `XPmClock`	clock	Identifier of the target clock
u32 *const	divider	Location to store the divider value

Returns

Status of performing the operation as returned by the PMU-FW

XPm_ClockGetDivider

Call this function to get divider of a clock.

Prototype

```
XStatus XPm_ClockGetDivider(const enum XPmClock clock, u32 *const divider);
```

Parameters

The following table lists the `XPm_ClockGetDivider` function arguments.

Table 35: **XPm_ClockGetDivider Arguments**

Type	Name	Description
const enum `XPmClock`	clock	Identifier of the target clock
u32 *const	divider	Location to store the divider value

Returns

XST_INVALID_PARAM or status of performing the operation as returned by the PMU-FW

XPm_ClockSetParent

Call this function to set parent for a clock.

***Note*:** If the access isn't permitted this function returns an error code.

Prototype

```
XStatus XPm_ClockSetParent(const enum XPmClock clock, const enum XPmClock
parent);
```

Parameters

The following table lists the `XPm_ClockSetParent` function arguments.

Table 36: **XPm_ClockSetParent Arguments**

Type	Name	Description
const enum `XPmClock`	clock	Identifier of the target clock
const enum `XPmClock`	parent	Identifier of the target parent clock

Returns

XST_INVALID_PARAM or status of performing the operation as returned by the PMU-FW.

XPm_ClockGetParent

Call this function to get parent of a clock.

Prototype

```
XStatus XPm_ClockGetParent(const enum XPmClock clock, enum XPmClock *const
parent);
```

Parameters

The following table lists the `XPm_ClockGetParent` function arguments.

Table 37: **XPm_ClockGetParent Arguments**

Type	Name	Description
const enum `XPmClock`	clock	Identifier of the target clock
enum `XPmClock` *const	parent	Location to store clock parent ID

Returns

XST_INVALID_PARAM or status of performing the operation as returned by the PMU-FW.

XPm_ClockSetRate

Call this function to set rate of a clock.

***Note*:** If the action isn't permitted this function returns an error code.

Prototype

```
XStatus XPm_ClockSetRate(const enum XPmClock clock, const u32 rate);
```

Parameters

The following table lists the `XPm_ClockSetRate` function arguments.

Table 38: **XPm_ClockSetRate Arguments**

Type	Name	Description
const enum `XPmClock`	clock	Identifier of the target clock
const u32	rate	Clock frequency (rate) to be set

Returns

Status of performing the operation as returned by the PMU-FW

XPm_ClockGetRate

Call this function to get rate of a clock.

Prototype

```
XStatus XPm_ClockGetRate(const enum XPmClock clock, u32 *const rate);
```

Parameters

The following table lists the `XPm_ClockGetRate` function arguments.

Table 39: **XPm_ClockGetRate Arguments**

Type	Name	Description
const enum `XPmClock`	clock	Identifier of the target clock
u32 *const	rate	Location where the rate should be stored

Returns

Status of performing the operation as returned by the PMU-FW

XPm_PllSetParameter

Call this function to set a PLL parameter.

***Note*:** If the access isn't permitted this function returns an error code.

Prototype

```
XStatus XPm_PllSetParameter(const enum XPmNodeId node, const enum
XPmPllParam parameter, const u32 value);
```

Parameters

The following table lists the `XPm_PllSetParameter` function arguments.

Table 40: **XPm_PllSetParameter Arguments**

Type	Name	Description
const enum `XPmNodeId`	node	PLL node identifier
const enum XPmPllParam	parameter	PLL parameter identifier
const u32	value	Value of the PLL parameter

Returns

Status of performing the operation as returned by the PMU-FW

XPm_PllGetParameter

Call this function to get a PLL parameter.

Prototype

```
XStatus XPm_PllGetParameter(const enum XPmNodeId node, const enum
XPmPllParam parameter, u32 *const value);
```

Parameters

The following table lists the `XPm_PllGetParameter` function arguments.

Table 41: **XPm_PllGetParameter Arguments**

Type	Name	Description
const enum `XPmNodeId`	node	PLL node identifier
const enum XPmPllParam	parameter	PLL parameter identifier
u32 *const	value	Location to store value of the PLL parameter

Returns

Status of performing the operation as returned by the PMU-FW

XPm_PllSetMode

Call this function to set a PLL mode.

***Note*:** If the access isn't permitted this function returns an error code.

Prototype

```
XStatus XPm_PllSetMode(const enum XPmNodeId node, const enum XPmPllMode
mode);
```

Parameters

The following table lists the `XPm_PllSetMode` function arguments.

Table 42: **XPm_PllSetMode Arguments**

Type	Name	Description
const enum XPmNodeId	node	PLL node identifier
const enum XPmPllMode	mode	PLL mode to be set

Returns

Status of performing the operation as returned by the PMU-FW

XPm_PllGetMode

Call this function to get a PLL mode.

Prototype

```
XStatus XPm_PllGetMode(const enum XPmNodeId node, enum XPmPllMode *const
mode);
```

Parameters

The following table lists the `XPm_PllGetMode` function arguments.

Table 43: **XPm_PllGetMode Arguments**

Type	Name	Description
const enum XPmNodeId	node	PLL node identifier
enum XPmPllMode *const	mode	Location to store the PLL mode

Returns

Status of performing the operation as returned by the PMU-FW

XPm_PinCtrlAction

Locally used function to request or release a pin control.

Prototype

```
XStatus XPm_PinCtrlAction(const u32 pin, const enum XPmApiId api);
```

Parameters

The following table lists the `XPm_PinCtrlAction` function arguments.

Table 44: **XPm_PinCtrlAction Arguments**

Type	Name	Description
const u32	pin	PIN identifier (index from range 0-77) @api API identifier (request or release pin control)

Returns

Status of performing the operation as returned by the PMU-FW

XPm_PinCtrlRequest

Call this function to request a pin control.

Prototype

```
XStatus XPm_PinCtrlRequest(const u32 pin);
```

Parameters

The following table lists the `XPm_PinCtrlRequest` function arguments.

Table 45: **XPm_PinCtrlRequest Arguments**

Type	Name	Description
const u32	pin	PIN identifier (index from range 0-77)

Returns

Status of performing the operation as returned by the PMU-FW

XPm_PinCtrlRelease

Call this function to release a pin control.

Prototype

```
XStatus XPm_PinCtrlRelease(const u32 pin);
```

Parameters

The following table lists the `XPm_PinCtrlRelease` function arguments.

Table 46: **XPm_PinCtrlRelease Arguments**

Type	Name	Description
const u32	pin	PIN identifier (index from range 0-77)

Returns

Status of performing the operation as returned by the PMU-FW

XPm_PinCtrlSetFunction

Call this function to set a pin function.

***Note*:** If the access isn't permitted this function returns an error code.

Prototype

```
XStatus XPm_PinCtrlSetFunction(const u32 pin, const enum XPmPinFn fn);
```

Parameters

The following table lists the `XPm_PinCtrlSetFunction` function arguments.

Table 47: **XPm_PinCtrlSetFunction Arguments**

Type	Name	Description
const u32	pin	Pin identifier
const enum XPmPinFn	fn	Pin function to be set

Returns

Status of performing the operation as returned by the PMU-FW

XPm_PinCtrlGetFunction

Call this function to get currently configured pin function.

Prototype

```
XStatus XPm_PinCtrlGetFunction(const u32 pin, enum XPmPinFn *const fn);
```

Parameters

The following table lists the `XPm_PinCtrlGetFunction` function arguments.

Table 48: **XPm_PinCtrlGetFunction Arguments**

Type	Name	Description
const u32	pin	PLL node identifier
enum XPmPinFn *const	fn	Location to store the pin function

Returns

Status of performing the operation as returned by the PMU-FW

XPm_PinCtrlSetParameter

Call this function to set a pin parameter.

***Note*:** If the access isn't permitted this function returns an error code.

Prototype

```
XStatus XPm_PinCtrlSetParameter(const u32 pin, const enum XPmPinParam
param, const u32 value);
```

Parameters

The following table lists the `XPm_PinCtrlSetParameter` function arguments.

Table 49: **XPm_PinCtrlSetParameter Arguments**

Type	Name	Description
const u32	pin	Pin identifier
const enum XPmPinParam	param	Pin parameter identifier
const u32	value	Value of the pin parameter to set

Returns

Status of performing the operation as returned by the PMU-FW

XPm_PinCtrlGetParameter

Call this function to get currently configured value of pin parameter.

Prototype

```
XStatus XPm_PinCtrlGetParameter(const u32 pin, const enum XPmPinParam
param, u32 *const value);
```

Parameters

The following table lists the `XPm_PinCtrlGetParameter` function arguments.

Table 50: **XPm_PinCtrlGetParameter Arguments**

Type	Name	Description
const u32	pin	Pin identifier
const enum XPmPinParam	param	Pin parameter identifier
u32 *const	value	Location to store value of the pin parameter

Returns

Status of performing the operation as returned by the PMU-FW

Error Status

This section lists the Power management specific return error statuses.

Definitions

Define XST_PM_INTERNAL

Definition

```
#define XST_PM_INTERNAL2000L
```

Description

An internal error occurred while performing the requested operation

Define XST_PM_CONFLICT

Definition

```
#define XST_PM_CONFLICT2001L
```

Description

Conflicting requirements have been asserted when more than one processing cluster is using the same PM slave

Define XST_PM_NO_ACCESS

Definition

```
#define XST_PM_NO_ACCESS2002L
```

Description

The processing cluster does not have access to the requested node or operation

Define XST_PM_INVALID_NODE

Definition

```
#define XST_PM_INVALID_NODE2003L
```

Description

The API function does not apply to the node passed as argument

Define XST_PM_DOUBLE_REQ

Definition

```
#define XST_PM_DOUBLE_REQ2004L
```

Description

A processing cluster has already been assigned access to a PM slave and has issued a duplicate request for that PM slave

Define XST_PM_ABORT_SUSPEND

Definition

```
#define XST_PM_ABORT_SUSPEND2005L
```

Description

The target processing cluster has aborted suspend

Define XST_PM_TIMEOUT

Definition

```
#define XST_PM_TIMEOUT2006L
```

Description

A timeout occurred while performing the requested operation

Define XST_PM_NODE_USED

Definition

```
#define XST_PM_NODE_USED2007L
```

Description

Slave request cannot be granted since node is non-shareable and used

Data Structure Index

The following is a list of data structures:

pm_acknowledge

Declaration

```
typedef struct
{
  bool received,
  enum XPmNodeId node,
  XStatus status,
  u32 opp
} pm_acknowledge;
```

Table 51: **Structure pm_acknowledge member description**

Member	Description
received	Has acknowledge argument been received?
node	Node argument about which the acknowledge is
status	Acknowledged status
opp	Operating point of node in question

pm_init_suspend

Declaration

```
typedef struct
{
  bool received,
  enum XPmSuspendReason reason,
  u32 latency,
  u32 state,
  u32 timeout
} pm_init_suspend;
```

Table 52: **Structure pm_init_suspend member description**

Member	Description
received	Has init suspend callback been received/handled
reason	Reason of initializing suspend
latency	Maximum allowed latency
state	Targeted sleep/suspend state
timeout	Period of time the client has to response

XPm_Master

`XPm_Master` - Master structure

Declaration

```
typedef struct
{
  enum XPmNodeId node_id,
  const u32 pwrctl,
  const u32 pwrdn_mask,
  XIpiPsu * ipi
} XPm_Master;
```

Table 53: **Structure XPm_Master member description**

Member	Description
node_id	Node ID
pwrctl	
pwrdn_mask	< Power Control Register Address Power Down Mask
ipi	IPI Instance

XPm_NodeStatus

`XPm_NodeStatus` - struct containing node status information

Declaration

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

Table 54: **Structure XPm_NodeStatus member description**

Member	Description
status	Node power state
requirements	Current requirements asserted on the node (slaves only)
usage	Usage information (which master is currently using the slave)

XPm_Notifier

`XPm_Notifier` - Notifier structure registered with a callback by app

Declaration

```
typedef struct
{
  void(*const callback)(struct XPm_Ntfier *const notifier),
  enum XPmNodeId node,
  enum XPmNotifyEvent event,
  u32 flags,
  u32 oppoint,
  u32 received,
  struct XPm_Ntfier * next
} XPm_Notifier;
```

Table 55: **Structure XPm_Notifier member description**

Member	Description
callback	Custom callback handler to be called when the notification is received. The custom handler would execute from interrupt context, 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

Table 55: **Structure XPm_Notifier member description** *(cont'd)*

Member	Description
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.

XILINX

Appendix A

Additional Resources and Legal Notices

Xilinx Resources

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

Documentation Navigator and Design Hubs

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

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

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 DocNav, click the **Design Hubs View** tab.
- On the website, see the Design Hubs page.

Note: For more information on DocNav, see the Documentation Navigator page on the website.

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 https://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 https://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