

# Xilinx Standalone Library Documentation

## *XilPM Library v2.1*

UG1225 (2017.1) April 5, 2017

# Table of Contents

## Chapter 1 XiIPM APIs

<b>Overview</b>	<b>3</b>
<b>Data Structure Documentation</b>	<b>4</b>
struct XPm_Notifier	4
struct XPm_NodeStatus	5
<b>Function Documentation</b>	<b>5</b>
XPm_InitXilpm	5
XPm_SuspendFinalize	6
XPm_GetBootStatus	6
XPm_RequestSuspend	6
XPm_SelfSuspend	7
XPm_ForcePowerDown	7
XPm_AbortSuspend	8
XPm_RequestWakeUp	8
XPm_SetWakeUpSource	9
XPm_SystemShutdown	10
XPm_SetConfiguration	10
XPm_InitSuspendCb	11
XPm_AcknowledgeCb	11
XPm_NotifyCb	12
XPm_RequestNode	12
XPm_ReleaseNode	13
XPm_SetRequirement	14
XPm_SetMaxLatency	14
XPm_GetApiVersion	15
XPm_GetNodeStatus	15
XPm_RegisterNotifier	16
XPm_UnregisterNotifier	17
XPm_GetOpCharacteristic	18
XPm_ResetAssert	18
XPm_ResetGetStatus	19
XPm_MmioWrite	19

## Appendix A Additional Resources and Legal Notices

## XilPM APIs

---

### Overview

Xilinx Power Management(XilPM) provides Embedded Energy Management Interface (EEMI) APIs for power management on Zynq® UltraScale+™ MPSoC. For more details about power management on Zynq Ultrascale+ MPSoC, see the Zynq UltraScale+ MPSoC Power Management User Guide (UG1199). For more details about EEMI, see the Embedded Energy Management Interface (EEMI) API User Guide(UG1200).

---

### Data Structures

- struct [XPm\\_Notifier](#)
- struct [XPm\\_NodeStatus](#)

---

### Functions

- XStatus [XPm\\_InitXilpm](#) (XlpiPsu \*lpInst)
- void [XPm\\_SuspendFinalize](#) ()
- enum XPmBootStatus [XPm\\_GetBootStatus](#) ()
- XStatus [XPm\\_RequestSuspend](#) (const enum XPmNodeId node, const enum XPmRequestAck ack, const u32 latency, const u8 state)
- XStatus [XPm\\_SelfSuspend](#) (const enum XPmNodeId node, const u32 latency, const u8 state, const u64 address)
- XStatus [XPm\\_ForcePowerDown](#) (const enum XPmNodeId node, const enum XPmRequestAck ack)
- XStatus [XPm\\_AbortSuspend](#) (const enum XPmAbortReason reason)
- XStatus [XPm\\_RequestWakeUp](#) (const enum XPmNodeId node, const bool setAddress, const u64 address, const enum XPmRequestAck ack)
- XStatus [XPm\\_SetWakeUpSource](#) (const enum XPmNodeId target, const enum XPmNodeId wkup\_node, const u8 enable)
- XStatus [XPm\\_SystemShutdown](#) (u32 type, u32 subtype)
- XStatus [XPm\\_SetConfiguration](#) (const u32 address)
- 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)
- void [XPm\\_NotifyCb](#) (const enum XPmNodeId node, const u32 event, const u32 oppoint)

- XStatus [XPm\\_RequestNode](#) (const enum XPmNodeId node, const u32 capabilities, const u32 qos, const enum XPmRequestAck ack)
- XStatus [XPm\\_ReleaseNode](#) (const enum XPmNodeId node)
- XStatus [XPm\\_SetRequirement](#) (const enum XPmNodeId node, const u32 capabilities, const u32 qos, const enum XPmRequestAck ack)
- XStatus [XPm\\_SetMaxLatency](#) (const enum XPmNodeId node, const u32 latency)
- XStatus [XPm\\_GetApiVersion](#) (u32 \*version)
- XStatus [XPm\\_GetNodeStatus](#) (const enum XPmNodeId node, [XPm\\_NodeStatus](#) \*const nodestatus)
- XStatus [XPm\\_RegisterNotifier](#) ([XPm\\_Notifier](#) \*const notifier)
- XStatus [XPm\\_UnregisterNotifier](#) ([XPm\\_Notifier](#) \*const notifier)
- XStatus [XPm\\_GetOpCharacteristic](#) (const enum XPmNodeId node, const enum XPmOpCharType type, u32 \*const result)
- XStatus [XPm\\_ResetAssert](#) (const enum XPmReset reset, const enum XPmResetAction assert)
- XStatus [XPm\\_ResetGetStatus](#) (const enum XPmReset reset, u32 \*status)
- XStatus [XPm\\_MmioWrite](#) (const u32 address, const u32 mask, const u32 value)
- XStatus [XPm\\_MmioRead](#) (const u32 address, u32 \*const value)

---

## Data Structure Documentation

### struct XPm\_Notifier

Notifier structure registered with a callback by application.

### Data Fields

- void(\*const [callback](#)) ([XPm\\_Notifier](#) \*const notifier)
- enum XPmNodeId [node](#)
- enum XPmNotifyEvent [event](#)
- u32 [flags](#)
- volatile u32 [oppoint](#)
- volatile u32 [received](#)
- [XPm\\_Notifier](#) \* [next](#)

### Field Documentation

**void(\*const callback) (XPm\_Notifier \*const notifier)** 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)

**enum XPmNodeId node** Node argument (the node to receive notifications about)

**enum XPmNotifyEvent event** Event argument (the event type to receive notifications about)

## u32 flags Flags

**volatile u32 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.

**volatile u32 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.

**XPm\_Notifier\* next** Pointer to next notifier in linked list. Must not be modified while the notifier is registered. User shall not ever modify this value.

## struct XPm\_NodeStatus

Data structure containing node status information.

### Data Fields

- u32 [status](#)
- u32 [requirements](#)
- u32 [usage](#)

### Field Documentation

**u32 status** Node power state

**u32 requirements** Current requirements asserted on the node (slaves only)

**u32 usage** Usage information (which master is currently using the slave)

---

## Function Documentation

### XStatus XPm\_InitXilpm ( XlpiPsu \* *IpInst* )

Initialize xilpm library.

#### Parameters

<i>IpInst</i>	Pointer to IPI driver instance
---------------	--------------------------------

#### Returns

XST\_SUCCESS if successful else XST\_FAILURE or an error code or a reason code

## Note

None

## void XPm\_SuspendFinalize ( void )

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.

## enum XPmBootStatus XPm\_GetBootStatus ( void )

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.

## 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.

## Note

None

## XStatus XPm\_RequestSuspend ( const enum XPmNodeid *target*, const enum XPmRequestAck *ack*, const u32 *latency*, const u8 *state* )

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.

## Parameters

<i>target</i>	Node ID of the PU node to be suspended
<i>ack</i>	Requested acknowledge type
<i>latency</i>	Maximum wake-up latency requirement in us(micro sec)
<i>state</i>	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

## Note

If 'ack' is set to PM\_ACK\_CB\_STANDARD, the requesting PU will be notified upon completion of suspend or if an error occurred, such as an abort or a timeout.

## XStatus XPm\_SelfSuspend ( const enum XPmNodeId *nid*, const u32 *latency*, const u8 *state*, const u64 *address* )

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.

### Parameters

<i>nid</i>	Node ID of the CPU node to be suspended.
<i>latency</i>	Maximum wake-up latency requirement in us(microsecs)
<i>state</i>	Instead of specifying a maximum latency, a CPU can also explicitly request a certain power state.
<i>address</i>	Address from which to resume when woken up.

## Returns

XST\_SUCCESS if successful else XST\_FAILURE or an error code or a reason code

## Note

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

## XStatus XPm\_ForcePowerDown ( const enum XPmNodeId *target*, const enum XPmRequestAck *ack* )

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.

### Parameters

<i>target</i>	Node ID of the PU node or power island/domain to be powered down.
<i>ack</i>	Requested acknowledge type

## Returns

XST\_SUCCESS if successful else XST\_FAILURE or an error code or a reason code

## Note

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

# XStatus XPm\_AbortSuspend ( const enum XPmAbortReason *reason* )

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.

## Parameters

<i>reason</i>	Reason code why the suspend can not be performed or completed <ul style="list-style-type: none"> <li>• ABORT_REASON_WKUP_EVENT : local wakeup-event received</li> <li>• ABORT_REASON_PU_BUSY : PU is busy</li> <li>• ABORT_REASON_NO_PWRDN : no external powerdown supported</li> <li>• ABORT_REASON_UNKNOWN : unknown error during suspend procedure</li> </ul>
---------------	--

## Returns

XST\_SUCCESS if successful else XST\_FAILURE or an error code or a reason code

## Note

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

# XStatus XPm\_RequestWakeUp ( const enum XPmNodeId *target*, const bool *setAddress*, const u64 *address*, const enum XPmRequestAck *ack* )

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

## Parameters

<i>target</i>	Node ID of the CPU or PU to be powered/woken up.
<i>setAddress</i>	Specifies whether the start address argument is being passed. <ul style="list-style-type: none"> <li>• 0 : do not set start address</li> <li>• 1 : set start address</li> </ul>
<i>address</i>	Address from which to resume when woken up. Will only be used if set_address is 1.
<i>ack</i>	Requested acknowledge type

## Returns

XST\_SUCCESS if successful else XST\_FAILURE or an error code or a reason code

## Note

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

**XStatus XPm\_SetWakeUpSource ( const enum XPmNodeId *target*, const enum XPmNodeId *wkup\_node*, const u8 *enable* )**

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.

## Parameters

<i>target</i>	Node ID of the target to be woken up.
<i>wkup_node</i>	Node ID of the wakeup device.
<i>enable</i>	Enable flag: <ul style="list-style-type: none"> <li>• 1 : the wakeup source is added to the list</li> <li>• 0 : the wakeup source is removed from the list</li> </ul>

## Returns

XST\_SUCCESS if successful else XST\_FAILURE or an error code or a reason code

## 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.

## XStatus XPm\_SystemShutdown ( *u32 type*, *u32 subtype* )

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

### Parameters

<i>restart</i>	Should the system be restarted automatically? <ul style="list-style-type: none"> <li>PM_SHUTDOWN : no restart requested, system will be powered off permanently</li> <li>PM_RESTART : restart is requested, system will go through a full reset</li> </ul>
----------------	--

### Returns

XST\_SUCCESS if successful else XST\_FAILURE or an error code or a reason code

## 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.

## XStatus XPm\_SetConfiguration ( *const u32 address* )

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.

### Parameters

<i>address</i>	Start address of the configuration object
----------------	---

### Returns

XST\_SUCCESS if successful, otherwise an error code

## Note

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

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

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

#### Parameters

<i>reason</i>	Suspend reason: <ul style="list-style-type: none"> <li>• SUSPEND_REASON_PU_REQ : Request by another PU</li> <li>• SUSPEND_REASON_ALERT : Unrecoverable SysMon alert</li> <li>• SUSPEND_REASON_SHUTDOWN : System shutdown</li> <li>• SUSPEND_REASON_RESTART : System restart</li> </ul>
<i>latency</i>	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.
<i>state</i>	Targeted sleep/suspend state.
<i>timeout</i>	Timeout in ms, specifying how much time a PU has to initiate its suspend procedure before it's being considered unresponsive.

#### Returns

None

#### 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.

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

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\_CB\_STANDARD.

## Parameters

<i>node</i>	ID of the component or sub-system in question.
<i>status</i>	Status of the operation: <ul style="list-style-type: none"> <li>• OK: the operation completed successfully</li> <li>• ERR: the requested operation failed</li> </ul>
<i>oppoint</i>	Operating point of the node in question

## Returns

None

## Note

None

**void XPm\_NotifyCb ( const enum XPmNodeId *node*, const u32 *event*, const u32 *oppoint* )**

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.

## Parameters

<i>node</i>	ID of the node the event notification is related to.
<i>event</i>	ID of the event
<i>oppoint</i>	Current operating state of the node.

## Returns

None

## Note

None

**XStatus XPm\_RequestNode ( const enum XPmNodeId *node*, const u32 *capabilities*, const u32 *qos*, const enum XPmRequestAck *ack* )**

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.

#### Parameters

<i>node</i>	Node ID of the PM slave requested
<i>capabilities</i>	Slave-specific capabilities required, can be combined <ul style="list-style-type: none"> <li>• PM_CAP_ACCESS : full access / functionality</li> <li>• PM_CAP_CONTEXT : preserve context</li> <li>• PM_CAP_WAKEUP : emit wake interrupts</li> </ul>
<i>qos</i>	Quality of Service (0-100) required
<i>ack</i>	Requested acknowledge type

#### Returns

XST\_SUCCESS if successful else XST\_FAILURE or an error code or a reason code

#### Note

None

## XStatus XPm\_ReleaseNode ( const enum XPmNodeId *node* )

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.

#### Parameters

<i>node</i>	Node ID of the PM slave.
-------------	--------------------------

#### Returns

XST\_SUCCESS if successful else XST\_FAILURE or an error code or a reason code

## Note

None

## **XStatus XPm\_SetRequirement ( const enum XPmNodeId *nid*, const u32 *capabilities*, const u32 *qos*, const enum XPmRequestAck *ack* )**

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

### Parameters

<i>nid</i>	Node ID of the PM slave.
<i>capabilities</i>	Slave-specific capabilities required.
<i>qos</i>	Quality of Service (0-100) required.
<i>ack</i>	Requested acknowledge type

### Returns

XST\_SUCCESS if successful else XST\_FAILURE or an error code or a reason code

## 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).

## **XStatus XPm\_SetMaxLatency ( const enum XPmNodeId *node*, const u32 *latency* )**

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.

### Parameters

<i>node</i>	Node ID of the PM slave.
<i>latency</i>	Maximum wake-up latency required.

### Returns

XST\_SUCCESS if successful else XST\_FAILURE or an error code or a reason code

#### Note

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

## XStatus XPm\_GetApiVersion ( u32 \* *version* )

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

#### Parameters

<i>version</i>	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

#### Note

None

## XStatus XPm\_GetNodeStatus ( const enum XPmNodeId *node*, XPm\_NodeStatus \*const *nodestatus* )

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.

## Parameters

<i>node</i>	ID of the component or sub-system in question.
<i>nodestatus</i>	<p>Used to return the complete status of the node.</p> <ul style="list-style-type: none"> <li>• status - The current power state of the requested node.           <ul style="list-style-type: none"> <li>◦ For CPU nodes:               <ul style="list-style-type: none"> <li>■ 0 : if CPU is powered down,</li> <li>■ 1 : if CPU is active (powered up),</li> <li>■ 2 : if CPU is suspending (powered up)</li> </ul> </li> <li>◦ For power islands and power domains:               <ul style="list-style-type: none"> <li>■ 0 : if island is powered down,</li> <li>■ 1 : if island is powered up</li> </ul> </li> <li>◦ For PM slaves:               <ul style="list-style-type: none"> <li>■ 0 : if slave is powered down,</li> <li>■ 1 : if slave is powered up,</li> <li>■ 2 : if slave is in retention</li> </ul> </li> </ul> </li> <li>• requirement - Slave nodes only: Returns current requirements the requesting PU has requested of the node.</li> <li>• usage - Slave nodes only: Returns current usage status of the node:           <ul style="list-style-type: none"> <li>◦ 0 : node is not used by any PU,</li> <li>◦ 1 : node is used by caller exclusively,</li> <li>◦ 2 : node is used by other PU(s) only,</li> <li>◦ 3 : node is used by caller and by other PU(s)</li> </ul> </li> </ul>

## Returns

XST\_SUCCESS if successful else XST\_FAILURE or an error code or a reason code

## Note

None

## XStatus XPm\_RegisterNotifier ( XPm\_Notifier \*const *notifier* )

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.

## Parameters

<i>notifier</i>	<p>Pointer to the notifier object to be associated with the requested notification. The notifier object contains the following data related to the notification:</p> <ul style="list-style-type: none"> <li>• <b>nodeID</b> : ID of the node to be notified about,</li> <li>• <b>eventID</b> : ID of the event in question, '-1' denotes all events ( - EVENT_STATE_CHANGE, EVENT_ZERO_USERS, EVENT_ERROR_CONDITION),</li> <li>• <b>wake</b> : true: wake up on event, false: do not wake up (only notify if awake), no buffering/queueing</li> <li>• <b>callback</b> : 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.</li> <li>• <b>received</b> : Variable indicating how many times the notification has been received since the notifier is registered.</li> </ul>
-----------------	---

## Returns

XST\_SUCCESS if successful else XST\_FAILURE or an error code or a reason code

## 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.

# XStatus XPm\_UnregisterNotifier ( XPm\_Notifier \*const *notifier* )

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

## Parameters

<i>notifier</i>	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

## Note

None

## XStatus XPm\_GetOpCharacteristic ( const enum XPmNodeId *node*, const enum XPmOpCharType *type*, u32 \*const *result* )

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

### Parameters

<i>node</i>	ID of the component or sub-system in question.
<i>type</i>	Type of operating characteristic requested: <ul style="list-style-type: none"> <li>• power (current power consumption),</li> <li>• latency (current latency in us to return to active state),</li> <li>• temperature (current temperature),</li> </ul>
<i>result</i>	Used to return the requested operating characteristic.

### Returns

XST\_SUCCESS if successful else XST\_FAILURE or an error code or a reason code

### Note

None

## XStatus XPm\_ResetAssert ( const enum XPmReset *reset*, const enum XPmResetAction *assert* )

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

### Parameters

<i>reset</i>	ID of the reset line
<i>assert</i>	Identifies action: <ul style="list-style-type: none"> <li>• PM_RESET_ACTION_RELEASE : release reset,</li> <li>• PM_RESET_ACTION_ASSERT : assert reset,</li> <li>• PM_RESET_ACTION_PULSE : pulse reset,</li> </ul>

## Returns

XST\_SUCCESS if successful else XST\_FAILURE or an error code or a reason code

## Note

None

# XStatus XPm\_ResetGetStatus ( const enum XPmReset *reset*, u32 \* *status* )

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

## Parameters

<i>reset</i>	Reset line
<i>status</i>	Status of specified reset (true - asserted, false - released)

## Returns

Returns 1/XST\_FAILURE for 'asserted' or 0/XST\_SUCCESS for 'released'.

## Note

None

# XStatus XPm\_MmioWrite ( const u32 *address*, const u32 *mask*, const u32 *value* )

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.

## Parameters

<i>address</i>	Physical 32-bit address of memory mapped register to write to.
<i>mask</i>	32-bit value used to limit write to specific bits in the register.
<i>value</i>	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

## Note

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

## **XStatus XPm\_MmioRead ( const u32 *address*, u32 \*const *value* )**

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.

### **Parameters**

<i>address</i>	Physical 32-bit address of memory mapped register to read from.
<i>value</i>	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

### **Note**

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

# 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.

---

## Please Read: Important Legal Notices

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

**Automotive Applications Disclaimer**

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

© Copyright 2017 Xilinx, Inc. Xilinx, the Xilinx logo, Artix, ISE, Kintex, Spartan, Virtex, Vivado, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. All other trademarks are the property of their respective owners.