

# Embedded Energy Management Interface

## *EEMI API Reference Guide*

UG1200 (v2.0) May 3, 2017

---

## Revision History

The following table shows the revision history for this document.

Date	Version	Revision
05/03/2017	2.0	Second Release In <a href="#">Chapter 2</a> , added <a href="#">system_shutdown</a> and <a href="#">get_boot_status</a> .
09/07/2016	1.0	Initial Xilinx release.

# Table of Contents

Revision History .....	2
<b>Chapter 1: Using EEMI APIs</b>	
Introduction .....	4
Architecture and API Layers .....	4
APIs in Alphabetical Order .....	6
<b>Chapter 2: System-Level and OS-Level EEMI API Functions</b>	
Introduction .....	7
Caller and Callee Responsibilities .....	7
Acknowledge Mechanism .....	8
Suspend APIs to Suspend Clusters .....	8
Wake API Functions .....	14
APIs for Managing PM Slaves on a Cluster .....	17
Miscellaneous API Functions .....	21
Direct Control API Functions .....	26
<b>Chapter 3: Error Codes</b>	
Error Codes .....	29
<b>Appendix A: Additional Resources and Legal Notices</b>	
Xilinx Resources .....	30
Solution Centers .....	30
Documentation Navigator and Design Hubs .....	30
References .....	31
Please Read: Important Legal Notices .....	31

# Using EEMI APIs

---

## Introduction

The embedded energy management interface (EEMI) is used to allow software components running across different processing clusters on a chip or device to communicate with a power management controller (PMC) on a device to issue or respond to power management requests.



---

**IMPORTANT:** This document describes a power management framework that is device independent. For the Zynq® UltraScale+™ MPSoC, refer to the *Zynq UltraScale+ MPSoC: Software Developers Guide* (UG1137) [Ref 5] and the SDK documentation [Ref 6].

---

## Architecture and API Layers

EEMI assumes an architecture in which one or more processing clusters share resources, managed by a single power management controller.

APIs for the different functionality of the power management controller are in layers of APIs, as shown in the following figure.

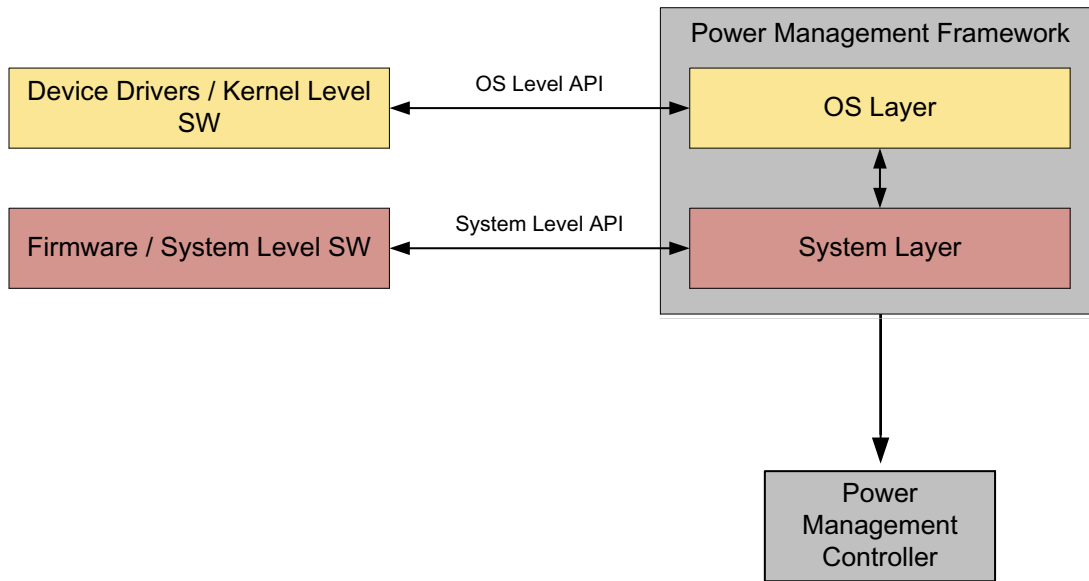


Figure 1-1: API Layers in the Power Management Framework

The layers are, as follows:

- **System-level:** These API allow low-level firmware to communicate directly with the power management controller. The system-level API consists of functions available to the clusters to send messages to the power management controller, as well as callback functions for the power management controller to send messages to the clusters. Additionally, there are API functions that are local to the clusters.
- **OS Layer:** These API allow the OS and its device drivers to communicate with the OS-layer of the power management framework.

Each layer of the API communicates the consolidated requirements from that layer down to the next API layer, until the system layer communicates with the power management controller.

The EEMI APIs are grouped into the following categories:

- Functions to manage suspending and waking clusters
- Functions to manage the power state of the slave devices, such as peripherals or memories
- Miscellaneous functions
- Direct-access functions

---

## APIs in Alphabetical Order

The following is a linked listing of the EEMI APIs from [Chapter 2, System-Level and OS-Level EEMI API Functions](#), in alphabetical order:

- [abort\\_suspend](#)
- [acknowledge\\_cb](#) Callback
- [force\\_powerdown](#)
- [get\\_api\\_version](#)
- [get\\_node\\_status](#)
- [get\\_op\\_characteristics](#)
- [init\\_suspend\\_cb](#) Callback
- [mmio\\_read](#)
- [mmio\\_write](#)
- [notify\\_cb](#) Callback
- [register\\_notifier](#)
- [reset\\_assert](#)
- [reset\\_get\\_status](#)
- [request\\_node](#)
- [release\\_node](#)
- [request\\_suspend](#)
- [request\\_wakeup](#)
- [set\\_wakeup\\_source](#)
- [set\\_max\\_latency](#)
- [set\\_requirement](#)
- [self\\_suspend](#)
- [set\\_wakeup\\_source](#)
- [system\\_shutdown](#)

# System-Level and OS-Level EEMI API Functions

---

## Introduction

Firmware on a cluster uses the EEMI APIs on the device. While the other API layers might only be available to clusters running an operating system, all clusters can make use of the system-level API.

For example, on an ARMv8 cluster, such as a Cortex®-A53 or a Cortex-A57, the system-level API is used both by the secure monitor in exception level 3 (EL3) as well as by the hypervisor in exception level 2 (EL2).

---

## Caller and Callee Responsibilities

Unless otherwise stated, the callers and callees bear the following responsibilities for any API call:

- The callers of the EEMI API functions are responsible for passing values in the arguments and keeping track of requested resources. Callback functions must be setup to receive acknowledge calls when requested.
- The callee is always responsible for verifying the calls and arguments, as well as the access permissions, making sure that a caller is authorized to perform the action requested. The callee is also responsible for providing acknowledgment as requested by the caller.

---

## Acknowledge Mechanism

Many of the EEMI API calls include an acknowledge argument for the caller to choose what kind of acknowledge they request.

- `REQUEST_ACK_NO`: No acknowledge is requested. The caller is not informed if the call was successful.
- `REQUEST_ACK_BLOCKING`: Blocking acknowledge is requested. The result is returned when the call is complete.
- `REQUEST_ACK_NONBLOCKING`: Non-blocking callback acknowledge is requested. The result is returned using the [acknowledge\\_cb Callback](#) function.

Unless explicitly stated otherwise, the acknowledge options are allowed for all APIs including an `ack` argument.

---

## Suspend APIs to Suspend Clusters

The API functions in this section are used by CPUs to either suspend themselves or to request suspend of other clusters or the entire system.

Available options are:

- [self\\_suspend](#)
- [request\\_suspend](#)
- [force\\_powerdown](#)
- [abort\\_suspend](#)
- [init\\_suspend\\_cb](#) Callback



## self\_suspend

**Description:** CPU declares a suspend or power-down. This function is used by a CPU to declare that it is about to suspend itself. When a CPU calls this function, the power management controller waits for the requesting CPU to inherit an idle state (for example: WFI for ARM). An architecture-specific mechanism (such as an interrupt) signals the entry of the idle state.

### Arguments:

- `node_id`: Suspend the CPU `node_id`. The `node_id` is required to identify the CPU node to be suspended when multiple CPU nodes within a cluster share a single communication channel.
- `latency`: Maximum wake-up latency requirement in  $\mu$ s. This value is considered by the power management controller when choosing the CPU power state.
- `state`: Instead of specifying a maximum latency, a CPU can also explicitly request a certain power state (as in the case of PSCI).
- `address`: Address to resume from when woken up.

### Returns:

`status`:

- On success, `EEMI_SUCCESS`.
- On failure, the appropriate error code from , [Error Codes](#). (Also, see [Acknowledge Mechanism](#)).

If both `latency` and `state` are specified, the callee should return `EEMI_INVALID_PARAM`.

**Caller Responsibility:** The caller must pass its own `node_id`; for example for SMP processors, it must determine the correct CPU instance. It is also responsible for ensuring that the resume address points to a legal entry point.

**Callee Responsibility:** The power management controller must keep track of the suspend request call for each CPU, until the CPU has either completed or aborted the suspend procedure, as more than one CPU can suspend themselves in parallel.

## request\_suspend

**Description:** Request suspend or power-down of another cluster. This call triggers the power management controller to notify the cluster identified by `node_id` that a suspend has been requested. This allows the cluster to gracefully suspend itself by calling `self_suspend()` for each of its CPU nodes, or else call `abort_suspend()` with its cluster node as argument and specify the reason. If an acknowledge is requested, the requesting cluster is notified upon completion of suspend or if an error occurred, such as an abort or a timeout.

### Arguments:

- `node_id`: The CPU `node_id` to be suspended. The `node_id` is required to identify the CPU node to be suspended when multiple CPU nodes within a cluster share a single communication channel.
- `latency`: Maximum wake-up latency requirement in  $\mu$ s. This value is considered by the power management controller when choosing the power state to put the CPU into.
- `state`: Instead of specifying a maximum latency, a CPU can also explicitly request a certain power state (as in the case of PSCI).
- `ack`: Acknowledge requested.

The `request_suspend()` sends a message to the processor to be suspended, through a callback; [init\\_suspend\\_cb Callback](#), which received by a processor, it can acknowledge that a cluster is requesting for it to suspend itself.

### Returns:

- `status`:
  - Returns the status of the operation, if `ack=REQUEST_ACK_BLOCKING`, (see [Acknowledge Mechanism](#)) Otherwise nothing is returned.
  - On success, `EEMI_SUCCESS`.
  - On failure, the appropriate error code from [Chapter 3, Error Codes](#).

If both `latency` and `state` are specified, the callee should return `EEMI_INVALID_PARAM`.

**Caller Responsibility:** None.

**Callee Responsibility:** The power management controller must verify that the `node_id` that is passed refers to another CPU or processing cluster. The controller must then invoke the `init_suspend` callback for that node.

## force\_powerdown

**Description:** This function can force a node or a cluster to be powered down. One cluster can request a forced power-off of another node. Can be used for an unresponsive cluster, in which case all resources of that cluster are released automatically.

### Arguments:

- `node_id`: The CPU `node_id` to be suspended. The `node_id` is required to identify the CPU node to be suspended when multiple CPU nodes within a cluster share a single communication channel.
- `ack`: Acknowledge requested.

### Returns:

- `status`:
  - Returns the status of the operation, if `ack=REQUEST_ACK_BLOCKING`, (see [Acknowledge Mechanism](#)). Otherwise nothing is returned.
  - On success, `EEMI_SUCCESS`.
  - On failure, the appropriate error code from , [Error Codes](#). (Also, see [Acknowledge Mechanism](#)).

**Caller Responsibility:** Force power-down can be requested by a cluster for itself, but a cold boot of that cluster is required for the cluster to resume operation again.

**Callee Responsibilities:** None.

## abort\_suspend

**Description:** This function is called by a node or CPU after a `self_suspend()` to notify the power management controller that CPU has aborted suspend, or by a cluster in response to an `init_suspend` request from the power management controller when the cluster refuses to suspend.

### Arguments:

- `reason`: The reason that the suspend cannot be performed or completed. The following are the arguments:
  - 100: Local wakeup-event received.
  - 101: Cluster is busy.
  - 102: No external power-down supported.
  - 103: Unknown error during suspend procedure.
- `node_id`: The `node_id` of the CPU or cluster node which is aborting the suspend.

### Returns:

- `status`:
  - On success, `EEMI_SUCCESS`.
  - On failure, the appropriate error code from , [Error Codes](#) (see [Acknowledge Mechanism](#)).

**Caller Responsibilities:** The caller must only call this function in one of two cases:

- During its self suspend request, such as after having called [self\\_suspend](#) but before completing the process.
- After receiving an [init\\_suspend\\_cb Callback](#) request, provided that a legitimate reason exists.

### Callee Responsibilities:

- If the abort suspend call is received during a CPU's suspend procedure, the power management controller must update its state machine accordingly.
- If the abort suspend is received as a response to a suspend request, the power management controller must return the `EEMI_ABORT_SUSPEND` to the CPU that originated the suspend request.

## init\_suspend\_cb Callback

**Description:** The callback function is implemented in each cluster, allowing the power management controller to request that the cluster suspend itself. If the cluster fails to act on this request, the power management controller or the requesting cluster can choose to force a power-down of the cluster.

### Arguments:

- `reason`: Suspend reason. the arguments are:
  - `SUSPEND_REASON_PU_REQ`
  - `SUSPEND_REASON_ALERT`
  - `SUSPEND_REASON_SYS_SHUTDOWN`
- `latency`: Maximum wake-up latency in  $\mu$ s. This information can be used by a CPU to decide what level of context saving might be required.
- `state`: Targeted sleep/suspend state.
- `timeout`: How much time in  $\mu$ s the cluster has to respond.

**Returns:** None.

**Caller Responsibilities:** None.

**Callee responsibilities:** The callee (in this case: the targeted CPU) must respond to the suspend request by either performing its own suspend procedure, or by calling the abort suspend function.

## Wake API Functions

The following are the available Wake API functions.

### request\_wakeup

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

#### Arguments:

- `node_id`: The CPU or cluster `node_id` to be powered or re-awakened.
- `set_address`: Specifies whether the start address argument is being passed; where:
  - `true`: Use address.
  - `false`: Ignore address.
- `address`: Address from which to resume when woken up; only used if `set_address = 1`.
- `ack`: Acknowledge requested.

#### Returns:

- `status`:
  - Returns the status of the operation, if `ack=REQUEST_ACK_BLOCKING`, (see [Acknowledge Mechanism](#)). Otherwise nothing is returned.
  - On success, `EEMI_SUCCESS`.
  - On failure, the appropriate error code from , [Error Codes](#). (Also, see [Acknowledge Mechanism](#)).

**Caller Responsibilities:** If the caller decides to pass a start address, it must ensure that the address passed refers to a legitimate entry point for the CPU in question.

**Callee Responsibilities:** The callee is responsible for passing the start address (if one is passed) to the CPU to be woken up with `set_wakeup_source`, which declares a node to be used as a wakeup source.

## set\_wakeup\_source

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

### Arguments:

- `target_id`: The ID of the target to be woken up.
- `node_id`: The `node_id` of the wakeup device.
- `enable`: Enable flag.
  - True: Enable wake source.
  - False: Disable wake source.

### Returns:

- `status`:
  - On success, `EEMI_SUCCESS`.
  - On failure, the appropriate error code from , [Error Codes](#), (Also, see [Acknowledge Mechanism](#)).

**Caller Responsibilities:** The caller is responsible for keeping the referenced node in a power state that will enable it to emit the required wake events, such as it must not release the node prior to going to sleep.

**Callee Responsibilities:** The callee must ensure that it can receive interrupts from the referenced node in order to power on the target node when necessary.

## system\_shutdown

**Description:** The `system_shutdown` interface provides methods to shut down or restart the system.

### Arguments:

- `type`: Shutdown type:
  - `EEMI_SHUTDOWN_TYPE_SHUTDOWN`: Shut down; the system is powered off permanently.
  - `EEMI_SHUTDOWN_TYPE_RESET`: Reset the system.
- `subtype`: Select the type of shutdown or reboot
  - `EEMI_SHUTDOWN_SUBTYPE_SUBSYSTEM`: Reset the subsystem associated with the calling master. Components that are not part of the calling subsystem must not be affected.
  - The implementation may specify additional subtypes.

### Returns:

- `status`:
  - On success, `EEMI_SUCCESS`.
  - On failure, the appropriate error code from [Error Codes](#) (see Acknowledge Mechanism).

**Caller Responsibilities:** The calling subsystem is responsible for shutting itself down through the *self-suspend* flow.

### Callee Responsibilities:

For types that affect more than the calling subsystem, the callee must do the following:

- Check whether the caller is authorized to initiate the selected type + subtype.
- Call registered `init_suspend()` callbacks.

If the requested action is permitted, the callee must shutdown or reset the targeted system (or sub-system) as requested once the caller completed its own suspend process.

When entering a shutdown state, all resources associated with the shutdown components shall be released.

- For a reset, the callee must put the system into the appropriate state to allow operating systems to be rebooted.



## APIs for Managing PM Slaves on a Cluster

The following are APIs that are available for managing slaves on a cluster.

### request\_node

**Description:** Request usage of a node. This function is used by a cluster to request the usage of a PM-slave.

#### Arguments:

- `node_id`: The requested PM slave `node_id`.
- `capabilities`: Slave-specific capabilities required.
- `qos`: Quality of Service (0-100) required, where 0 is minimum and 100 is maximum QoS.
- `ack`: Acknowledge requested.

#### Returns:

- `status`: Returns the status of the operation, if `ack=REQUEST_ACK_BLOCKING`, (see [Acknowledge Mechanism](#)). Otherwise nothing is returned.
- On success, `EEMI_SUCCESS`.
- On failure, the appropriate error code from [Error Codes](#). (Also, see [Acknowledge Mechanism](#)).

**Caller responsibilities:** The caller must keep track of the node it is requesting, because the node will not be automatically released when the caller suspends itself. The caller is also responsible for configuring and managing the device in question.

Calling [release\\_node](#) ensures that the device is kept in a power state providing the requested capabilities.




---

**RECOMMENDED:** *It is recommended that the caller uses the acknowledge feature to verify that it has actually been granted the requested node.*

---

**Callee responsibilities:** The power management controller must keep track of the users of a node, and deny requests for nodes that are currently in use, and cannot have more than one user.

## release\_node

**Description:** Release usage of a node. This function is used by a cluster to release the usage of a PM slave node. This instructs the power management controller that the node is no longer needed by that cluster, potentially allowing the node to be placed into an inactive state.




---

**IMPORTANT:** *If a cluster requires a certain wake-up latency for that node, then it must not call `release_node`, but instead use `set_requirement` and `set_max_latency`, to let the node enter a low-power state with a guaranteed maximum wakeup latency.*

---

### Arguments:

- `node_id`: The `node_id` of the PM slave node that is requested.

### Returns:

- `status`:
  - On success, `EEMI_SUCCESS`.
  - On failure, the appropriate error code from , [Error Codes](#). (Also, see [Acknowledge Mechanism](#)).

**Caller responsibilities:** The caller must know which nodes it has requested (including acknowledge). If a caller expects to be woken up by a device it must not call `release_node()` but use `set_requirement` instead.

**Callee responsibilities:** The power management controller must keep track of the current active users of a node and pick an appropriate power state for the node, satisfying the capabilities requested by all users. Whenever a node is powered off as a result of `release_node` call, the power management controller must also evaluate whether the parent power island or power domain can be turned off as well.

## set\_requirement

**Description:** Change the capability and/or the QoS requirements for a node in use. This function is used by a cluster 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 cluster calls [self\\_suspend](#), the requirement change is performed after the CPU signals the end of suspend to the power management controller, (for example: wait for interrupt (WFI) interrupt).

### Arguments:

- `node_id`: The `node_id` of the PM slave.
- `capabilities`: Slave-specific capabilities required.
- `qos`: Quality of Service (0-100) required, 100 being the highest QoS. (Not available).
- `ack`: Acknowledge requested.

### Returns:

- `status`: Returns the status of the operation, if `ack=REQUEST_ACK_BLOCKING`, (see [Acknowledge Mechanism](#)). Otherwise nothing is returned.
- On success, `EEMI_SUCCESS`.
- On failure, the appropriate error code from [Error Codes](#). (Also, see [Acknowledge Mechanism](#)).

**Caller responsibilities:** The caller must only call `set_requirement()` for those nodes that it has successfully called [request\\_node](#).

**Callee responsibilities:** The power management controller must only honor `set_requirement()` requests of current valid users of the identified node.

## set\_max\_latency

**Description:** Change the wakeup latency requirements for a node. This function is used by a cluster to announce a change in the maximum wake-up latency requirements for a specific slave node currently used by that cluster.



---

**TIP:** *Setting maximum wake-up latency can constrain the set of possible power states of a resource.*

---

### Arguments:

- `latency`: Maximum wake-up latency in microseconds required.
- `node_id`: The node ID of the slave.

### Returns:

- `status`:
  - On success, `EEMI_SUCCESS`.
  - On failure, the appropriate error code from [Error Codes](#). (Also, see [Acknowledge Mechanism](#)).

**Caller responsibilities:** Latency requirements can only be set for nodes that the caller uses; for example, the caller must have previously called `request_node()`.

**Callee responsibilities:** Only current users are allowed to assert latency requirements.

---

## Miscellaneous API Functions

### get\_boot\_status

**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.

**Returns:**

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

### get\_api\_version

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

**Arguments:** None.

**Returns:**

- `status`:
  - On success, `EEMI_SUCCESS`.
  - On failure, the appropriate error code from [Error Codes](#). (Also, see [Acknowledge Mechanism](#)).
- `api_version`: Implemented version of the EEMI.

**Caller responsibilities:** Not applicable.

**Callee responsibilities:** Not applicable.

## get\_node\_status

**Description:** Request information about current status. This function is used to obtain information about the operating state of a component.

### Arguments:

- `node_id`: ID of the component or sub-system in question.
- `nodestatus`: A structure as defined below.

### Returns:

- `status`:
  - On success, `EEMI_SUCCESS`.
  - On failure, the appropriate error code from [Error Codes](#). (Also, see [Acknowledge Mechanism](#)).

- `nodestatus`:

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:

- 0: If island is powered down.
- 1: If island is powered up.

For power domains: An integer value of the current voltage level in millivolts.

For PM slaves:

- 0: If slave is powered down.
  - 1: If slave is powered up.
  - 2: If slave is in retention.
- `requirements`: Slave nodes only: Returns current requirements the requesting cluster has requested for the node.

- usage: Slave nodes only: Current usage status of the node:
  - 0: Node is not used by any cluster.
  - 1: Node is used by caller exclusively.
  - 2: Node is used by other cluster(s) only.
  - 3: Node is used by caller and by other clusters.

**Caller responsibilities:** None.

**Callee responsibilities:** None.

## get\_op\_characteristics

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

### Arguments:

- `node_id`: ID of the component or sub-system in question.
- `type`: Type of operating characteristic requested. Common types of operating characteristics include—but are not limited to—power, energy, and temperature.

### Returns:

- `status`:
  - On success, `EEMI_SUCCESS`.
  - On failure, the appropriate error code from [Error Codes](#). (Also, see [Acknowledge Mechanism](#)).
- `result`: Depending on requested type:
  - `power`: Current power consumption in milliwatts.
  - `latency`: Current latency to return to active state in microseconds.
  - `temperature`: Current temperature in Celsius.

## register\_notifier

**Description:** Request notifications. A cluster 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 either for a specific event related to a specific node, a specific event related to any node, or any event related to a specific node.

### Arguments:

- `node_id`: ID of the component or sub-system in question.
- `event_id`: ID of the event state. The arguments are:
  - `EVENT_STATE`
  - `EVENT_ZERO_USERS`
  - `EVENT_ERROR_CONDITION`
- `wake`: Wake the component or sub-system. The arguments are:
  - `true`: Wake up on event.
  - `false`: Do not wake up (only notify if awake), no buffering or queueing.
- `enable`: Register boolean. The arguments are:
  - `true`: Register.
  - `false`: Unregister.

### Returns:

- `status`:
  - On success, `EEMI_SUCCESS`.
  - On failure, the appropriate error code from , [Error Codes](#). (Also, see [Acknowledge Mechanism](#)).

**Note:** Additional implementation-specific events can be added.

**Caller responsibilities:** The caller must ensure that a callback function is setup to handle incoming [notify\\_cb Callback](#) calls.

**Callee responsibilities:** The callee must keep track of all registered notifiers and emit notify calls as needed.



## acknowledge\_cb Callback

**Description:** Acknowledge callback. This function is called by the power management controller in response to any request where an acknowledge callback was requested.

**Arguments:**

- `node_id`: ID of the component or sub-system in question.
- `status`: Status of the operation:
  - `asserted`
  - `released`
- `node_state`: The current operating state of the component or sub-system in question.

**Returns:** None.

**Caller responsibilities:** None.

**Callee responsibilities:** None.

## notify\_cb Callback

**Description:** Notify callback. This function is called by the power management controller if an event has occurred for which the cluster was registered.

**Arguments:**

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

**Returns:** None.

**Caller responsibilities:** None.

**Callee responsibilities:** None.

---

## Direct Control API Functions

The following set of API functions allow access to resources that are not directly accessible due to bus access restrictions. The power management controller verifies access authorization.

### reset\_assert

**Description:** Call this function to assert or release reset on the specified reset line.

**Arguments:**

- `reset`: Identifier of the reset line.
- `assert`: Arguments:
  - `assert`: Assert the specified reset line.
  - `release`: Release the specified reset line.
  - `pulse`: Assert then release the specified reset line.

**Returns:**

- `status`:
  - On success, `EEMI_SUCCESS`.
  - On failure, the appropriate error code from , [Error Codes](#). (Also, see [Acknowledge Mechanism](#)).

**Caller responsibilities:** The caller must ensure that it is safe to call `reset` for the resource.

**Callee responsibilities:** The callee can assert the provided `reset` only if the caller is authorized.

## reset\_get\_status

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

**Arguments:**

- `reset`: Identifier of the reset line.

**Returns:**

- `status`:
  - On success, `EEMI_SUCCESS`.
  - On failure, the appropriate error code from , [Error Codes](#). (Also, see [Acknowledge Mechanism](#)).
- `reset_status`: Current state of `reset`, asserted or released.

**Caller responsibilities:** None.

**Callee responsibilities:** None.

## mmio\_write

**Description:** This function writes a value into a register that is not accessible directly, such as registers in a shared clock control unit. This call bypasses the power management logic.

**Arguments:**

- `address`: Physical address to which to write.
- `mask`: Bitmask used to limit the write to specific bits at the target address. Only the bits set in the mask are updated with the corresponding bit provided in value.
- `value`: Value to write to address.

**Returns:**

- `status`:
  - On success, `EEMI_SUCCESS`.
  - On failure, the appropriate error code from [Chapter 3, Error Codes](#). (Also, see [Acknowledge Mechanism](#)).

**Caller responsibilities:** The caller is responsible for making sure that it is safe to write the desired value to the register in question (for example: the device is in a state permitting accesses to its registers).

**Callee responsibilities:** The callee must only allow writes to addresses the caller is authorized to access. It is left up to the callee to identify unauthorized access attempts.

## mmio\_read

**Description:** Call this function to read a value from a register that is not directly accessible.

**Arguments:**

- `address`: Physical address from which to read.

**Returns:**

- `status`:
  - On success, `EEMI_SUCCESS`.
  - On failure, the appropriate error code from [Chapter 3, Error Codes](#). (Also, see [Acknowledge Mechanism](#)).
- `value`: Value read from address (only valid if status indicates success).

**Caller responsibilities:** The caller is responsible for making sure that it is safe to read from the desired register (the device is in a state permitting accesses to its registers).

**Callee responsibilities:** The callee must only allow reads from addresses the caller is authorized to access.

# Error Codes

## Error Codes

The following table lists the error codes used in the EEMI API.

*Table 3-1: EEMI 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	<code>request_node</code> : 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.



**TIP:** Error code values are documented in the “XilPM Library Reference” appendix of the Zynq UltraScale+ MPSoC: Software Developers Guide (UG1137) [Ref 5].

# Additional Resources and Legal Notices

---

## Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see the Xilinx® Support website at: [www.xilinx.com/support](http://www.xilinx.com/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

Zynq® UltraScale+™ MPSoC Device Documentation:

1. *Zynq UltraScale+ MPSoC Technical Reference Manual* ([UG1085](#))
2. *Zynq UltraScale+ MPSoC Register Reference* ([UG1087](#))
3. *Zynq Architecture System Monitor Guide* ([UG580](#))
4. *UltraFast Embedded Design Methodology Guide* ([UG1046](#))
5. *Zynq UltraScale+ MPSoC Software Developers Guide* ([UG1137](#))
6. [Xilinx Software Development Kit documentation](#)

---

## 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>.

© Copyright 2016-2017 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.