Embedded Energy Management Interface

EEMI API, v1.0 Reference Guide

UG1200 (v3.0) May 4, 2018

XILINX

Revision History

The following table shows the revision history for this document.

Section	Revision Summary
05/04/2018 Version 3.0	
General updates	Editorial updates and corrections.
Vivado ECO TCL Flow to Replace Existing Debug Probes	Added new section. You can use the Vivado TCL flow as an alternative to the GUI flow.

XILINX®

Table of Contents

XILINX

Chapter 1

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.

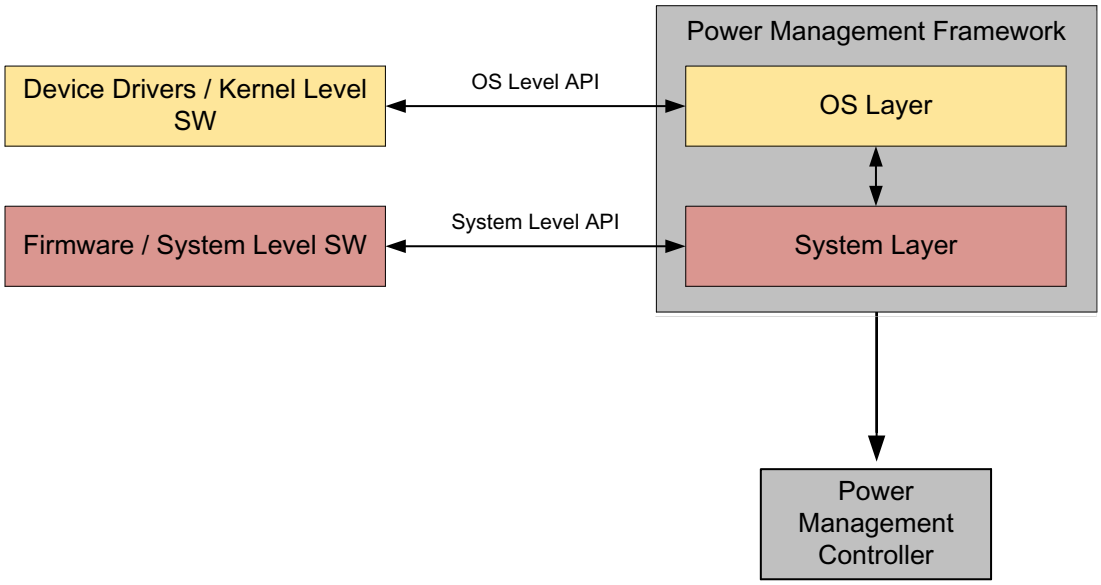

Figure 1-1: **API Layers in the Power Management Framework**

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
  - Clock control functions
  - Pin control 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
- clock_disable
- clock_enable
- clock_getdivider
- clock_getparent
- clock_getrate
- clock_getstate
- clock_setdivider
- clock_setparent
- clock_setrate
- force_powerdown
- get_api_version
- get_node_status
- get_op_characteristics
- init_finalize
- init_suspend_cb Callback
- ioctl
- mmio_read
- mmio_write
- notify_cb Callback
- pinctrl_get_config
- pinctrl_get_function
- pinctrl_request
- pinctrl_ release
- pinctrl_set_config
- pinctrl_set_function

- query_data
- register_notifier
- release_node
- request_node
- request_suspend
- request_wakeup
- reset_assert
- reset_get_status
- set_configuration
- set_max_latency
- set_requirement
- set_wakeup_source
- 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 Arm v8 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 µs. 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 µs. 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 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 CPUs 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 µs. 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 µs 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.

init_finalize

Description: Inform the power management controller that the caller master has initialized its own power management. Until a master calls init_finalize, the power management controller will keep all slaves which the master can use powered on. If a master doesn't have PM support, it will never call init_finalize, so the power management controller will always ensure that all slaves that the master can use remain powered up. This allows a PM-capable master to quickly start up without having to wait for all the saves to power up. It also allow for masters running applications that do not call any power management APIs.

Arguments: None.

Returns:

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

Caller responsibilities: Call this API when it has finished initializing the slaves.

Callee responsibilities: Keep all the slaves powered up until this API is called.

set_configuration

Description: This API 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

Arguments:

- `address`: Address to the configuration object.

Returns:

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

Caller responsibilities: The address must be accessible by the callee.

Callee responsibilities: Update the power management configuration.

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 queuing.
- `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.

query_data

Description: Request data from firmware. This function is used to obtain platform specific information from firmware like clocks, pins etc. Query parameter can be platform specific.

Arguments:

- `query id`: query parameter id to identify specific query

- `arg1`: argument1 specific to query operation
- `arg2`: argument2 specific to query operation
- `arg3`: argument3 specific to query operation

Returns:

- `status`:
  - On success, `EEMI_SUCCESS`.
  - On failure, the appropriate error code from Error Codes.
- `out`
  - Pointer to output data buffer based on type of query.

Caller responsibilities: Caller should provide valid arguments according to query ID and handle response based on type of query.

Callee responsibilities: Callee should validate parameters and provide requested data information in predefined response format.

ioctl

Description: This API can be used by a master to control any device specific configurations. Ioctl definitions can be platform specific. Use this API to manage shared device configurations.

Arguments:

- `node_id`: ID of the device to be controlled.
- `ioctl_id`: ID of control operation.
- `arg1`: argument1 specific to control operation
- `arg2`: argument2 specific to control operation

Returns:

- status
  - On success, `EEMI_SUCCESS`.
  - On failure, the appropriate error code from Error Codes.
- `out`
  - Output data based on type of control operation.

Caller responsibilities: Caller should provide valid arguments based on `ioctl` id and handle response accordingly.

Callee responsibilities: Callee should validate node id, `ioctl` id and its parameters, perform requested operation and provide response accordingly. Callee should only allow control operation if caller is authorized to access the control device.

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 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 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.

Clock Control API Functions

clock_enable

Description: Enable the requested clock. This function is used by a master to enable specific clock.

Arguments:

- `clock_id`: ID of the clock/pll to be enabled.

Returns:

- `status`
  - On success, `EEMI_SUCCESS`.
  - On failure, the appropriate error code from Error Codes.

Caller responsibilities: Caller should handle return code to verify and handle clock status. The caller should call `disable_clock` when not in use.

Callee responsibilities: Callee should make sure caller has authorized access to requested clock node. If not, it should deny the request. Callee must keep track of users of a particular

clock. Callee should enable clock only if clock is disabled. If clock type is PLL, callee needs to make sure that PLL gets locked after enabling it.

clock_disable

Description: Disable the requested clock. This function is used by a master to disable a specific clock.

Arguments:

- `clock_id`: ID of the clock/pll to be disabled.

Returns:

- `status:`
  - On success, `EEMI_SUCCESS`.
  - On failure, the appropriate error code from Error Codes.

Caller responsibilities: Caller can check return code to verify clock status

Callee responsibilities: Callee should make sure caller has authorized access to requested clock node. Callee should make sure that clock is not being used by any other user before disabling it. Callee should maintain usage count for particular clock.

clock_getstate

Description: Get state for given clock. This function is used by a master to query status of a specific clock.

Arguments:

- `clock_id`: ID of the clock.

Returns:

- `status:`
  - On success, `EEMI_SUCCESS`.
  - On failure, the appropriate error code from Error Codes.
- `state`
  - State of the clock (`enabled/disabled`).

Caller responsibilities: Not applicable.

Callee responsibilities: Callee should return current state of clock.

clock_setdivider

Description: Set divider for given clock node id. This function is used by a master to set divider for any clock to achieve desired rate.

Arguments:

- `clock_id`: ID of the clock.
- `divider`: Divider value.

Returns:

- `status`:
  - On success, `EEMI_SUCCESS`.
  - On failure, the appropriate error code from Error Codes.

Caller responsibilities: Caller should disable clock before calling this API. Caller is responsible to calculate proper divider value for desired rate. Caller should know parent and its rate to calculate accurate divider.

Callee responsibilities: Callee should make sure caller has authorized access to requested clock node. Callee should change rate only if clock is disabled. Callee should validate divider value before setting it.

clock_getdivider

Description: Get divider for given clock node id. This function is used by a master to get divider for any clock.

Arguments:

- `clock_id`: ID of the clock.

Returns:

- `status:`
  - On success, `EEMI_SUCCESS`.
  - On failure, the appropriate error code from Error Codes.
- `Divider`
  - divider value

Caller responsibilities: Caller should know parent and its rate to recalculate the rate for clock which divider belongs to.

Callee responsibilities: Callee should make sure caller has authorized access to requested clock node. Callee should properly calculate divider and return it.

clock_setrate

Description: Set rate for given clock node id. This function is used by a master to set rate for any clock.

Arguments:

- `clock_id`: ID of the clock.
- `rate`: Frequency rate value.

Returns:

- `status`:
  - On success, `EEMI_SUCCESS`.
  - On failure, the appropriate error code from Error Codes.

Caller responsibilities: Caller should disable clock before calling this API.

Callee responsibilities: Callee should make sure caller has authorized access to requested clock node. Callee should change rate only if clock is disabled. Callee should calculate divider value to achieve given rate based on parent rate.

clock_getrate

Description: Get rate for given clock. This function is used by a master to get current rate for any clock.

Arguments:

- `clock_id`: ID of the clock.

Returns:

- `status`:
  - On success, `EEMI_SUCCESS`.
  - On failure, the appropriate error code from Error Codes.
- `rate`
  - Frequency rate value.

Caller responsibilities: Not applicable.

Callee responsibilities: Callee should make sure caller has authorized access to requested clock node. Callee should read dividers and then recalculate the rate based on parent rate.

clock_setparent

Description: Set parent for given clock. This function is used by a master to set parent for any clock.

Arguments:

- `clock_id`: ID of the clock.
- `parent_id`: Parent index to be set.

Returns:

- status
  - On success, `EEMI_SUCCESS`.
  - On failure, the appropriate error code from Error Codes.

Caller responsibilities: Caller is responsible to recalculate dividers and set desired rate based on rate of new parent selected.

Callee responsibilities: Callee should make sure caller has authorized access to requested clock node. Callee should make sure parent clock is enabled.

clock_getparent

Description: Get parent for given clock node id. This function is used by a master to get parent for any clock.

Arguments:

- `clock_id`: ID of the clock.

Returns:

- status
  - On success, `EEMI_SUCCESS`.
  - On failure, the appropriate error code from Error Codes.
- `parent_id`
  - Parent index.

Caller responsibilities: Caller should identify parent clock based on returned parent index.

Callee responsibilities: Callee should make sure caller has authorized access to requested clock node.

Pin Control API Functions

pinctrl_request

Description: Request usage of pin. This API is used by a master to request usage of pin.

Arguments:

- `pin`: Pin number.

Returns:

- status:
  - On success, `EEMI_SUCCESS`.
  - On failure, the appropriate error code from Error Codes.

Caller responsibilities: Caller should request all pins before calling `set function/set config` for that pin. Once usage is finished, `release_pin` needs to be called.

Callee responsibilities: Callee should validate request against other users for same pin. If pin is already in use, request should be denied.

pinctrl_ release

Description: Inform PMU that Pin control is released.

Arguments:

- `pin`: Pin number.

Returns:

- `status`
  - On success, `EEMI_SUCCESS`.
  - On failure, the appropriate error code from Error Codes.

Caller responsibilities: Not applicable.

Callee responsibilities: Callee must keep track of all pin usage with respect to request and release calls.

pinctrl_set_function

Description: Set requested function for given pin.

Arguments:

- `pin`: Pin for which function is to be selected.
- `func_id`: Function to be set for given pin

Returns:

- `status`
  - On success, `EEMI_SUCCESS`.
  - On failure, the appropriate error code from Error Codes.

Caller responsibilities: Caller should request the pin before calling this API.

Callee responsibilities: Callee should make sure that pin is requested by caller and set function only if that function is supported by given pin.

pinctrl_get_function

Description: Get current selected function for given pin.

Arguments:

- `pin`: Pin for which function is to be selected

Returns:

- `status`
  - On success, `EEMI_SUCCESS`.
  - On failure, the appropriate error code from Error Codes.
- `func_id`
  - Function currently set for given pin.

Caller responsibilities: Not applicable.

Callee responsibilities: Not applicable.

pinctrl_get_config

Description: Get value of requested configuration parameter for given pin.

Arguments:

- `pin`: Pin for which configuration is to be read.
- `param`: Configuration parameter to be read.

Returns:

- `status`
  - On success, `EEMI_SUCCESS`.
  - On failure, the appropriate error code from Error Codes.
- `value`
  - Value of requested configuration parameter for the given pin.

Caller responsibilities: Not applicable.

Callee responsibilities: Not applicable.

pinctrl_set_config

Description: Set value of requested configuration parameter for given pin.

Arguments:

- `pin`: Pin for which configuration is to be set.
- `param`: Configuration parameter to be set.
- `value`: Value to be set for requested configuration parameter for the given pin.

Returns:

- `status`
  - On success, `EEMI_SUCCESS`.
  - On failure, the appropriate error code from Error Codes.

Caller responsibilities: Caller should request the pin before calling this API.

Callee responsibilities: Callee should make sure that pin is requested by callee. Callee should validate config value before setting it.

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	request_node: A processing cluster has already been assigned access to a PM slave and has issued a duplicate request for that PM slave.
EEMI_INVALID_NODE	The API function does not apply to the node passed as argument.
EEMI_NO_ACCESS	The processing cluster does not have access to the requested node or operation.
EEMI_ABORT_SUSPEND	The target processing cluster has aborted suspend.

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

XILINX

Appendix A

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.

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