Reservoir Labs

Power API
2015.01.21

Sponsored by:
Defense Advanced Research Projects Agency
Microsystems Technology Office (MTO)
Program: Power Efficiency Revolution for Embedded Computing Technologies (PERFECT)
Issued by DARPA/CMO under Contract No: HR0011-12-C-0123

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressly or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

Copyright © 2003-2015 Reservoir Labs, Inc.

Contents

Chapter 1

Reservoir Labs Power API

1.1 Overview

The Power API is divided into high-level and low-level interfaces. The high-level interface allows a programmer or compiler to specify power and energy management goals and choose strategies to achieve these goals. The implementation of strategies and the means to track and enforce power management goals is the responsibility of the implementor of the API on a platform.

The low-level interface of the Power API is close to the hardware and provides direct control over DVFS settings for voltage islands. It also provides access to energy probes, whenever they are available on the platform.

The high-level goals of the API are as follows:

- Provide a cross-platform interface for compilers and programmers to control and measure power and energy consumption
- Be concise, intuitive and make minimal assumptions about the underlying hardware
- Maintain a level of abstraction high enough to allow implementation in terms of DARPA PERFECT team APIs.
- Take advantage of features provided by leading edge task-based runtime environments

The API assumes that any system components bound to the same voltage and frequency settings are grouped together in an island. An island is the atomic unit for which frequency and voltage can be modified through the Power API. In the current library state, the energy is also measured at the granularity of the island, although this may change in the future.

The library is not multithread safe.

1.2 High-Level Interface

The high-level interface of the Power API is accessed through an initialization function and the data structures passed to this function.

The programmer (or compiler) must set up three things at Power API initialization:

- A model of hardware behavior
- A speed adjustment policy
- A scheduling policy

See Also

pwr_initialize()

Once configured, the combination of these 3 elements guides power and energy management decisions at program execution time. The 3 elements and associated data structures are described in the following sections. Power API implementations must define a default for each element on the targeted architecture.

1.2.1 Hardware Behavior

This defines the valid combinations of voltage and speed / speed level, possibly as functions of external factors such as the current temperature. Hardware behavior may be changed by hardware, software, or a combination of both.

Consider a near-threshold voltage architecture that may trade accuracy for power savings when voltage drops near threshold. An application that is resillient to errors would use a hardware behavior that allowed to use all voltage / frequency combinations supported by the architecture. An application that demands accurate results would use a hardware behavior that limited available voltage-frequency combinations to those well above threshold voltage.

Todo Clarify hardware behavior relationship with temperature / external factors.

Clarify definition of task, processing element.

1.2.2 Speed Policy

This defines a blanket policy for determining the speed level at which voltage islands are set.

Rather than exposing a notion of frequency, we expose a notion of speed level. This decision addresses the following:

- Permissible frequencies at discrete values
- Heterogeneity of architectures. Two different chips may have the same frequency but they won't have the same speed level (i.e., they won't execute the same piece of code in the same amount of time).

In the Power API, we define frequency, speed level, and speed:

- Frequency: The clock rate of a voltage island, in KHz
- Speed: A real number that corresponds to the absolute performance of a voltage island
- Speed Level: An integer greater than or equal to -1 that identifies a legal combination of voltage and frequency for a voltage island

See Also

speed_policies
pwr_num_speed_levels()
pwr_request_speed_level()
pwr_current_speed_level()
speed_policy_t

1.2.3 Scheduling Policy

This defines a blanket policy for the run-time assignment of tasks to processing elements.

See Also

scheduling_policies
scheduling_policy_t

Todo More options for scheduling policy. Specifically, space and time mapping.

1.3 Low-Level Interface

1.3.1 Speed Level

This interface allows the user to modify the speed level and voltage of an island. Speed level and voltage are not necessarily independent, so modifying one may modify the other.

A user is expected to be interested in upping the speed level or lowering the voltage knowing that the corresponding power consumption and speed will be negatively affected. An advanced user or compiler familiar with both island and application characteristics could modify speed and voltage in the same direction and still achieve power or performance gains. The goal of the low-level interface is to enable these modifications.

1.3.2 Energy Measurement

The low-level interface also exposes energy measurement capabilities. That capability allows energy profiling of programs and provides feedback to the user whenever frequency or voltage has been changed.

See Also

pwr_start_energy_count()
pwr_stop_energy_count()

1.3.3 Agility

Agility is defined as the best and worst case amount of time it takes to switch from one speed level to another on a voltage island.

See Also

pwr_agility()

1.4 Error checking

There are numerous situations in which hardware access is not possible. To detect those situations and help the user fixing potential configuration issues, the API provides convenient error checking. The last error that occured is stored in the library context and can be accessed either as a numeric error code or as a string for printing.

See Also

pwr_error()
pwr_strerror()

1.5 Example Code

```
#include <power_api.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[]) {
    long num_islands;
    pwr_ctx_t *ctx;

    // Initialize with default values
    ctx = pwr_initialize(NULL, NULL, NULL);

    // Get the number of islands
    num_islands = pwr_num_phys_islands(ctx);

    // Set each island to max speed
    for (long i = 0; i < num_islands; ++i) {
        unsigned int num_speed_lvls = pwr_num_speed_levels(ctx, i);

        // set maximum frequency
        pwr_request_speed_level(ctx, i, num_speed_lvls - 1);

        // check for errors
        if (pwr_error(ctx) != PWR_OK) {
            printf("Error while setting the frequency: %s\n",
                pwr_strerror(ctx));
        }
    }

    // Finalize
    pwr_finalize(ctx);

    return 0;
}
```

Author

M. Deneroff deneroff@reservoir.com
Benoit Meister meister@reservoir.com
Tom Henretty henretty@reservoir.com
Athanasios Konstantinidis konstantinidis@reservoir.com
Benoit Pradelle pradelle@reservoir.com

Chapter 2

Todo List

Global pwr_agility (pwr_ctx_t *ctx, unsigned long island, unsigned int from_level, unsigned int to_level)

Decide on unit for agility. Currently ns because of cpufreq default. If agility in cycles, at what speed level? Full speed? to_speed?

Global pwr_efficiency (pwr_ctx_t *ctx, unsigned long island, efficiency_t *efficiency)

IMPLEMENT

What time period to sample power over?

Is Joules / Flop the best unit?

Corresponding function for power efficiency?

Global pwr_increase_voltage (pwr_ctx_t *ctx, unsigned long island, int delta)

IMPLEMENT

Most architectures don't make this adjustment available. How to deal with this fact?

Global pwr_num_speed_levels (pwr_ctx_t *ctx, unsigned long island)

Map speed level to integer between 0 and 100 inclusive?

Global pwr_set_power_priority (pwr_ctx_t *ctx, void *task, int priority)

IMPLEMENT

How to represent tasks?

Create a corresponding register_task() method that returns a task id?

Take in a task_id for setting power and performance efficiency priorities?

Enforce `power_priority + performance_priority = 100`?

Global pwr_set_speed_priority (pwr_ctx_t *ctx, void *task, int priority)

IMPLEMENT

page Reservoir Labs Power API

Clarify hardware behavior relationship with temperature / external factors.

Clarify definition of task, processing element.

More options for scheduling policy. Specifically, space and time mapping.

Chapter 3

Module Documentation

3.1 Limitations

Describes various limitations imposed by the API.

Macros

- #define PWR_MAX_PHYS_CPU (1024∗1024)

  The maximum number of physical CPU in a system supported by the Power API.
- #define PWR_MAX_PHYS_ISLANDS (PWR_MAX_PHYS_CPU)

  The maximum number of physical voltage islands in a system supported by the Power API.
- #define PWR_MAX_VIRT_ISLANDS (PWR_MAX_PHYS_ISLANDS)

  The maximum number of virtual voltage islands in a system supported by the Power API.
- #define PWR_MAX_CPU_PER_PHYS_ISLAND (PWR_MAX_PHYS_CPU)

  The maximum number of CPU in a physical voltage island supported by the Power API.
- #define PWR_MAX_CPU_PER_VIRT_ISLAND (PWR_MAX_PHYS_CPU)

  The maximum number of CPU in a virtual voltage island supported by the Power API.
- #define PWR_MAX_SPEED_LEVELS (1024∗1024)

  The maximum number of speed levels supported by the Power API.

3.1.1 Detailed Description

3.2 Error Codes

List of all the possible error codes.

Macros

- #define PWR_ARCH_UNSUPPORTED (-3)

  Error: Feature unsupported by hardware.
- #define PWR_UNIMPLEMENTED (-2)

  Error: Feature not implemented.
- #define PWR_UNINITIALIZED (-1)

  Error: Power API has not been initialized.
- #define PWR_OK (0)

  Command executed successfully.
- #define PWR_ERR (1)

  Error: Unspecified error.
- #define PWR_UNAVAILABLE (2)

  Error: Feature temporarily unavailable.
- #define PWR_REQUEST_DENIED (4)

  Error: Request was denied.
- #define PWR_INIT_ERR (5)

  Error: Unspecified error during initialization.
- #define PWR_FINAL_ERR (6)

  Error: Unspecified error during finalization.
- #define PWR_ALREADY_INITIALIZED (7)

  Error: Attempt to initialize API after it has been initialized.
- #define PWR_IO_ERR (8)

  Error: Unspecified input / output error.
- #define PWR_UNSUPPORTED_SPEED_LEVEL (9)

  Error: Speed level not supported by hardware or API.
- #define PWR_UNSUPPORTED_VOLTAGE (10)

  Error: Voltage not supported by hardware or API.
- #define PWR_ALREADY_MINMAX (11)

  Error: Feature is already set to minimum or maximum value.
- #define PWR_OVER_E_BUDGET (12)

  Error: Request denied, over energy budget.
- #define PWR_OVER_P_BUDGET (13)

  Error: Request denied, over power budget.
- #define PWR_OVER_T_BUDGET (14)

Error: Request denied, over thermal budget.

- #define PWR_INVALID_ISLAND (15)

  Error: Specified island does not exist.

- #define PWR_DVFS_ERR (16)

  Error: Unspecified error when changing voltage and/or frequency.

3.2.1 Detailed Description

The context contains the last error that occurred. It can be retrived by pwr_error() or pwr_strerror().

3.3 Units

Typedefs

- typedef double efficiency_t

  Efficiency in Joules/Flop.
- typedef double voltage_t

  Voltage in Volts.
- typedef long speed_t

  Speed in UNDEFINED units.
- typedef long speed_level_t

  Unit-less speed level.
- typedef long freq_t

  Frequency in KHz.
- typedef long agility_t

  Agility in UNDEFINED units.
- typedef long power_t

  Power in Watts.
- typedef double energy_t

  Energy in Joules.

3.3.1 Detailed Description

3.4 Modules

Modules-related features.

Typedefs

- typedef unsigned int pwr_module_id_t

  A module identifier.

Enumerations

- enum pwr_module_id_t {
  PWR_MODULE_STRUCT = 0, PWR_MODULE_DVFS, PWR_MODULE_ENERGY, PWR_MODULE_HIGH_LEVEL,
  PWR_NB_MODULES }

  All the module ids.

3.4.1 Detailed Description

The functionalities in the API are provided by different modules. Every module has an id which uniquely identifies it. A module works independently from the others and some of them may be available at runtime while others may not be loaded.

See Also

pwr_is_initialized()

3.4.2 Enumeration Type Documentation

3.4.2.1 enum pwr_module_id_t

Enumerator

PWR_MODULE_STRUCT Hardware structure discovery.

PWR_MODULE_DVFS DVFS functions.

PWR_MODULE_ENERGY Energy measurement.

PWR_MODULE_HIGH_LEVEL High level interface.

PWR_NB_MODULES Number of existing modules.

Definition at line 379 of file power-api.h.

3.5 Initialization code

Functions

- bool pwr_is_initialized (const pwr_ctx_t *ctx, const pwr_module_id_t module)

  Checks if the Power API has been initialized.
- pwr_ctx_t * pwr_initialize (void *hw_behavior, void *speed_policy, void *scheduling_policy)

  Allocates resources used by the Power API.
- void pwr_finalize (pwr_ctx_t *ctx)

  Frees resources used by the Power API.
- int pwr_error (const pwr_ctx_t *ctx)

  Returns the last error code that was set on the given context.
- const char * pwr_strerror (const pwr_ctx_t *ctx)

  Returns a string describing the last error occuring in the library.

3.5.1 Detailed Description

3.5.2 Function Documentation

3.5.2.1 int pwr_error (const pwr_ctx_t * *ctx*)

Parameters

ctx	The current library context.

Returns

The code of the last error that occured.

3.5.2.2 void pwr_finalize (pwr_ctx_t * *ctx*)

Must be called after all Power API functions have been called. Reusing the context after calling that method leads to undefined results.

Parameters

ctx	The current library context.

3.5.2.3 pwr_ctx_t* pwr_initialize (void * *hw_behavior,* void * *speed_policy,* void * *scheduling_policy*)

Must be called before any other function in the Power API can be used.

Parameters

hw_behavior	Reserved for future use
speed_policy	Reserved for future use
scheduling_policy	Reserved for future use

Returns

A new context valid to be used by other Power API calls.

3.5.2.4 bool pwr_is_initialized (const pwr_ctx_t * *ctx,* const pwr_module_id_t *module*)

Parameters

ctx	The current library context
module	The id of the module to test

Returns

True if the given module has been correctly initialized, false otherwise.

3.5.2.5 const char* pwr_strerror (const pwr_ctx_t * *ctx*)

Parameters

ctx	The current library context

Returns

A string describing the last error that occured.

Chapter 4

Data Structure Documentation

4.1 phys_island_t Struct Reference

You are not supposed to directly access any of those fields.

`#include <internals.h>`

Data Fields

- unsigned long **num_cpu**
- unsigned long * **cpus**
- unsigned int **num_speed_levels**
- speed_level_t **current_speed_level**
- speed_level_t **min_speed_level**
- speed_level_t **max_speed_level**
- freq_t * **freqs**
- long **num_voltages**
- voltage_t * **voltages**
- voltage_t **current_voltage**
- agility_t **agility**

4.1.1 Detailed Description

Definition at line 43 of file internals.h.

4.2 pwr_ctx_t Struct Reference

You are not supposed to directly access any of those fields.

`#include <internals.h>`

Data Fields

- unsigned int **module_init**
- pwr_err_t **error**
- FILE * **err_fd**
- unsigned long **num_phys_cpu**
- unsigned long **num_phys_islands**
- phys_island_t ** **phys_islands**
- FILE ** **island_throttle_files**
- bool **emeas_running**
- pwr_emeas_t * **emeas**
- int **event_set**

4.2.1 Detailed Description

Definition at line 99 of file internals.h.

4.3 pwr_emeas_t Struct Reference

Energy measurement results.

`#include <energy.h>`

Data Fields

- double duration

  Execution time, in s.
- unsigned long nbValues

  How many values are profiled.
- long long * values

  Counter values.
- char ** names

  Counter names.
- char ** units

  Counter units.

4.3.1 Detailed Description

The structure is returned by pwr_stop_energy_count(). The result is made of an execution time and various hardware counter values. The number of hardware counters in the result depends on what is available on your machine. Only energy is measured by the counters, thus valid units are "J" and "nJ".

Definition at line 40 of file energy.h.

Chapter 5

File Documentation

5.1 dvfs.h File Reference

The file contains all the functions related to frequencies.

Functions

- unsigned int pwr_num_speed_levels (pwr_ctx_t *ctx, unsigned long island)

  Number of discrete speed levels supported by a voltage island.
- unsigned int pwr_current_speed_level (pwr_ctx_t *ctx, unsigned long island)

  The current speed level of a voltage island.
- void pwr_request_speed_level (pwr_ctx_t *ctx, unsigned long island, unsigned int new_level)

  Requests speed level change on a voltage island.
- void pwr_increase_speed_level (pwr_ctx_t *ctx, unsigned long island, int delta)

  Request a speed level modification of the given island.
- long pwr_agility (pwr_ctx_t *ctx, unsigned long island, unsigned int from_level, unsigned int to_level)

  Calculates the cost of switching speed levels.

5.1.1 Function Documentation

5.1.1.1 long pwr_agility (pwr_ctx_t * *ctx*, unsigned long *island*, unsigned int *from_level*, unsigned int *to_level*)

Todo Decide on unit for agility. Currently ns because of cpufreq default. If agility in cycles, at what speed level? Full speed? to_speed?

Parameters

ctx	The current library context.
island	The island of interest
from_level	Starting speed level
to_level	Finishing speed level

Returns

The agility value.

5.1.1.2 unsigned int pwr_current_speed_level (pwr_ctx_t * *ctx*, unsigned long *island*)

Parameters

ctx	The current library context.
island	The island to check speed level on.

Returns

The current speed level. The value returned is in range [0, num speed).

5.1.1.3 void pwr_increase_speed_level (pwr_ctx_t * *ctx*, unsigned long *island*, int *delta*)

`delta` can be positive or negative.

Parameters

ctx	The current library context.
island	The island to speed up or slow down
delta	The number of speed levels to increment by

5.1.1.4 unsigned int pwr_num_speed_levels (pwr_ctx_t * *ctx*, unsigned long *island*)

The slowest speed level is '`0`' and speed levels increase monotonically until the fastest speed level at '`num_speed_levels - 1`'.

Todo Map speed level to integer between 0 and 100 inclusive?

Parameters

ctx	The current library context.
island	The ID of the island to get speed level count for

Returns

The number of speed levels for that island.

5.1.1.5 void pwr_request_speed_level (pwr_ctx_t * *ctx*, unsigned long *island*, unsigned int *new_level*)

Requires `0 <= new_level < num_speed_levels`.

Parameters

ctx	The current library context.
island	The island to change speed level on
new_level	The requested speed level

5.2 dvfs.h

```
00001 /*
00002   * Copyright 2013-15 Reservoir Labs, Inc.
00003   *
00004   * Licensed under the Apache License, Version 2.0 (the "License");
00005   * you may not use this file except in compliance with the License.
00006   * You may obtain a copy of the License at
00007   *
00008   *     http://www.apache.org/licenses/LICENSE-2.0
00009   *
00010   * Unless required by applicable law or agreed to in writing, software
00011   * distributed under the License is distributed on an "AS IS" BASIS,
00012   * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
00013   * See the License for the specific language governing permissions and
00014   * limitations under the License.
00015   */
00016
00017 /**
00018  * @file
00019  *  The file contains all the functions related to frequencies.
00020  */
00021
00022 #ifndef __DVFS_H__
00023 #define __DVFS_H__
00024
00025 #ifndef __POWER_API_H__
00026     #error "Never directly include that file, rather use power_api.h"
00027 #endif
00028
00029 //============================================================================
00030 // Functions
00031 //----------------------------------------------------------------------------
00032
00033 /**
00034   * Number of discrete speed levels supported by a voltage island
00035   *
00036   * The slowest speed level is '<code>0</code>' and speed levels increase
00037   * monotonically until the fastest speed level at
00038   * '<code>num_speed_levels - 1</code>'.
00039   *
00040   * @todo Map speed level to integer between 0 and 100 inclusive?
```

```
00041   *
00042   * @param ctx The current library context.
00043   * @param island  The ID of the island to get speed level count for
00044   *
00045   * @return The number of speed levels for that island.
00046   */
00047 unsigned int pwr_num_speed_levels(pwr_ctx_t *ctx, unsigned long island);
00048
00049
00050 /**
00051   * The current speed level of a voltage island
00052   *
00053   * @param ctx The current library context.
00054   * @param island  The island to check speed level on.
00055   *
00056   * @return The current speed level. The value returned is in range [0, num speed).
00057   */
00058 unsigned int pwr_current_speed_level(pwr_ctx_t *ctx, unsigned long island);
00059
00060
00061 /**
00062   * Requests speed level change on a voltage island
00063   *
00064   * Requires <code>0 <= new_level < num_speed_levels</code>.
00065   *
00066   * @param ctx The current library context.
00067   * @param island  The island to change speed level on
00068   * @param new_level  The requested speed level
00069   */
00070 void pwr_request_speed_level(pwr_ctx_t *ctx, unsigned long island,
00071     unsigned int new_level);
00072
00073
00074 /**
00075   * Request a speed level modification of the given island
00076   *
00077   * <code>delta</code> can be positive or negative.
00078   *
00079   * @param ctx The current library context.
00080   * @param island  The island to speed up or slow down
00081   * @param delta  The number of speed levels to increment by
00082   */
00083 void pwr_increase_speed_level(pwr_ctx_t *ctx, unsigned long island, int
      delta);
00084
00085 /**
00086   * Calculates the cost of switching speed levels
00087   *
00088   * @todo Decide on unit for agility. Currently ns because of cpufreq default.
00089   *       If agility in cycles, at what speed level? Full speed? to_speed?
00090   *
00091   * @param ctx The current library context.
00092   * @param island  The island of interest
00093   * @param from_level  Starting speed level
00094   * @param to_level  Finishing speed level
00095   *
00096   * @return The agility value.
00097   */
00098 long pwr_agility(pwr_ctx_t *ctx, unsigned long island, unsigned int from_level,
00099     unsigned int to_level);
00100
00101 #endif
```

5.3 energy.h File Reference

This file contains the functions related to energy counters.

Data Structures

- struct pwr_emeas_t

  Energy measurement results.

Functions

- void pwr_start_energy_count (pwr_ctx_t *ctx)

  Starts the energy counters, measuring the current energy consumption.
- const pwr_emeas_t * pwr_stop_energy_count (pwr_ctx_t *ctx)

  Stops the energy counters and retrive the energy consummed since the last call to pwr_start_energy.

5.3.1 Function Documentation

5.3.1.1 void pwr_start_energy_count (pwr_ctx_t * *ctx*)

Parameters

ctx	The current library context.

5.3.1.2 const pwr_emeas_t* pwr_stop_energy_count (pwr_ctx_t * *ctx*)

Parameters

ctx	The current library context.

Returns

A pointer to energy measurement results.

5.4 energy.h

```
00001 /*
00002   * Copyright 2013-15 Reservoir Labs, Inc.
00003   *
00004   * Licensed under the Apache License, Version 2.0 (the "License");
```

```
00005   * you may not use this file except in compliance with the License.
00006   * You may obtain a copy of the License at
00007   *
00008   *     http://www.apache.org/licenses/LICENSE-2.0
00009   *
00010   * Unless required by applicable law or agreed to in writing, software
00011   * distributed under the License is distributed on an "AS IS" BASIS,
00012   * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
00013   * See the License for the specific language governing permissions and
00014   * limitations under the License.
00015   */
00016
00017 /**
00018  * @file
00019  * This file contains the functions related to energy counters.
00020  */
00021
00022 #ifndef __ENERGY_H__
00023 #define __ENERGY_H__
00024
00025 #ifndef __POWER_API_H__
00026     #error "Never directly include that file, rather use power_api.h"
00027 #endif
00028
00029 //====------------------------------------------------------------------------
00030 // Public data types
00031 //------------------------------------------------------------------------------
00032
00033 /**
00034  * Energy measurement results.
00035  * The structure is returned by pwr_stop_energy_count(). The result is made of
00036  * an execution time and various hardware counter values. The number of
00037  * hardware counters in the result depends on what is available on your machine.
00038  * Only energy is measured by the counters, thus valid units are "J" and "nJ".
00039  */
00040 typedef struct {
00041     double duration;        //!< Execution time, in s.
00042     unsigned long nbValues; //!< How many values are profiled
00043     long long *values;      //!< Counter values
00044     char **names;           //!< Counter names
00045     char **units;           //!< Counter units
00046 } pwr_emeas_t;
00047
00048
00049 //====------------------------------------------------------------------------
00050 // Public Functions
00051 //------------------------------------------------------------------------------
00052
00053 /**
00054  * Starts the energy counters, measuring the current energy consumption.
00055  *
00056  * @param ctx The current library context.
00057  */
00058 void pwr_start_energy_count(pwr_ctx_t *ctx);
00059
00060 /**
00061  * Stops the energy counters and retrive the energy consummed since the last
00062  * call to pwr_start_energy.
00063  *
00064  * @param ctx The current library context.
00065  *
00066  * @return A pointer to energy measurement results.
00067  */
00068 const pwr_emeas_t *pwr_stop_energy_count(
      pwr_ctx_t *ctx);
00069
00070 #endif
```

5.5 high_level.h File Reference

The file contains all the high level functions, mostly not implemented.

Functions

- void pwr_increase_voltage (pwr_ctx_t *ctx, unsigned long island, int delta)

  Requests a voltage level modification of the given island.
- void pwr_efficiency (pwr_ctx_t *ctx, unsigned long island, efficiency_t *efficiency)

  Current energy efficiency of an island (Joules / Flop)
- void pwr_set_power_priority (pwr_ctx_t *ctx, void *task, int priority)

  Sets the importance of power efficiency for the given task.
- void pwr_set_speed_priority (pwr_ctx_t *ctx, void *task, int priority)

  Sets the importance of performance for the given task.

5.5.1 Function Documentation

5.5.1.1 void pwr_efficiency (pwr_ctx_t * *ctx*, unsigned long *island*, efficiency_t * *efficiency*)

Todo IMPLEMENT

What time period to sample power over?

Is Joules / Flop the best unit?

Corresponding function for power efficiency?

Parameters

ctx	The current library context.
island	The voltage island to calculate efficiency for
efficiency[out]	The power efficiency of the island in Joules / Watt

5.5.1.2 void pwr_increase_voltage (pwr_ctx_t * *ctx*, unsigned long *island*, int *delta*)

`delta` can be positive or negative.

Todo IMPLEMENT

Most architectures don't make this adjustment available. How to deal with this fact?

Parameters

ctx	The current library context.
island	The island to change voltage on
delta	The number of voltage levels to increment by

5.5.1.3 void pwr_set_power_priority (pwr_ctx_t * *ctx*, void * *task*, int *priority*)

Todo IMPLEMENT

How to represent tasks?

Create a corresponding register_task() method that returns a task id?

Take in a task_id for setting power and performance efficiency priorities?

Enforce `power_priority + performance_priority = 100`?

Parameters

ctx	The current library context.
task	A task managed by the Power API
priority	An integer indicating the importance of power efficiency for the given task. Possible values are between 0 (power efficiency is lowest priority) and 100 (power efficiciency is highest priority) inclusive.

5.5.1.4 void pwr_set_speed_priority (pwr_ctx_t * *ctx*, void * *task*, int *priority*)

Todo IMPLEMENT

Parameters

ctx	The current library context.
task	A task managed by the Power API
priority	An integer indicating the importance of performance for the given task. Possible values are between 0 (power efficiency is lowest priority) and 100 (power efficiciency is highest priority) inclusive.

5.6 high_level.h

```
00001 /*
00002   * Copyright 2013-15 Reservoir Labs, Inc.
00003   *
00004   * Licensed under the Apache License, Version 2.0 (the "License");
00005   * you may not use this file except in compliance with the License.
00006   * You may obtain a copy of the License at
00007   *
00008   *     http://www.apache.org/licenses/LICENSE-2.0
00009   *
00010   * Unless required by applicable law or agreed to in writing, software
00011   * distributed under the License is distributed on an "AS IS" BASIS,
00012   * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
00013   * See the License for the specific language governing permissions and
00014   * limitations under the License.
00015   */
00016
00017 /**
00018  * @file
00019  *  The file contains all the high level functions, mostly not implemented.
00020  */
00021
00022 #ifndef __HIGH_LEVEL_H__
00023 #define __HIGH_LEVEL_H__
00024
00025 #ifndef __POWER_API_H__
00026     #error "Never directly include that file, rather use power_api.h"
00027 #endif
00028
00029 /**
00030   * Requests a voltage level modification of the given island
00031   *
00032   * <code>delta</code> can be positive or negative.
00033   *
00034   * @todo IMPLEMENT
00035   * @todo Most architectures don't make this adjustment available. How to deal
00036   *       with this fact?
00037   *
00038   * @param ctx The current library context.
00039   * @param island  The island to change voltage on
00040   * @param delta  The number of voltage levels to increment by
00041   */
00042 void pwr_increase_voltage(pwr_ctx_t *ctx, unsigned long island, int delta);
00043
00044 /**
00045   * Current energy efficiency of an island (Joules / Flop)
00046   *
00047   * @todo IMPLEMENT
00048   * @todo What time period to sample power over?
00049   * @todo Is Joules / Flop the best unit?
00050   * @todo Corresponding function for power efficiency?
00051   *
00052   * @param ctx The current library context.
00053   * @param island  The voltage island to calculate efficiency for
00054   * @param efficiency[out]  The power efficiency of the island in Joules / Watt
00055   */
00056 void pwr_efficiency(pwr_ctx_t *ctx, unsigned long island,
00057                     efficiency_t* efficiency);
00058
00059 /**
00060   * Sets the importance of power efficiency for the given task
00061   *
00062   * @todo IMPLEMENT
00063   * @todo How to represent tasks?
00064   * @todo Create a corresponding register_task() method that returns a task id?
00065   * @todo Take in a task_id for setting power and performance efficiency
```

```
00066   *       priorities?
00067   * @todo Enforce <code>power_priority + performance_priority = 100</code>?
00068   *
00069   * @param ctx The current library context.
00070   * @param task  A task managed by the Power API
00071   * @param priority  An integer indicating the importance of power efficiency
00072   *                  for the given task. Possible values are between 0
00073   *                  (power efficiency is lowest priority) and 100 (power
00074   *                  efficiciency is highest priority) inclusive.
00075   */
00076 void pwr_set_power_priority(pwr_ctx_t *ctx, void* task, int priority);
00077
00078 /**
00079   * Sets the importance of performance for the given task
00080   *
00081   * @todo IMPLEMENT
00082   *
00083   * @param ctx The current library context.
00084   * @param task  A task managed by the Power API
00085   * @param priority  An integer indicating the importance of performance
00086   *                  for the given task. Possible values are between 0
00087   *                  (power efficiency is lowest priority) and 100 (power
00088   *                  efficiciency is highest priority) inclusive.
00089   */
00090 void pwr_set_speed_priority(pwr_ctx_t *ctx,void* task, int priority);
00091
00092
00093 #endif
00094
```

5.7 internals.h File Reference

The file contains all the fields and functions used internally.

```
#include <glib.h>
#include <stdbool.h>
#include <stddef.h>
#include <stdio.h>
#include "power-api.h"
```

Data Structures

- struct phys_island_t

  You are not supposed to directly access any of those fields.
- struct pwr_ctx_t

  You are not supposed to directly access any of those fields.

Typedefs

- typedef int pwr_err_t

  Error codes returned by API functions.

Functions

- GString * **sysfs_filename** (unsigned long cpu_id, const char *filename)
- void **init_struct_module** (pwr_ctx_t *ctx)
- void **free_structure_data** (pwr_ctx_t *ctx)
- void **init_speed_levels** (pwr_ctx_t *ctx)
- void **free_speed_data** (pwr_ctx_t *ctx)
- void **init_energy** (pwr_ctx_t *ctx)
- void **free_energy_data** (pwr_ctx_t *ctx)

5.7.1 Detailed Description

Never directly use them unless you really know what you are doing.

Definition in file internals.h.

5.8 internals.h

```
00001 /*
00002   * Copyright 2013-15 Reservoir Labs, Inc.
00003   *
00004   * Licensed under the Apache License, Version 2.0 (the "License");
00005   * you may not use this file except in compliance with the License.
00006   * You may obtain a copy of the License at
00007   *
00008   *     http://www.apache.org/licenses/LICENSE-2.0
00009   *
00010   * Unless required by applicable law or agreed to in writing, software
00011   * distributed under the License is distributed on an "AS IS" BASIS,
00012   * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
00013   * See the License for the specific language governing permissions and
00014   * limitations under the License.
00015   */
00016
00017 /**
00018  * @file
00019  *  The file contains all the fields and functions used internally.
00020  *  Never directly use them unless you really know what you are doing.
00021  */
00022
00023 #include <glib.h>
00024 #include <stdbool.h>
00025 #include <stddef.h>
00026 #include <stdio.h>
00027
00028 #include "power-api.h"
00029
00030 //====-------------------------------------------------------------------------
00031 // Typedefs
00032 //----------------------------------------------------------------------------
00033
00034 /** Error codes returned by API functions */
00035 typedef int pwr_err_t;
00036
00037 //====-------------------------------------------------------------------------
00038 // Internal types for the modules
```

```
00039 //----------------------------------------------------------------------------
00040
00041 /** You are not supposed to directly access any of those fields */
00042 /* Holds information related to a physical voltage island */
00043 typedef struct phys_island {
00044     /* The count of CPU */
00045     unsigned long num_cpu;
00046
00047     /* IDs of the CPU */
00048     unsigned long *cpus;
00049
00050     /* Total number of speed levels supported */
00051     unsigned int num_speed_levels;
00052
00053     /*
00054       * Nominal speed level set by Power API clients, may be overridden by
00055       * hardware PMU
00056       */
00057     speed_level_t current_speed_level;
00058
00059     /* Minimum available speed level */
00060     speed_level_t min_speed_level;
00061
00062     /* Maximum available speed level */
00063     speed_level_t max_speed_level;
00064
00065     /*
00066       * Maps speed levels to physical frequencies, i.e.
00067       * <code>freqs[speed_level] = frequency</code>
00068       */
00069     freq_t* freqs;
00070
00071     /*
00072       * If supported by hardware, the total number of voltages this island can
00073       * be set at
00074       */
00075     long num_voltages;
00076
00077     /* Voltages supported by this island */
00078     voltage_t* voltages;
00079
00080     /*
00081       * Nominal voltage as set by Power API client, may be overridden by
00082       * hardware PMU
00083       */
00084     voltage_t current_voltage;
00085
00086     /*
00087       * Worst case time to transition from on frequency / voltage to another
00088       */
00089     agility_t agility;
00090 } phys_island_t;
00091
00092
00093 //====-------------------------------------------------------------------------
00094 // Private structures shared across all modules
00095 //----------------------------------------------------------------------------
00096
00097 /** You are not supposed to directly access any of those fields */
00098 /* Generic structure to store the current library status */
00099 typedef struct pwr_ctx {
00100     /* bitfield to determine if a module was initialized */
00101     unsigned int module_init;
00102
00103     /* The last error that occurred */
00104     pwr_err_t error;
00105
00106     /* Where to write the error messages. Can be NULL to print no message. */
```

```
00107     FILE *err_fd;
00108
00109     /* --- structure module --- */
00110
00111     /* How many physical CPU are in the system? */
00112     unsigned long num_phys_cpu;
00113
00114     /* How many physical voltage islands are in the system? */
00115     unsigned long num_phys_islands;
00116
00117     /* Physical power islands on the system */
00118     phys_island_t **phys_islands;
00119
00120     /* --- DVFS module --- */
00121
00122     /* File pointers fpr sysfs frequency control files, one per CPU */
00123     FILE** island_throttle_files;
00124
00125     /* --- Power measurements --- */
00126
00127     /* Are we measuring energy right now? */
00128     bool emeas_running;
00129
00130     /* Last measurement */
00131     pwr_emeas_t *emeas;
00132
00133     /* Event set identifier (used by PAPI) */
00134     int event_set;
00135 } pwr_ctx_t;
00136
00137
00138 //====--------------------------------------------------------------------------
00139 // Private functions shared across all modules
00140 //----------------------------------------------------------------------------
00141
00142
00143 // ###### General functions ######
00144
00145
00146 /*
00147   * Builds a filename of the form
00148   * <code>/sys/devices/system/cpu/cpu_id/cpufreq/filename</code>
00149   *
00150   * @param cpu_id The unique identifier of the cpu to build a filename for
00151   * @param filename The cpufreq file to use in the filename
00152   *
00153   * @return A sysfs filename as a <code>GString</code>
00154   */
00155 GString* sysfs_filename(unsigned long cpu_id, const char* filename);
00156
00157
00158 // ###### Structure functions ######
00159
00160
00161 /*
00162   * Sets the number of physical voltage islands in the system and creates a
00163   * struct for each.
00164   *
00165   * @param ctx The current library context.
00166   */
00167 void init_struct_module(pwr_ctx_t *ctx);
00168
00169 /*
00170  * Frees the memory used to store structural information.
00171  */
00172 void free_structure_data(pwr_ctx_t *ctx);
00173
00174
```

```
00175 // ###### Frequencies-related functions ######
00176
00177
00178 /*
00179   * Sets the speed levels and frequencies for each voltage island
00180   *
00181   * @param ctx The current library context.
00182   */
00183 void init_speed_levels(pwr_ctx_t *ctx);
00184
00185 /*
00186  * Frees the memory used to store speed information.
00187  *
00188  * @return PWR_OK.
00189  */
00190 void free_speed_data(pwr_ctx_t *ctx);
00191
00192
00193 // ###### Energy-related functions ######
00194
00195
00196 /*
00197   * Allocate resources used to collect energy counter data
00198   *
00199   * @param ctx The current library context.
00200   */
00201 void init_energy(pwr_ctx_t *ctx);
00202
00203 /*
00204   * Release resources used to gather energy counter data
00205   */
00206 void free_energy_data(pwr_ctx_t *ctx);
00207
```

5.9 structure.h File Reference

The file contains all the functions related to hardware structure queries.

Functions

- unsigned long pwr_num_phys_cpus (pwr_ctx_t *ctx)

  The number of actual CPU controlled by the Power API.
- unsigned long pwr_num_phys_islands (pwr_ctx_t *ctx)

  Get the number of voltage islands controlled by the Power API.
- unsigned long pwr_island_of_cpu (pwr_ctx_t *ctx, unsigned long cpu)

  Returns the id of the island that contains the given CPU.

5.9.1 Function Documentation

5.9.1.1 unsigned long pwr_island_of_cpu (pwr_ctx_t * *ctx*, unsigned long *cpu*)

Parameters

ctx	The current API context.
cpu	The Linux id of the CPU whose island is searched for.

Returns

The identifier of the island that contains the given CPU core.

5.9.1.2 unsigned long pwr_num_phys_cpus (pwr_ctx_t * *ctx*)

Parameters

ctx	The current library context

Returns

How many CPUs are controlled by the library.

5.9.1.3 unsigned long pwr_num_phys_islands (pwr_ctx_t * *ctx*)

The islands can then be addressed using a number in [0, nb islands).

Parameters

ctx	The current library context.

Returns

The number of physical islands available on the machine.

5.10 structure.h

```
00001 /*
00002   * Copyright 2013-15 Reservoir Labs, Inc.
00003   *
00004   * Licensed under the Apache License, Version 2.0 (the "License");
00005   * you may not use this file except in compliance with the License.
00006   * You may obtain a copy of the License at
00007   *
00008   *     http://www.apache.org/licenses/LICENSE-2.0
00009   *
00010   * Unless required by applicable law or agreed to in writing, software
00011   * distributed under the License is distributed on an "AS IS" BASIS,
00012   * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
00013   * See the License for the specific language governing permissions and
00014   * limitations under the License.
```

```
00015   */
00016
00017 /**
00018  * @file
00019  *  The file contains all the functions related to hardware structure queries.
00020  */
00021
00022 #ifndef __STRUCTURE_H__
00023 #define __STRUCTURE_H__
00024
00025 #ifndef __POWER_API_H__
00026     #error "Never directly include that file, rather use power_api.h"
00027 #endif
00028
00029 //====--------------------------------------------------------------------------
00030 // Public Functions
00031 //----------------------------------------------------------------------------
00032
00033 /**
00034   * The number of actual CPU controlled by the Power API.
00035   *
00036   * @param ctx The current library context
00037   *
00038   * @return How many CPUs are controlled by the library.
00039   */
00040 unsigned long pwr_num_phys_cpus(pwr_ctx_t *ctx);
00041
00042 /**
00043  * Get the number of voltage islands controlled by the Power API. The islands
00044  * can then be addressed using a number in [0, nb islands).
00045  *
00046  * @param ctx The current library context.
00047  *
00048  * @return The number of physical islands available on the machine.
00049  */
00050 unsigned long pwr_num_phys_islands(pwr_ctx_t *ctx);
00051
00052 /**
00053  * Returns the id of the island that contains the given CPU.
00054  *
00055  * @param ctx The current API context.
00056  * @param cpu The Linux id of the CPU whose island is searched for.
00057  *
00058  * @return The identifier of the island that contains the given CPU core.
00059  */
00060 unsigned long pwr_island_of_cpu(pwr_ctx_t *ctx, unsigned long cpu);
00061
00062 #endif
00063
```

Index