High Performance Computing - Power Application Programming Interface Specification Version 1.0

Chair: Ryan E. Grant[1]
Editor: Barry Rountree[2]
Secretary: Jeff Hanson[3]

Contributors:

Chris Cantalupo[4], Jonathan Eastep[4],
Scott Hara[5], Siddhartha Jana[4],
Jaymin Jasoliya[4], Matthew Kappel[6],
James H. Laros III[1], Steve Leak [7],
Michael Levenhagen[1], Steve Martin[6],
Ramakumar Nagappan[4], Kevin Pedretti[1],
Todd Rosedahl[8], Andy Warner[3], Andrew Younge[1]

November 2019

Contributing Organizations:
[1] Sandia National Laboratories
[2] Lawrence Livermore National Laboratory
[3] Hewlett Packard Enterprise (HPE)
[4] Intel
[5] Qualcomm
[6] Cray
[7] NERSC
[8] IBM

Abstract

Measuring and controlling the power and energy consumption of high performance computing systems by various components in the software stack is an active research area [14, 3, 5, 11, 4, 22, 20, 17, 7, 18, 21, 19, 12, 1, 6, 15, 13]. Implementations in lower level software layers are beginning to emerge in some production systems, which is very welcome. To be most effective, a portable interface to measurement and control features would significantly facilitate participation by all levels of the software stack. We present a proposal for a standard power Application Programming Interface (API) that endeavors to cover the entire software space, from generic hardware interfaces to the input from the computer facility manager.

Chapter 1

Acknowledgment

The Power API Community Specification is managed via the Power API Committee, an open specifications body operating under the Energy-Efficient High Performance Working Group (EEHPC-WG). The community version of the specification was developed based on the Power API Specification, originally developed at Sandia National Laboartories and supported through the Advanced Simulation and Computing (ASC) program funded by U.S. Department of Energy's National Nuclear Security Agency. The Sandia developed Power API released up to version 2.0, this document build upon that work starting over at Community Version 1.0.

The Sandia National Laboratories version of the specification was retired to support the community-lead version. The original specification can be found at `powerapi.sandia.gov`.

The original publication describing the design and operation of the Power API [8] is: Grant, R.E., Levenhagen, M., Olivier, S.L., DeBonis, D., Pedretti, K.T. and Laros III, J.H., 2016. Standardizing power monitoring and control at exascale. Computer, 49(10), pp.38-46.

We wish to thank our colleagues, Steve Hammond, Ryan Elmore, and Kris Munch at the National Renewable Energy Laboratory (NREL) for their contributions to the use case model which was the progenitor of this work. This effort was greatly enhanced by interactions with staff throughout Sandia as well as many external organizations.

The addition of the Python language bindings in version 2.0 of the Power API specification would not have been possible without contributions from Steve Martin (Cray), Matthew Kappel (Cray) and Leo Maurer (Cray), Paul Falde (Cray) and valuable feedback from Johnathan Woodring (Los Alamos

National Laboratory)

Feedback and additions to the application hints interface were provided by Chris Cantalupo and Steve Sylvester of Intel.

The following individuals contributed to the specification during the version 1.X series: Sue Kelly (Sandia National Laboratories) and David DeBonis (Sandia National Laboratories).

Prior to the first open release of this specification a select group of individuals agreed to review an early draft of the specification and provide feedback. We would like to recognize the very significant contributions these individuals made and thank them for their time and efforts. The following individuals participated in an all day face-to-face review of the specification and provided written feedback (listed in alphabetical order): David Jackson (Adaptive Computing), Steve Martin (Cray), Indrani Paul (AMD), Phil Pokorny (Penguin Computing), Avi Purkayastha (National Renewable Energy Laboratory), Muralidhar Rajappa (Intel), and Jeff Stuecheli (IBM). The following individuals provided written feedback of the specification (listed in alphabetical order): Dorian Arnold (University of New Mexico), Natalie Bates (EEHPC), and Chung-Hsing Hsu (Oak Ridge National Laboratory). We hope to continue these important collaborations and develop new ones in an effort to represent and serve the HPC community as best we can.

Contents

Chapter 2

Introduction

Achieving practical exascale supercomputing will require massive increases in energy efficiency. The bulk of this improvement will likely be derived from hardware advances such as improved semiconductor device technologies and tighter integration, hopefully resulting in more energy efficient computer architectures. Still, software will have an important role to play. With every generation of new hardware, more power measurement and control capabilities are exposed. Many of these features require software involvement to maximize feature benefits. This trend will allow algorithm designers to add power and energy efficiency to their optimization criteria. Similarly, at the system level, opportunities now exist for energy-aware scheduling to meet external utility constraints such as time of day cost charging and power ramp rate limitations. Finally, future architectures might not be able to operate all components at full capability for a range of reasons including temperature considerations or power delivery limitations. Software will need to make appropriate choices about how to allocate the available power budget given many, sometimes conflicting considerations.

For these reasons, we have developed a portable API for power measurement and control. This Power API provides multiple levels of abstractions to satisfy the requirements of multiple types of users [10]. The remainder of this document describes the details of this Power API specification.

2.1 Background

We draw our inspiration from efforts such as the MPI forum's[1] process. We seek to develop a de facto standard, led by a neutral national laboratory, which is funded by a neutral federal agency. Community involvement is critical to the effort. The laboratory team has been garnering participation by making presentations at workshops and operational group meetings. We desire community participation from university and other researchers, as well as HPC practitioners. Concurrent with the specification development, the authors are creating a reference implementation comprising a subset of the overall API functionality. This task is important to ensure that the specification is usable. The ultimate goal, however, is that vendors of the hardware and software components provide their own implementations. It is likely that some portion of these functions have already been written by vendors, but with slightly different calling arguments. For portability sake, we are hopeful that the specific implementations can be melded to this proposed community API.

2.2 Motivation

The introductory paragraph above, offers a few examples where a Power API would be useful. This document's abstract provides references to a small subset of the current research activities that would benefit from a community-adopted power API. Additional, more fleshed out examples are described in the appendices of the *Power/Energy Use Cases for High Performance Computing* document [10]. To provide the proper mindset for reading this document, we offer the following list as well.

- A job is entering a checkpoint phase. The application requests a reduced processor frequency during the long I/O period.
- A developer is trying to understand frequency sensitivity of an algorithm and starts a tool that analyzes performance and power consumption while the job is running.
- Once an application's power signature is analyzed, future job submissions give power hints to the resource manager.

[1] http://www.mpi-forum.org

- A data center has a maximum of capacity of nn MW. One HPC system is down for extended maintenance. Other systems can have a higher maximum power cap.

- For electric bills based on peak usage periods, determine a maximum HPC load that minimizes loss of HPC use. Then direct the scheduler to enforce that peak usage.

2.3 Use Case Development

The *Power/Energy Use Cases for High Performance Computing* document [10] identifies the requirements for the Power API. Rather than a list, the requirements are specified as formal use cases employing the ISO/IEC 19501:2005 Unified Modeling Language (UML) standard, which is described in the reference manual by Booch, et al. [2]. While the term use case has come to be almost synonymous with scenario, the standard defines a use case *model*. The use case model does include scenario-like requirement specifications, but it also clearly identifies the roles and scope of the requirements. For this document, the key concepts from the use case model are *actor* and *system*. Each identified actor plays a distinct role in using the power API. Actors can be persons, other systems, or something else (e.g. cron, asynchronous event, etc.). For the Power API use case model, an HPC computer is broken down into logical systems. By breaking down the requirements into this use case model, we can clearly see the demarcation points requiring an API between external actors and each system. And by subsequently viewing systems as actors to the other systems, we obtain the complete set of necessary interfaces.

The specific actor/system pairs used for the power API are shown in Figure 2.1. The external actors are shown on the left portion of the diagram. Systems are shown as rectangles. The four systems conjoined with the actor symbol also serve as actors for some use cases. The ten sections within Chapter 7 provide function specifications for the ten actor/system pairs (Role/System pairs in the specification). The two missing interfaces are Facility Manager to Facility Hardware and Facility Manager to HPCS manager. These were included in the use case model to identify the boundaries of the specification and recognize important points of information input.

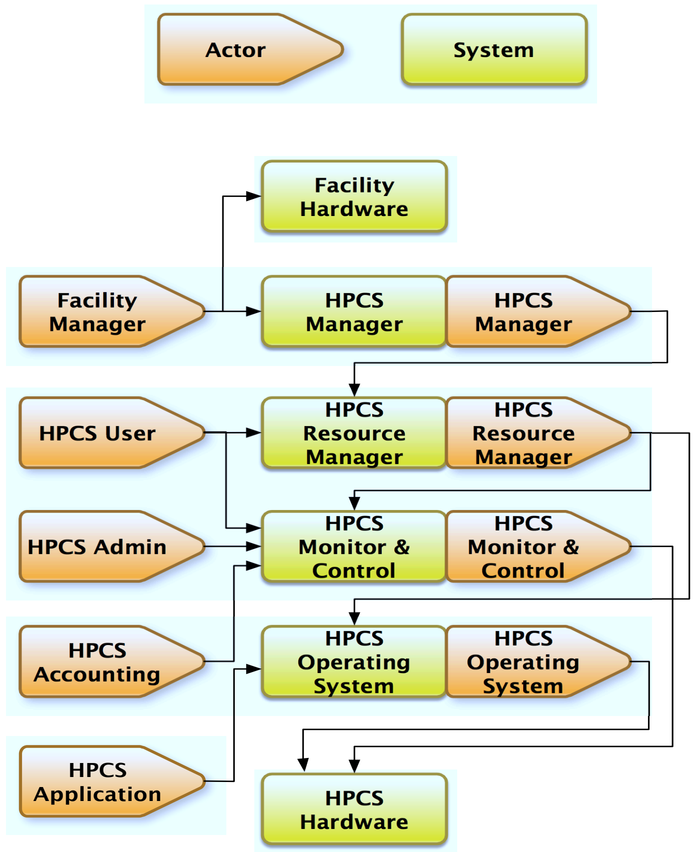

Figure 2.1: Top Level Conceptual Diagram representing the culmination of all Use Case Diagrams covered.

2.4 Security Model

The specification assumes traditional hardware (e.g. protection rings) and operating system support for access control. Implementations should only need traditional restrictions based on authenticated individual identity and/or the groups to which the individual belongs. A super user is likely needed as well. Depending on the implementation, the context structure (Section 4.3) may be sufficiently protected to allow for secure storage of access information. Future releases of the specification will address security and policy considerations in more detail.

Chapter 3

Theory of Operation

3.1 Overview

This section discusses many of the foundational concepts leveraged throughout the Power API specification. It should be noted that many terms commonly used when discussing object oriented languages are used in this section and the document as a whole. The use of these terms in no way implies that the Power API specification must be implemented using an object oriented language. We have attempted to achieve two goals, listed in order of priority: 1) programmer portability, where the programmer is the user of the API, and 2) the latitude of the implementor who will often become the user of the API benefitting from our first priority.

3.2 Power API Initialization

Using any of the Power API interfaces requires initialization. Initializaton returns a context. In the specification, the context is defined as an opaque pointer. This approach was taken to allow the maximum amount of flexibility to the implementor. The context returned will contain (act as the entry point to) the system description that is exposed to the user, all policy and privilege information, basically everything the user of the API requires to perform the functionality specified by the API. The system description is not required to be changed or updated during the life of a specific context. Initialization is accomplished by calling `PWR_CntxtInit()`. Resources created, like groups, p. 34
by the user during the life of the context should be cleaned up (destroyed)

by the user when no longer needed. The implementation is required to clean up all context resources when the user calls `PWR_CntxtDestroy()`. p. 35

3.3 Roles

The Power API specification leverages the concept of Roles. Roles represent the different types of users that exist which include:

- **Application** The application or application library executing on the compute resource. May also include run-time components running in user space.
- **Monitor and Control** Cluster management or Reliability Availability and Serviceability (RAS) systems, for example.
- **Operating System** Linux or specialized Light Weight Kernels which are found on HPC platforms and potentially portions of run-time systems.
- **User** The user of the HPC platform.
- **Resource Manager** This can include work load managers, schedulers, allocators and even portions of run-time systems.
- **Administrator** The system administrator or HPC platform manager.
- **HPCS Manager** The individual or individuals responsible for managing policy for the HPC platform, for example.
- **Accounting** Individual or software that produces reports of metrics for the HPC platform.

These brief definitions are not meant to be exhaustive. Roles are analogous with the *Actors* discussed in section 2.3. In some cases roles become the system that other roles interact with. For example, we specify an interface between the Application role (HPCS Application in figure 2.1) and the Operating System (HPCS Operating System in figure 2.1). The Operating System is the system (in UML terminology) that the Application role is interacting with. Notice in figure 2.1 that the specification also includes an interface between the Operating System role and the Hardware (HPCS Hardware in figure 2.1). These and other interfaces are described in chapter 7. The user of the API is required to specify what role they will assume when interacting with the system upon initialization of the API.

Roles are also provided as a mechanism for the implementation to express priority or precedence in circumstances where, for example, conflicting operations are requested.

3.4 System Description

The system description is the *view* of the system exposed to the user upon initialization via the context that is returned. Figure 3.1 depicts an example of a system description showing a hierarchical arrangement of objects. All object types listed in the specification must be defined by any implementation, but do not have to be used in the system description. The implementation chooses which objects will be employed in the system description and how they will be arranged. An object can only have a single parent but may have multiple children. Currently, a system description may only describe a single platform and have a single object of type `Platform` which represents the top of the hierarchy. Later revisions of the specification may include the ability to combine multiple platforms in the system description. This might be useful, for example, in representing an entire datacenter. While figure 3.1 depicts a homogeneous system description, homogeneity is *not* a requirement. In practice a system description can be heterogeneous and unbalanced.

To summarize the requirements:

- The `Platform` object type must be defined by the implementation and must appear at the top of the system description.
- All object types in this specification must be defined in any implementation. The use of the object types, with the exception of the `Platform` object type, is optional.
- Objects can only have one parent but may have many children. Currently the `Platform` object has no parent since it appears at the top of the system description. This will likely change in future versions of the specification.
- If an implementation chooses to add objects not defined in the specification they should only be exposed to the user in a vendor specific context to avoid unpredictable or non-portable behaviour (see `PWR_CntxtInit()`). p. 34

The following is a list of the object types currently included in the specification along with a short description of each.

- Platform - Currently, the one and only Platform object is the top level object of the system description exposed to the user of the API. The Platform object is intended to conceptually represent the entire Platform. For example, if the Platform object has a power or energy measurement or control capability exposed through the Platform objects attributes the scope of these attributes should be platform wide.

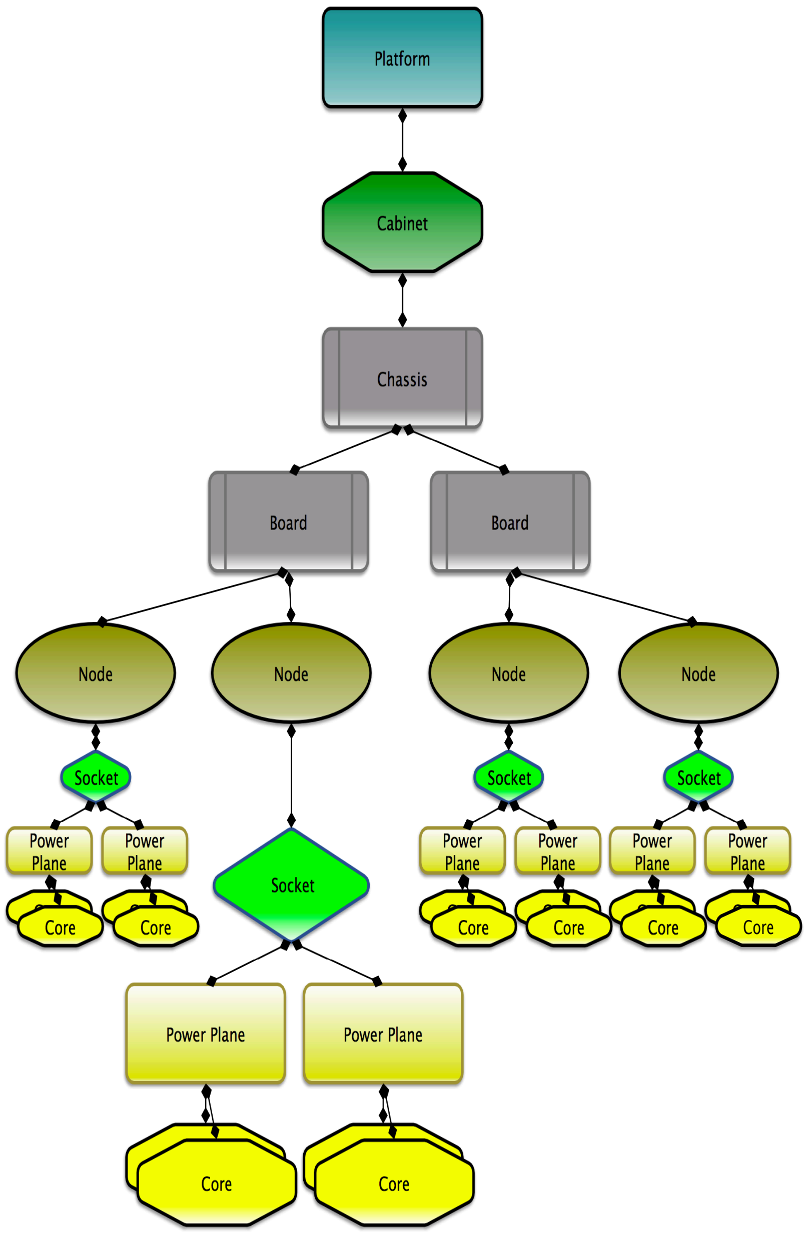

Figure 3.1: Hierarchical Depiction of System Objects

- Cabinet - Objects of type Cabinet are intended to represent the cabinets or racks that act as enclosures (or logical groupings) for the platform equipment. Beyond the utility of convenient groups of lower level objects (equipment) cabinets may have power or energy relevant capabilities which can be exposed through attributes associated with each Cabinet object.
- Chassis - Objects of type Chassis are intended to be used for finer grained organization of objects within the higher level Cabinet object. Chassis, like cabinets may have power or energy relevant capabilities that can be exposed to the user.
- Board - Board objects offer another method of organization for underlying objects (equipment). Boards may also have power and or energy relevant capabilities which can be exposed through associated attributes. For example, a board could contain the power supply and the point of instrumentation for collecting power or energy samples for a node or multiple nodes.
- Node - The Node type is probably one of the most universally important object types. Measuring and controlling the power and or energy characteristics of a node or multiple nodes (grouped into multiple Boards, Chassis or Cabinets) is important for a many reasons and provides a wide range of flexibility of configuration to the implementor. For example, on HPC platforms a single application typically executes on many nodes. Understanding the energy use of an application run can be obtained by collecting the energy use (via the appropriate Node attribute) for each node participating in that application execution. Node objects will likely have many attributes exposing many power and energy relevant capabilities.
- Socket - The Socket object is intended to represent the one or more processor sockets, or other component types that can be thought of as sockets, that make up a Node. For example, a single Node object may be a dual socket (dual CPU) node. The implementor may choose to enclose other component types (a NIC for example) within a Socket object, or add other object types as they see fit to represent the architecture they are describing. They can also decide to omit the use of this, or any other object type (currently other than Platform) in the system description.
- Power Plane - The Power Plane object is used to organize lower level objects (any types of objects) within a power domain or single point of

measurement and or control. For example, a pair of cores may share a power plane within a socket. This configuration is depicted in figure 3.1. This organization allows a pair of cores to be controlled from a single power control point in the hierarchy for convenience. This object type allows these power and energy relevant relationships to be expressed anywhere in the system description.

- Core - Core objects are intended to represent the individual processor cores within multi-core CPUs (or possibly GPUs). Modern architectures have an increasing number of cores per CPU (or GPU). In the near future it is likely that an abstraction between Socket and core would become useful as the number of cores increase. Physical and logical groupings of cores already exist in current architectures.
- Memory - The Memory object type is included to represent the growing range of memory types that exist on HPC platforms. Individual cores, for example, have Memory in the form of cache which the implementor may choose to organize differently from the main memory of the Node or a tertiary level of memory such as NVRAM.
- NIC - The NIC object is intended to represent the Network Interface Controller. As with many other object types, the organization of a NIC in relation to Boards, Nodes or even Cores is architecture dependent. The NIC object type is included in hopes that there are power and energy relevant capabilities included in future NICs.
- HT - The HT (Hardware Thread) object represents an OS-visible CPU. While from a physical perspective frequency and voltage changes occur at the physical core level, it is usually the case that these must be configured by software at the OS-visible CPU level. Typically the lowest-common denominator among all OS-visible CPUs is used to configure the physical core.

Additional object types may be defined by the implementor and placed anywhere in the hierarchy as long as the previously stated rules are not violated. Ultimately, the object types defined in this specification, and those added by the implementor, will be used to produce a system description describing the system presented to the user via the context returned upon initialization. Objects are used as interfaces to underlying functionality. The specification does not assume state is retained for objects. Additionally, the specification makes no guarantees with regards to race conditions between processes or threads.

3.5 Attributes

Attributes are an important part of the Power API. A large amount of basic functionality is exposed through the use of attributes. The term attribute is used somewhat conceptually since some attributes are implicit while others are explicitly defined as part of a required specification data structure (page 24). Attributes are used for a number of reasons such as to navigate through the system description, to access information or a measurement (sensor information for example) and for control (setting a P-state for example). Global attributes are attributes that are present for every object defined; whether required by the specification or added by the implementor.

The following is the list of global attributes:

- name - Unique identifying name of the object (see `PWR_ObjGetName()`). p. 37
- entry point - The position in the hierarchy after initialization (see `PWR_CntxtGetEntryPoint()`). p. 36
- type - The type of the object (see `PWR_ObjGetType()`). p. 36
- parent - The parent of an object is the object that is above it in the hierarchy (see `PWR_ObjGetParent()`). The only exception is the currently p. 38 single platform object whose parent is a pointer to NULL.
- children - Object or objects directly below an object in the hierarchy (see `PWR_ObjGetChildren()`). p. 39

Note, in the list above all the attributes are implicit. Explicit attributes are defined in the `PWR_AttrName` type definition. The majority of the at- p. 24 tributes defined in the specification, and likely those added by an implementator, are, and will be, explicit. The implicit attributes defined above are primarily used for navigation and are accessed through attribute specific functions which are described in Section 5.2.

Explicit attributes are either accessed through the generic attribute interface (Section 5.4) or attribute specific functions found in either the section describing the specific interface in which they are used or in Chapter 5, *Core (Common) Interface Functions*.

The attribute interface is intended to keep the specification from growing every time additional functionality is either specified or added by an implementor. As long as the new functionality fits within the defined attribute interfaces no additional API functions are required to be specified.

3.6 Metadata

Each object and object attribute pair can have additional descriptive metadata associated with it. This information is often useful for getting a better understanding of the meaning of objects and attributes and how to interpret the values read from attributes. Examples include a human readable name and description strings, the list of values supported by an attribute, and measurement accuracy and precision. The metadata interface (see section 5.5) returns information relevant to either a specific object or a specific attribute of a specific object. A given attribute name may have different metadata for different objects, even if the objects are of the same type (e.g., the voltage attribute of two node objects may have different metadata accuracy values).

3.7 Thread Safety

Implementations of the Power API are not required to provide thread safety to multiple threads of the same process. If necessary, users of the Power API must use locking or some other mechanism to ensure that only one thread per process calls into the Power API at a time. This requirement only applies to threads of the same process that may issue conflicting operations. Different processes may make simultaneous Power API calls without any coordination. If thread concurrency within a process is required, the `PWR_CntxtInit()` function can be called multiple times to initialize multi- p. 34
ple Power API contexts. Multiple threads of the same process may then simultaneously call into the Power API, so long as each thread operates on a different Power API context. For example, a process with four threads may create four Power API contexts and associate one context with each thread. The threads may then make Power API calls without any additional coordination, so long as each thread operates only on its assigned context and the objects exposed by its assigned context. Threads should not operate on objects exposed by another thread's context without employing locking or some other coordination mechanism.

Chapter 4

Type Definitions

4.1 Opaque Types

The following type definitions are specified to be opaque pointers from the point of view of Power API users. Power API implementations will typically map these pointers to internal implementation-specific state. The reason for using opaque pointers is to hide non-portable implementation details from users and give implementors of the API maximum flexibility.

```
typedef  void*  PWR_Cntxt
typedef  void*  PWR_Grp
typedef  void*  PWR_Obj
typedef  void*  PWR_Status
typedef  void*  PWR_Stat
```

4.2 Globally Relevant Definitions

The following definitions are specified on a global basis. The `PWR_MAJOR_VERSION`and `PWR_MINORVERSION` definitions are compile time constants that indicate the Power API version supported by the implementation. The `PWR_MAXSTRINGLEN` definition is a compile time constant that defines the maximum length of strings that can be returned from Power API calls, with the actual value being a vendor specific length.

`#define`	`PWR_MAJOR_VERSION`	`2`
`#define`	`PWR_MINOR_VERSION`	`0`
`#define`	`PWR_MAX_STRING_LEN`	*vendor-defined*

4.3 Context Relevant Type Definitions

The PWR_CntxtType and PWR_Role types are required to be defined by all implementations of the Power API. p. 22 When a new Power API context is created, one value from each of these types is used to determine the kind of context created (see section 5.1). For PWR_CntxtType, the only required value that an implementation must define is PWR_CNTXT_DEFAULT. This indicates that the new context will only contain Power API functionality that is explicitly defined in the specification, with no implementation-specific extentions present. Implementors may extend PWR_CntxtType with additional values, such as PWR_CNTXT_VENDOR, to provide contexts with additional functionality.

We anticipate that most implementations of the Power API will define additional PWR_CntxtType values that provide additional functionality, such as vendor, platform, or model specific extentions. If an implementation extends the specification, the extensions should only be visible to the user when they use a context that was created with an implementation-specific PWR_CntxtType value. If the implementation-specific extensions are not available to the user, initialization using an implementation-specific PWR_CntxtType value should result in failure. The user must always be able to initialize a context using PWR_CNTXT_DEFAULTto to get a context containing only the p. 22 standard specification features.

Differentiation between context types is the mechanism used by the Power API to enable extended vendor, platform or model specific capabilities while, at the same time, allowing portability for applications or tools that only leverage standard specification features. For example, a tool that leverages only the object and attribute types defined in the standard specification can initialize a Power API context using PWR_CNTXT_DEFAULTand not have to worry about dealing with any implementation-specific functionality. The context it receives will only provide functionality that is explicitly defined by the Power API specification.

PWR_Role is used to specify the role that the user is acting in when they p. 22 initialize a new context. Additional roles may not be added by the implementor. Notice that there is a role defined for every actor in Chapter 7 - Role/Systems Interfaces. We intend that the user's role will serve many purposes, such as determining the view of the system that is provided within the context when combined with the system the user is acting on. Roles can also be used to help determine the privilege of the user's context for purposes such as resolving the precedence of conflicting operations.

PWR_CntxtType

```
typedef   int                  PWR_CntxtType
#define   PWR_CNTXT_DEFAULT    0
#define   PWR_CNTXT_VENDOR     0
```

PWR_Role

```
typedef enum {
       PWR_ROLE_APP = 0, /* Application */
       PWR_ROLE_MC, /* Monitor and Control */
       PWR_ROLE_OS, /* Operating System */
       PWR_ROLE_USER, /* User */
       PWR_ROLE_RM, /* Resource Manager */
       PWR_ROLE_ADMIN, /* Administrator */
       PWR_ROLE_MGR, /* HPCS Manager */
       PWR_ROLE_ACC, /* Accounting */
       PWR_NUM_ROLES,
       /* */
       PWR_ROLE_INVALID = -1,
       PWR_ROLE_NOT_SPECIFIED = -2
} PWR_Role;
```

4.4 Object Relevant Type Definitions

The `PWR_ObjType` type is required to be defined by all implementations of the Power API specification. Objects with types defined by `PWR_ObjType` are used by the implementor to create the system description (see section 3.4) that is exposed to the user upon initialization. An implementation may extend this type by adding new object enumeration type, which must be added prior to `PWR_NUM_OBJ_TYPES`. The added implementation-specific object types will only be used by implementation-specific contexts (see section 4.3). Contexts that were initialized using the default context, `PWR_CNTXT_DEFAULT`, will only expose objects types defined in the list below. p. 23

PWR_ObjType

```
typedef enum {
        PWR_OBJ_PLATFORM = 0,
        PWR_OBJ_CABINET,
        PWR_OBJ_CHASSIS,
        PWR_OBJ_BOARD,
        PWR_OBJ_NODE,
        PWR_OBJ_SOCKET,
        PWR_OBJ_CORE,
        PWR_OBJ_POWER_PLANE,
        PWR_OBJ_MEM,
        PWR_OBJ_NIC,
        PWR_OBJ_HT,
        PWR_NUM_OBJ_TYPES,
        /* */
        PWR_OBJ_INVALID = -1,
        PWR_OBJ_NOT_SPECIFIED = -2
} PWR_ObjType;
```

4.5 Attribute Relevant Type Definitions

The `PWR_AttrName` and `PWR_AttrDataType` types are required to be implemented. Both may be extended by the implementor and exposed using an implementation specified context type (see section 4.3). If new `PWR_AttrName` entries are added it is required that the attribute name is specified and commented as shown in the `PWR_AttrName` structure. Likewise, new types must be added to the `PWR_AttrDataType` structure. It's important to note that the attribute interface currently supports only numeric types. Attributes should only be added to this definition if they can be meaningfully supported by the attribute interface (section 5.4). Additional attributes must be added prior to `PWR_NUM_ATTR_NAMES`. The Attributes in `PWR_AttrName` expose what we consider foundational measurement and control interfaces. Additional capabilities are and can be added using additional operations and often interface specific functions. p. 24

The `PWR_AttrAccessError` type is used to hold the error returns that are popped from the `PWR_Status` handle (see section 4.1) using the `PWR_StatusPopError()` function. The `PWR_AttrTracking` type is used to define the mode in which tracking the change for a given attribute is done, wither through the Power API only or polling to capture changes from all sources. p. 25 p. 34

PWR_AttrName

```
typedef enum {
	PWR_ATTR_PSTATE = 0, /* uint64_t */
	PWR_ATTR_CSTATE, /* uint64_t */
	PWR_ATTR_CSTATE_LIMIT, /* uint64_t */
	PWR_ATTR_SSTATE, /* uint64_t */
	PWR_ATTR_CURRENT, /* double, amps */
	PWR_ATTR_VOLTAGE, /* double, volts */
	PWR_ATTR_POWER, /* double, watts */
	PWR_ATTR_POWER_LIMIT_MIN, /* double, watts */
	PWR_ATTR_POWER_LIMIT_MAX, /* double, watts */
	PWR_ATTR_FREQ, /* double, Hz */
	PWR_ATTR_FREQ_LIMIT_MIN, /* double, Hz */
	PWR_ATTR_FREQ_LIMIT_MAX, /* double, Hz */
	PWR_ATTR_ENERGY, /* double, joules */
	PWR_ATTR_TEMP, /* double, degrees Celsius */
	PWR_ATTR_OS_ID, /* uint64_t */
	PWR_ATTR_THROTTLED_TIME, /* uint64_t */
	PWR_ATTR_THROTTLED_COUNT, /* uint64_t */
	PWR_ATTR_GOV, /* uint64_t */
	PWR_NUM_ATTR_NAMES,
	/* */
	PWR_ATTR_INVALID = -1,
	PWR_ATTR_NOT_SPECIFIED = -2
} PWR_AttrName;
```

PWR_AttrDataType

```
typedef enum {
	PWR_ATTR_DATA_DOUBLE = 0,
	PWR_ATTR_DATA_UINT64,
	PWR_NUM_ATTR_DATA_TYPES,
	/* */
	PWR_ATTR_DATA_INVALID = -1,
	PWR_ATTR_DATA_NOT_SPECIFIED = -2
} PWR_AttrDataType;
```

PWR_AttrAccessError

```
typedef struct {
       PWR_Obj obj; /* The object associated with the error */
       PWR_AttrName attr; /* The attribute associated with the error
           */
       int index; /* The index in the output array where the error
           occurred */
       int error; /* The error code, see Error Return Definitions
           section */
} PWR_AttrAccessError;
```

PWR_AttrGov

```
typedef enum {
       PWR_GOV_LINUX_ONDEMAND,
       PWR_GOV_LINUX_PERFORMANCE,
       PWR_GOV_LINUX_CONSERVATIVE,
       PWR_GOV_LINUX_POWERSAVE,
       PWR_GOV_LINUX_USERSPACE
} PWR_AttrGov;
```

PWR_AttrTracking

```
typedef enum {
       PWR_TRACKING_APIONLY,
       PWR_TRACKING_POLL
} PWR_AttrTracking;
```

4.6 Metadata Relevant Type Definitions

The `PWR_MetaName` type is required to be implemented. The type may be extended by the implementor and the additional capabilities may be exposed using an implementation specified context type (see section 4.3). If new `PWR_MetaName` items are added, it is required that the metadata name be specified and commented as shown in the `PWR_MetaName` definition. Additional metadata items must be added prior to `PWR_NUM_META_NAMES`.

PWR_MetaName

```
typedef enum {
       PWR_MD_NUM = 0, /* uint64_t */
       PWR_MD_MIN, /* either uint64_t or double, depending on
           attribute type */
       PWR_MD_MAX, /* either uint64_t or double, depending on
           attribute type */
       PWR_MD_PRECISION, /* uint64_t */
       PWR_MD_ACCURACY, /* double */
       PWR_MD_UPDATE_RATE, /* double */
       PWR_MD_SAMPLE_RATE, /* double */
       PWR_MD_TIME_WINDOW, /* PWR_Time */
       PWR_MD_TS_LATENCY, /* PWR_Time */
       PWR_MD_TS_ACCURACY, /* PWR_Time */
       PWR_MD_MAX_LEN, /* uint64_t, max strlen of any returned
           metadata string. */
       PWR_MD_NAME_LEN, /* uint64_t, max strlen of PWR_MD_NAME */
       PWR_MD_NAME, /* char *, C-style NULL-terminated ASCII string */
       PWR_MD_DESC_LEN, /* uint64_t, max strlen of PWR_MD_DESC */
       PWR_MD_DESC, /* char *, C-style NULL-terminated ASCII string */
       PWR_MD_VALUE_LEN, /* uint64_t, max strlen returned by
           PWR_MetaValueAtIndex */
       PWR_MD_VENDOR_INFO_LEN, /* uint64_t, max strlen of
           PWR_MD_VENDOR_INFO */
       PWR_MD_VENDOR_INFO, /* char *, C-style NULL-terminated ASCII
           string */
       PWR_MD_MEASURE_METHOD, /* uint64_t, 0/1 depending on real/model
            mesurement */
       PWR_NUM_META_NAMES,
       /* */
       PWR_MD_INVALID = -1,
       PWR_MD_NOT_SPECIFIED = -2
} PWR_MetaName;
```

4.7 Error Return Definitions

The following required definitions are the available status returns for the functions described in this specification. It is anticipated that this list will grow. The implementor is also free to add status returns to express conditions not currently covered in the specification and expose them using an implementation specified context type (see section 4.3). The range -127 through 128 are reserved for use by the Power API specification. Positive numbers greater than zero are to be used for warnings.

```
#define PWR_RET_WARN_TRUNC 5
#define PWR_RET_WARN_NO_GRP_BY_NAME 4
#define PWR_RET_WARN_NO_OBJ_BY_NAME 3
#define PWR_RET_WARN_NO_CHILDREN 2
#define PWR_RET_WARN_NO_PARENT 1
#define PWR_RET_SUCCESS 0
#define PWR_RET_FAILURE -1
#define PWR_RET_NOT_IMPLEMENTED -2
#define PWR_RET_EMPTY -3
#define PWR_RET_INVALID -4
#define PWR_RET_LENGTH -5
#define PWR_RET_NO_ATTRIB -6
#define PWR_RET_NO_META -7
#define PWR_RET_READ_ONLY -8
#define PWR_RET_BAD_VALUE -9
#define PWR_RET_BAD_INDEX -10
#define PWR_RET_OP_NOT_ATTEMPTED -11
#define PWR_RET_NO_PERM -12
#define PWR_RET_OUT_OF_RANGE -13
#define PWR_RET_NO_OBJ_AT_INDEX -14
```

4.8 Time Related Definitions

`PWR_Time` is defined as a 64-bit value used to hold timestamps in nanoseconds for a wide range of functionality. For those timestamps that are to be used in relation to an epoch, midnight January 1st, 1970 will be considered the beginning of the epoch. This will provide for hundreds of years to be expressed from the epoch point, which is sufficient for the purposes of the Power API. `PWR_Time` is also used for other structures designed to record time values (`PWR_TimePeriod`, page 28 for example). `PWR_TIME_UNINIT` is used as an indicator that the time value has not been initialized. This is intended to allow the implementation to make decisions on how a function is being used based on whether a time value has been specified or not (for example, the Statistics functions in section 5.6). `PWR_TIME_UNKNOWN` is an output, which indicates that the time of an event was not recorded. For example, a maximum value for an attribute could be known for a given time period, but the instant at which the maximum occurred is unknown. The `PWR_TimePeriod` type allows for three timestamps, start, stop and instant.

Instant is available to indicate when a statistically significant event occurred within the window delineated by start and stop. For example, if the user requests the PWR_ATTR_STAT_MAX statistic for PWR_ATTR_POWER, the start and stop times will indicate the window of time over which the maximum value was calculated. The instant would indicate the instant in time the maximum value occurred. Defining PWR_Time, PWR_TIME_UNINIT, PWR_TIME_UNKNOWN, and PWR_TimePeriod as specified is required.

```
typedef uint64_t PWR_Time;
#define PWR_TIME_UNINIT 0
#define PWR_TIME_UNKNOWN 0
```

PWR_TimePeriod

```
typedef struct {
       PWR_Time start;
       PWR_Time stop;
       PWR_Time instant;
} PWR_TimePeriod;
```

4.9 Statistics Relevant Type Definitions

The PWR_AttrStat type includes the list of currently defined statistics potentially available to the user of an implementation. Potentially, because this feature requires either direct device or software support. Statistics are generated on a per-attribute basis (see PWR_AttrName on page 24). The statistics type definitions are required to be implemented and are used with the statistics functions (see section 5.6).

PWR_AttrStat

```
typedef enum {
        PWR_ATTR_STAT_MIN = 0,
        PWR_ATTR_STAT_MAX,
        PWR_ATTR_STAT_AVG,
        PWR_ATTR_STAT_STDEV,
        PWR_ATTR_STAT_CV,
        PWR_ATTR_STAT_SUM,
        PWR_NUM_ATTR_STATS,
        /* */
        PWR_ATTR_STAT_INVALID = -1,
        PWR_ATTR_STAT_NOT_SPECIFIED = -2
} PWR_AttrStat;
```

PWR_ID

```
typedef enum {
        PWR_ID_USER = 0,
        PWR_ID_JOB,
        PWR_ID_RUN,
        PWR_NUM_IDS,
        /* */
        PWR_ID_INVALID = -1,
        PWR_ID_NOT_SPECIFIED = -2
} PWR_ID;
```

4.10 OS Hardware Interface Type Definitions

The following definitions are used in the Operating system to Hardware interface described in section 7.1. Each definition will be described below along with its specification. All of the definitions in this section are required, even if the corresponding OS/HW functions are not implemented.

PWR_OperState

The `PWR_OperState` type is used to describe the state being requested by OS to Hardware interface functions that require power/performance state information such as P-State and C-State information. Both `c_state_num` and `p_state_num` must be provided.

```
typedef struct {
        uint64_t c_state_num;
        uint64_t p_state_num;
} PWR_OperState;
```

4.11 Application OS Interface Type Definitions

The following definitions are primarily used in the Application to Operating system interface described in section 7.3. Each definition will be described below along with its specification. All of the definitions in this section are required, even if the corresponding App/OS functions are not implemented.

PWR_RegionHint

The `PWR_RegionHint` type is an abstraction intended to allow the application to communicate power and performance significant information to the operating system. It is used in conjunction with `PWR_RegionIntensity` to describe the type and extent of the behavior described for a given execution region. This information can then be used to *tune* components, with the intent being a more power/performance efficient use of the components results. For example, if an application is going into a serial region, the performance of the application may benefit from the core running the serial portion of the code at a higher frequency, thereby completing that serial portion faster. Since the application is in a serial portion, the implementation may determine that the remaining cores may be put into a more power efficient state (a sleep state for example), thus possibly resulting in both a performance increase and a decrease in the amount of power/energy the application uses. Regions may be specified as `PWR_REGION_DEFAULT` to indicate that the application is no longer providing a hint as to the region characteristics of currently executing code.

```
typedef enum {
	PWR_REGION_DEFAULT = 0,
	PWR_REGION_SERIAL,
	PWR_REGION_PARALLEL,
	PWR_REGION_COMPUTE,
	PWR_REGION_COMMUNICATE,
	PWR_REGION_IO,
	PWR_REGION_MEM_BOUND,
	PWR_REGION_GLOBAL_LOOP,
	PWR_NUM_REGION_HINTS,
	/* */
	PWR_REGION_INVALID = -1,
	PWR_REGION_NOT_SPECIFIED = -2
} PWR_RegionHint;
```

PWR_RegionIntensity

The `PWR_RegionIntensity` type is an abstraction of a given level of intensity for a `PWR_RegionHint`. It provides five levels of intensity as well as `PWR_Region_INT_NONE`, which can be used in the case where the intensity is not known, is not applicable, or in cases where the operating system or runtime may be better equipped to determine the intensity of a given code region.

```
typedef enum {
	PWR_REGION_INT_HIGHEST = 0,
	PWR_REGION_INT_HIGH,
	PWR_REGION_INT_MEDIUM,
	PWR_REGION_INT_LOW,
	PWR_REGION_INT_LOWEST,
	PWR_REGION_INT_NONE,
	PWR_NUM_REGION_INTENSITIES,
	/* */
	PWR_REGION_INT_INVALID = -1,
	PWR_REGION_INT_NOT_SPECIFIED = -2
} PWR_RegionIntensity;
```

PWR_SleepState

The `PWR_SleepState` type is a high level abstraction of the different sleep state levels that may be provided on a given system. The sleep levels are translated into the appropriate hardware level constructs by lower layers of the PowerAPI.

```
typedef enum {
	PWR_SLEEP_NO = 0,
	PWR_SLEEP_SHALLOW,
	PWR_SLEEP_MEDIUM,
	PWR_SLEEP_DEEP,
	PWR_SLEEP_DEEPEST,
	PWR_NUM_SLEEP_STATES,
	/* */
	PWR_SLEEP_INVALID = -1,
	PWR_SLEEP_NOT_SPECIFIED = -2
} PWR_SleepState;
```

PWR_PerfState

The PWR_PerfState type is an abstraction meant to describe the different possible performance states in which hardware may be placed.

```
typedef enum {
	PWR_PERF_FASTEST = 0,
	PWR_PERF_FAST,
	PWR_PERF_MEDIUM,
	PWR_PERF_SLOW,
	PWR_PERF_SLOWEST,
	PWR_NUM_PERF_STATES,
	/* */
	PWR_PERF_INVALID = -1,
	PWR_PERF_NOT_SPECIFIED = -2
} PWR_PerfState;
```

Chapter 5

Core (Common) Interface Functions

Core, or so called Common, interface functions are functions that can be used, at least in par, by most of the interfaces described in the Power API specification. Core functions include the following areas:

- **Initialization**, required to use any of the functionality described in this specification,
- **Navigation** functions allow the user to traverse the system description and discover information about the underlying platform,
- **Group** functions, primarily a convenience abstraction,
- **Attribute** functions expose measurement and control functionality,
- **Metadata** functions allow the user to access additional information about objects and attributes (often device or instrumentation specific information),
- **Statistics** functions are used to generate statistical information based on fundamental attribute information (measurements),

and other functionality that is common across a number of interfaces.

5.1 Initialization

Initialization using `PWR_CntxtInit` is required to use any of the functionality documented in this specification. The user supplies the type of the context requested and their role. Currently, the specification's only required context type is `PWR_CNTXT_DEFAULT`. The context type is intended to be one way in which the implementor can distinguish their implementation from the standard specification and other implementations (see section 4.3). The user must also supply their role (see page 22 for the PWR_Role definition). One purpose of specifying the role is to convey what type of user they intend to be, and therefore, how they would like to interact with or how the underlying implementation manages the privileges granted to the user/role combination. A system administrator (PWR_ROLE_ADMIN) will desire and require different capabilities, privileges and level of abstraction than the application user (PWR_ROLE_APP), for example.

The user also has the opportunity to specify a name that will be associated with the context. This *feature* is anticipated to be useful in supporting advanced functionality. Initialization returns a context to the user. The context contains the user's view of the system, dependent on what type of context was requested, the user's role and implementation specifics. The system description that the user is exposed to must conform to the rules outlined in the specification (see sections 3.2 and 3.4). The context should be destroyed (cleaned up) by using the PWR_CntxtDestroy function when no longer needed.

Function Prototype for PWR_CntxtInit()

The PWR_CntxtInit function is required to be called before using any other Power API function. The context returned is passed to other Power API functions either explicitly as an argument or implicitly through an argument associated with the context.

```
int PWR_CntxtInit(PWR_CntxtType type, PWR_Role role, const char*
name, PWR_Cntxt* context)
```

Arguments		Description
IN	PWR_CntxtType type	The requested context type.
IN	PWR_Role role	The role of the user.
IN	const char* name	User specified string name to be associated with the context.
OUT	PWR_Cntxt* context	The user's context.

See page 22 for a discussion of contexts and roles.

Return Code(s)	Description
PWR_RET_SUCCESS	Upon SUCCESS, context is set to a valid user context.
PWR_RET_FAILURE	Upon FAILURE.

Function Prototype for PWR_CntxtDestroy()

The PWR_CntxtDestroy function is used to destroy (clean up) the context obtained with PWR_CntxtInit. The implementation is required to clean up, unlink, destroy (as appropriate) all context resources as a result of this call.

```
int PWR_CntxtDestroy(PWR_Cntxt context)
```

Arguments	Description
IN PWR_Cntxt context	The context obtained using PWR_CntxtInit the user wishes to destroy.

Return Code(s)	Description
PWR_RET_SUCCESS	Upon SUCCESS.
PWR_RET_FAILURE	Upon FAILURE.

5.2 Hierarchy Navigation Functions

Hierarchy navigation (also called discovery) is accomplished using attributes (EntryPoint, Type, Parent and Children) that are implicit to every object in the system description whether defined in the specification or added by the implementor. Navigation is accomplished using these attributes, through the associated function calls, within the context made available to the user upon initialization. After initialization the first call will generally be PWR_CntxtGetEntryPoint to determine the user's entry point in the system hierarchy provided within the user's context. Depending on the user, the interface and the role, the context could contain a view of the entire system description or a subset of the system description. Navigating through the hierarchy is accomplished with PWR_ObjGetParent to navigate up and PWR_ObjGetChildren to navigate down. To understand what kind of object was returned with either of these calls the user can utilize PWR_ObjGetType call.

The name of the object can be discovered using the `PWR_ObjGetName` function and if the user has a name, the associated object can be discovered using the `PWR_CntxtGetObjByName` function.

The Power API does not provide an explicit "Free Object" interface. Specifically, objects returned by Power API interfaces do not need to be later freed or released explicitly. This design choice was made in order to keep usage of the Power API as simple as possible, with the potential cost of an increased burden on the Power API implementor to limit implementation-internal memory usage.

Function Prototype for `PWR_CntxtGetEntryPoint()`

The `PWR_CntxtGetEntryPoint` call is typically used immediately following initialization. Whenever `PWR_CntxtGetEntryPoint` is called the implementation defined entry point (location) in the system description is returned. `PWR_CntxtGetEntryPoint` can always be called to reposition or reorient the user to the initial entry location.

```
int PWR_CntxtGetEntryPoint(PWR_Cntxt context, PWR_Obj* entry_
point)
```

Arguments		Description
IN	`PWR_Cntxt context`	The user's context.
OUT	`PWR_Obj* entry_point`	The user's entry point into the system description (the same for the life of the context).

Return Code(s)	Description
`PWR_RET_SUCCESS`	Upon SUCCESS, entry_point set to system description entry point (object).
`PWR_RET_FAILURE`	Upon FAILURE.

Function Prototype for `PWR_ObjGetType()`

The `PWR_ObjGetType` function returns the type of the object specified. See page 23 for valid object types.

```
int PWR_ObjGetType(PWR_Obj object, PWR_ObjType* type)
```

Arguments	Description
IN PWR_Obj object	The object that the user wishes to determine the type of.
OUT PWR_ObjType* type	The type of the specified object.

Return Code(s)	Description
PWR_RET_SUCCESS	Upon SUCCESS, type is set to the type of the specified object.
PWR_RET_FAILURE	Upon FAILURE, type is set to PWR_OBJ_INVALID.

Function Prototype for PWR_ObjGetName()

The PWR_ObjGetName function copies the name of the specified object into the user provided buffer. See page 40 to get the object based on the unique name using PWR_CntxtGetObjByName.

```
int PWR_ObjGetName(PWR_Obj object, char* dest, size_t len)
```

Arguments	Description
IN PWR_Obj object	The object that the user wishes to determine the name of.
IN char* dest	The address of the user provided buffer.
IN size_t len	The length of the user provided buffer.

Return Code(s)	Description
PWR_RET_SUCCESS	Upon SUCCESS, the buffer will contain the name of the object, the string will include a terminating null byte.
PWR_RET_WARN_TRUNC	Call succeeded, but the length of object name was longer than the provided buffer and the name was truncated.
PWR_RET_FAILURE	Upon FAILURE.

Function Prototype for PWR_ObjGetSizeOfName()

The PWR_ObjGetSizeOfName returns the length of an object's name. The len parameter will contain the length of the name of the specified object including any string terminators upon return. See page 40 to get the object based on the unique name using PWR_CntxtGetObjByName.

```
int PWR_ObjGetSizeOfName(PWR_Obj object, size_t* len)
```

Arguments	Description
IN PWR_Obj object	The object that the user wishes to determine the name of.
IN/OUT size_t* len	The length of the user provided buffer.

Return Code(s)	Description
PWR_RET_SUCCESS	Upon SUCCESS, the len parameter will contain the size of buffer required to successfully call PWR_ObjGetName, including terminating null byte.
PWR_RET_FAILURE	Upon FAILURE.

Function Prototype for PWR_ObjGetParent()

The PWR_ObjGetParent function is used to find the object immediately above the specified object in the system description available to the user through the current context. Note, currently, there are some cases where an object has no parent, namely the platform object.

```
int PWR_ObjGetParent(PWR_Obj object, PWR_Obj* parent)
```

Arguments	Description
IN PWR_Obj object	The object that the user wishes to determine the parent of.
OUT PWR_Obj* parent	The parent object of the specified input object.

Return Code(s)	Description
PWR_RET_SUCCESS	Upon SUCCESS, parent set to parent of specified object.
PWR_RET_WARN_NO_PARENT	Call succeeded but specified object does not have a parent.
PWR_RET_FAILURE	Upon FAILURE.

Function Prototype for `PWR_ObjGetChildren()`

The `PWR_ObjGetChildren` function returns the child or children of the specified object. The caller is expected to check the return code of `PWR_ObjGetChildren` to determine if the object has children or not. If the specified object has one or more children, indicated by a return code of `PWR_RET_SUCCESS`, a new group (`PWR_Grp`) is returned that contains the object's children. The user is responsible for destroying this group when it is no longer needed (see `PWR_GrpDestroy` on page 42). If the specified object has no children, indicated by a return code of `PWR_RET_WARN_NO_CHILDREN`, no group is returned and the input (`PWR_Grp`) is not modified.

```
int PWR_ObjGetChildren(PWR_Obj objec, PWR_Grp* group)
```

Arguments		Description
IN	`PWR_Obj objec`	The object that the user wishes to determine the children of.
OUT	`PWR_Grp* group`	On input, this should be set to point to an uninitialized `PWR_Grp` (i.e., the caller should not call `PWR_GrpCreate` ahead of time). If `PWR_RET_SUCCESS` is returned, *group will be set to a newly created group containing the `object`'s children. If `PWR_RET_WARN_NO_CHILDREN` is returned, the input `PWR_Grp` is not modified.

Return Code(s)	Description
PWR_RET_SUCCESS	Upon SUCCESS, group is set to a newly created group containing the child or children of specified object.
PWR_RET_WARN_NO_CHILDREN	Call succeeded but specified object does not have any children. The input PWR_Grp is not modified.
PWR_RET_FAILURE	Upon FAILURE.

Function Prototype for PWR_CntxtGetObjByName()

The PWR_CntxtGetObjByName function returns the object given the context and unique object name. See page 37 to get the name of a specified object using PWR_ObjGetName.

```
int PWR_CntxtGetObjByName(PWR_Cntxt context, const char * name,
PWR_Obj* object)
```

Arguments		Description
IN	PWR_Cntxt context	The context containing the object that the user wishes to retrieve given its unique name. Note, the object may be present in the system but not available to the user through the current context.
IN	const char * name	The unique name of the object that the user wishes to retrieve.
OUT	PWR_Obj* object	The object that corresponds to the name specified by the user.

Return Code(s)	Description
PWR_RET_SUCCESS	Upon SUCCESS, object is set to object corresponding to name specified by user.
PWR_RET_WARN_NO_OBJ_BY_NAME	If no object exists corresponding to name provided.
PWR_RET_FAILURE	Upon FAILURE.

5.3 Group Functions

Group functions are provided as a convenience in situations, for example, where an operation, or operations are required to be executed on multiple objects. Rather than executing the same operation multiple times, once for each object, some operations provide a group variant to streamline this type of functionality. Groups can be dynamically created (`PWR_GrpCreate`) when needed and can exist for short periods of time and destroyed with `PWR_GrpDestroy`, or exist for the duration of the users context. Groups may not contain multiple instances of the same object, i.e. duplicate objects are not allowed. When a new group is the product of a function (`PWR_GrpUnion, PWR_GrpIntersection, PWR_GrpDifference`) and the result of the function operation is the empty set (no objects) an empty group (group with no objects) should be the result and the function should return `PWR_RET_SUCCESS`. It is the responsibility of the user to clean up all groups produced as a result of group functions using `PWR_GrpDestroy`. Groups can only contain objects from a single `PWR_cntxt`. Group operations that involve multiple groups must be performed with groups from the same context.

Function Prototype for `PWR_GrpCreate()`

The `PWR_GrpCreate` function is used to create a new group which will be associated with and unique to the users context.

```
int PWR_GrpCreate(PWR_Cntxt context, PWR_Grp* group)
```

Arguments		Description
IN	`PWR_Cntxt context`	The user's context that the group, when created, will be associated with.
OUT	`PWR_Grp* group`	The new (empty) group.

Return Code(s)	Description
`PWR_RET_SUCCESS`	Upon SUCCESS, group is set to new (empty) group.
`PWR_RET_FAILURE`	Upon FAILURE.

Function Prototype for `PWR_GrpDestroy()`

The `PWR_GrpDestroy` function is used to destroy (clean up) a group created by a user.

```
int PWR_GrpDestroy(PWR_Grp group)
```

Arguments	Description
IN PWR_Grp group	The group that the user is acting on.

Return Code(s)	Description
PWR_RET_SUCCESS	Upon SUCCESS.
PWR_RET_FAILURE	Upon FAILURE.

Function Prototype for `PWR_GrpAddObj()`

The `PWR_GrpAddObj` function is used to add a specified object to a specified group. Duplicate objects are not allowed in groups. Adding an object that would be a duplicate of one already in the group will result in no insertion and returns `PWR_RET_SUCCESS`.

```
int PWR_GrpAddObj(PWR_Grp group, PWR_Obj object)
```

Arguments	Description
IN/OUT PWR_Grp group	The group that the user is acting on.
IN PWR_Obj object	The object to be added to the specified group.

Return Code(s)	Description
PWR_RET_SUCCESS	Upon SUCCESS.
PWR_RET_FAILURE	Upon FAILURE.

Function Prototype for `PWR_GrpRemoveObj()`

The `PWR_GrpRemoveObj` function is used to remove a specified object from a specified group. Attempting to remove an object that is not a member of a group will result in `PWR_RET_SUCCESS`.

```
int PWR_GrpRemoveObj(PWR_Grp group, PWR_Obj object)
```

Arguments	Description
IN/OUT `PWR_Grp group`	The group that the user is acting on.
IN `PWR_Obj object`	The object to be removed from the specified group.

Return Code(s)	Description
`PWR_RET_SUCCESS`	Upon SUCCESS.
`PWR_RET_FAILURE`	Upon FAILURE.

Function Prototype for `PWR_GrpGetNumObjs()`

The `PWR_GrpGetNumObjs` function is used to get the number of objects contained in the specified group.

```
int PWR_GrpGetNumObjs(PWR_Grp group)
```

Arguments	Description
IN `PWR_Grp group`	The group that the user is acting on.

Return Code(s)	Description
`int`	Upon SUCCESS, the number of objects contained in the specified group.
`PWR_RET_FAILURE`	Upon FAILURE.

Function Prototype for `PWR_GrpGetObjByIndx()`

The `PWR_GrpGetObjByIndx` is used to get the object from the specified group at the specified index.

```
int PWR_GrpGetObjByIndx(PWR_Grp group, int index, PWR_Obj*
object)
```

Arguments		Description
IN	PWR_Grp group	The group that the user is acting on.
IN	int index	The index within the specified group of the desired object.
OUT	PWR_Obj* object	The object at the specified index in the specified group.

Return Code(s)	Description
PWR_RET_SUCCESS	Upon SUCCESS, object is set to object at specified index.
PWR_RET_NO_OBJ_AT_INDEX	No object at specified index in specified group.
PWR_RET_FAILURE	Upon FAILURE.

Function Prototype for PWR_GrpDuplicate()

The PWR_GrpDuplicate function is used to duplicate an existing group. The duplicate group is a new separate group from the original group specified. Actions on the duplicate group do not affect the original group and vice versa.

```
int PWR_GrpDuplicate(PWR_Grp group1, PWR_Grp* group2)
```

Arguments		Description
IN	PWR_Grp group1	The original group (group1).
OUT	PWR_Grp* group2	Duplicate (group2) of the original group (group1) specified by user even if the original group contains no objects.

Return Code(s)	Description
PWR_RET_SUCCESS	Upon SUCCESS, duplicate group of original group created.
PWR_RET_FAILURE	Upon FAILURE.

Function Prototype for `PWR_GrpUnion()`

The `PWR_GrpUnion` function is used to create a group that is the union ($\cup$) of two specified groups. The union group created is a new separate group from the original groups specified. Actions on the union group do not affect the original groups and vice versa.

```
int PWR_GrpUnion(PWR_Grp group1, PWR_Grp group2, PWR_Grp* group3)
```

Arguments		Description
IN	`PWR_Grp group1`	The first of the two groups used in the union, ($\cup$) operation.
IN	`PWR_Grp group2`	The second of the two groups used in the union, ($\cup$) operation.
OUT	`PWR_Grp* group3`	he output group (group3) is the union, ($\cup$) operation, of the first (group1) and second (group2) groups specified. If the result of the union operation is the empty set group3 is an empty group (valid group with no objects).

Return Code(s)	Description
`PWR_RET_SUCCESS`	Upon SUCCESS, group3 contains the union of group1 and group2.
`PWR_RET_FAILURE`	Upon FAILURE.

Function Prototype for `PWR_GrpIntersection()`

The `PWR_GrpIntersection` function is used to create a group that is the Intersection ($\cap$) of two specified groups. The intersection group is a new separate group from the original groups specified. Actions on the intersection group do not affect the original groups and vice versa.

```
int PWR_GrpIntersection(PWR_Grp group1, PWR_Grp group2, PWR_Grp*
group3)
```

Arguments		Description
IN	`PWR_Grp group1`	The first of the two groups used in the Intersection (∩) operation.
IN	`PWR_Grp group2`	The second of the two groups used in the intersection (∩) operation.
OUT	`PWR_Grp* group3`	The output group (group3) is the intersection, (∩) operation, of the first (group1) and second (group2) groups specified. If the result of the intersection operation is the empty set group3 is an empty group (valid group with no objects).

Return Code(s)	Description
`PWR_RET_SUCCESS`	Upon SUCCESS, group3 contains the intersection of group1 and group2.
`PWR_RET_FAILURE`	Upon FAILURE.

Function Prototype for `PWR_GrpDifference()`

The `PWR_GrpDifference` function is used to create a group that is the Difference (\) of two specified groups. The difference group is a new separate group from the original groups specified. Actions on the difference group do not affect the original groups and vice versa. In the event that the output `PWR_Grp` contains no objects see 5.3 for the definition of the output, `PWR_Grp`.

```
int PWR_GrpDifference(PWR_Grp group1, PWR_Grp group2, PWR_Grp*
group3)
```

Arguments		Description
IN	`PWR_Grp group1`	The first of the two groups used in the difference (\) operation.
IN	`PWR_Grp group2`	The second of the two groups used in the difference (\) operation.
OUT	`PWR_Grp* group3`	The output group (group3) is the difference, (\) operation, of the first (group1) and second (group2) groups specified. If the result of the difference operation is the empty set group3 is an empty group (valid group with no objects).

Return Code(s)	Description
`PWR_RET_SUCCESS`	Upon SUCCESS, group3 contains the difference of group1 and group2.
`PWR_RET_FAILURE`	Upon FAILURE.

Function Prototype for PWR_GrpSymDifference()

Function Prototype for `PWR_GrpSymDifference()`

The `PWR_GrpSymDifference` function is used to create a group that is the Symmetric Difference ($\triangle$) of two specified groups. The symmetric difference group is a new separate group from the original groups specified. Actions on the symmetric difference group do not affect the original groups and vice versa. In the event that the output `PWR_Grp` contains no objects see 5.3 for the definition of the output, `PWR_Grp`.

```
int PWR_GrpSymDifference(PWR_Grp group1, PWR_Grp group2, PWR_Grp*
group3)
```

Arguments	**Description**
Input `PWR_Grp group1`	The first of the two groups used in the symmetric difference ($\triangle$) operation.
Input `PWR_Grp group2`	The second of the two groups used in the symmetric difference ($\triangle$) operation.
Output`PWR_Grp* group3`	The output group (group3) is the symmetric difference, ($\triangle$) operation, of the first (group1) and second (group2) groups specified. If the result of the symmetric difference operation is the empty set group3 is an empty group (valid group with no objects).

Return Code(s)	**Description**
`PWR_RET_SUCCESS`	Upon SUCCESS, group3 contains the symmetric difference of group1 and group2.
`PWR_RET_FAILURE`	Upon FAILURE.

Function Prototype for `PWR_CntxtGetGrpByName()`

The `PWR_CntxtGetGrpByName` function returns a group in a given context via a unique group name. This function is included to allow the user to make use of groups that are provided with the initial context by the implementation. The list of valid group names should be provided by the vendor in their documentation. Due to the defined group names being vendor specific, use of this function should be considered non-portable. The group returned by this call must be functionally identical to a group created via `PWR_GrpCreate()`. Like a group created with `PWR_GrpCreate()` groups returned by `PWR_CntxtGetGrpByName()` must be destroyed with the `PWR_GrpDestroy()` call.

```
int PWR_CntxtGetGrpByName(PWR_Cntxt context, const char* name,
PWR_grp* group)
```

Arguments		Description
IN	PWR_Cntxt context	The context containing the group that the user wishes to retrieve given its unique name.
IN	const char* name	The unique name of the group that the user wishes to retrieve.
OUT	PWR_grp* group	The implementation provided group corresponding to the specified name.

Return Code(s)	Description
PWR_RET_SUCCESS	Upon SUCCESS, group corresponding to the specified name.
PWR_RET_WARN_NO_GRP_BY_NAME	If no implementation supplied group exists corresponding to name provided.
PWR_RET_FAILURE	Upon FAILURE.

5.4 Attribute Functions

The Attribute functions make up the foundation of the Power API specification, providing measurement (get) and control (set) interfaces for a wide range of power and energy related functionality. Get and set interfaces are provided for single attribute/single object, multiple attribute/single object, single attribute/multiple objects (group) and multiple attributes/multiple objects (group). In each case the user specifies the attribute or attributes to get or set. The valid attribute names are defined in the PWR_AttrName structure (see page 24). A complete list of all the valid attributes and their meanings can be found in table 5.1, section 5.8. The timestamp is a critical part of the get (measurement) interface for power and energy related information. It is very important that the timestamp returned (PWR_Time) be an accurate representation of when the value returned was measured to the best possible temporal accuracy, not when the function was called. It is required by the specification that the value returned is the value that was measured as close as possible to when the get function was called. The quality of the measurement and timestamp are device and implementation dependent. Information about each attribute can be obtained through the metadata interface, described in section 5.5.

Function Prototype for `PWR_ObjAttrGetValue()`

The `PWR_ObjAttrGetValue` function is provided to get the value of a single specified attribute (`PWR_AttrName attr`) from a single specified object (`PWR_Obj object`). The timestamp returned (`PWR_Time *ts`) should accurately represent when the value was measured.

```
int PWR_ObjAttrGetValue(PWR_Obj object, PWR_AttrName attr, void*
value, PWR_Time* ts)
```

Arguments		Description
IN	`PWR_Obj object`	The target object.
IN	`PWR_AttrName attr`	The target attribute. See section 4.5 for a list of available attributes
OUT	`void* value`	Pointer to caller-allocated storage, of 8 bytes, to hold the value read from the attribute.
OUT	`PWR_Time* ts`	Pointer to caller-allocated storage to hold the timestamp of when the value was read from the attribute. Pass in `NULL` if the timestamp is not needed.

Return Code(s)	Description
`PWR_RET_SUCCESS`	Upon SUCCESS.
`PWR_RET_NOT_IMPLEMENTED`	The requested attribute is not supported for the target object.
`PWR_RET_FAILURE`	Upon FAILURE.

Function Prototype for `PWR_ObjAttrSetValue()`

The `PWR_ObjAttrSetValue` function is provided to set the value of a single specified attribute (`PWR_AttrName attr`) of a single specified object (`PWR_Obj object`).

```
int PWR_ObjAttrSetValue(PWR_Obj object, PWR_AttrName attr, const
void* value)
```

Arguments		Description
IN	`PWR_Obj object`	The target object.
IN	`PWR_AttrName attr`	The target attribute. See section 4.5 for a list of available attributes.
IN	`const void* value`	Pointer to the 8 byte value to write to the attribute.

Return Code(s)	Description
PWR_RET_SUCCESS	Upon SUCCESS.
PWR_RET_NOT_IMPLEMENTED	The requested attribute is not supported for the target object.
PWR_RET_BAD_VALUE	The value was not appropriate for the target attribute.
PWR_RET_OUT_OF_RANGE	The value was out of range for the target attribute.
PWR_RET_FAILURE	Upon FAILURE.

Function Prototype for `PWR_ObjAttrStartTrackingValue()`

The `PWR_ObjAttrTrackValue` function is provided to start tracking for changes in the value of a single specified attribute (`PWR_AttrName attr`) of a single specified object (`PWR_Obj object`).

```
int PWR_ObjAttrStartTrackingValue(PWR_Obj object, PWR_AttrName
attr, void* trackingptr, PWR_AttrTracking type)
```

Arguments		Description
IN	`PWR_Obj object`	The target object.
IN	`PWR_AttrName attr`	The target attribute. See section 4.5 for a list of available attributes.
IN	`void* trackingptr`	Pointer to a fucntion to be called when the attribute changes.
IN	`PWR_AttrTracking type`	Specify `PWR_TRACKING_APIONLY` for changes only done through the Power API or `PWR_TRACKING_POLL` for polling from any changes (warning polling may have adverse performance ramifications.

Return Code(s)	Description
PWR_RET_SUCCESS	Upon SUCCESS.
PWR_RET_NOT_IMPLEMENTED	The requested attribute is not supported for the target object.
PWR_RET_FAILURE	Upon FAILURE.

Function Prototype for `PWR_ObjAttrStopTrackingValue()`

The `PWR_ObjAttrStopTrackingValue` function is provided to stop tracking the value of a single specified attribute (`PWR_AttrName attr`) of a single specified object (`PWR_Obj object`).

```
int PWR_ObjAttrStopTrackingValue(PWR_Obj object, PWR_AttrName
attr)
```

Arguments	Description
IN `PWR_Obj object`	The target object.
IN `PWR_AttrName attr`	The target attribute. See section 4.5 for a list of available attributes.

Return Code(s)	Description
PWR_RET_SUCCESS	Upon SUCCESS.
PWR_RET_FAILURE	Upon FAILURE.

Function Prototype for `PWR_StatusCreate()`

The `PWR_StatusCreate` function is provided to create the `PWR_Status` object that will be used in functions that perform multiple operations and potentially return individual statuses for each operation. It is up to the implementation to create the appropriate amount of storage for the `PWR_Status` structure based on the implementation and the number of statuses that will be held. `PWR_Status` objects can only be used in the context in which they are created, attempting to use a `PWR_Status` object in a context other than the one it was created for will result in an error. For example see `PWR_ObjAttrGetValues` on page 55. Note, `PWR_Status` is an opaque handle, its backing definition is determined by the implementor (see 4.1). It is intended that the implementation only allocate space for failed operations.

Errors are read from the `PWR_Status` by popping them off the structure which requires the structure to only be as large as the number of error returns require. When status objects are passed into a function, they are automatically cleared, therefore errors should always be checked on a status object before reuse. *Note to Users: Caution is advised when reusing status objects in multiple threads. Common thread safety practices must be followed to ensure that errors are properly caught. Creating status objects for each thread is advised to avoid potential race conditions.*

```
int PWR_StatusCreate(PWR_Cntxt context, PWR_Status* status)
```

Arguments		Description
IN	`PWR_Cntxt context`	The context in which the new status is to be used.
OUT	`PWR_Status* status`	The new status structure.

Return Code(s)	Description
`PWR_RET_SUCCESS`	Upon SUCCESS.
`PWR_RET_FAILURE`	Upon FAILURE.

Function Prototype for `PWR_StatusDestroy()`

The `PWR_StatusDestroy` function is provided to destroy the `PWR_Status` object created using `PWR_StatusCreate` (see page 52. Note, `PWR_Status` is an opaque handle, its backing definition is determined by the implementor (see 4.1).

```
int PWR_StatusDestroy(PWR_Status status)
```

Arguments		Description
IN	`PWR_Status status`	The `PWR_Status` structure the user wishes to destroy.

Return Code(s)	Description
`PWR_RET_SUCCESS`	Upon SUCCESS.
`PWR_RET_FAILURE`	Upon FAILURE.

Function Prototype for PWR_StatusPopError()

The PWR_StatusPopError function is provided to iterate through the PWR_Status object created using PWR_StatusCreate (see page 52) and populated using any of the function calls that leverage this structure. Using this method allows the PWR_Status structure to only grow as large as necessary storing only error returns. Note, PWR_Status is an opaque handle, its backing definition is determined by the implementor (see 4.1). The PWR_AttrAccessError structure that is returned will always have its obj, attr, and error fields set to the object, attribute, and error code associated with the error. The PWR_AttrAccessError structure's index field will only be set for attribute get functions (e.g., PWR_ObjAttrGetValues), and indicates the index in the output value array where the error occurred. For attribute get functions, errors are returned by PWR_StatusPopError in ascending order by index.

```
int PWR_StatusPopError(PWR_Status status, PWR_AttrAccessError*
error)
```

Arguments		Description
IN	PWR_Status status	The PWR_Status structure the user wishes to examine (iterate over).
OUT	PWR_AttrAccessError* error	Pointer to a PWR_AttrAccessError structure (see page 25) to hold the status that is popped from the PWR_Status structure.

Return Code(s)	Description
PWR_RET_SUCCESS	Upon SUCCESS.
PWR_RET_EMPTY	Returned when all errors have been popped.
PWR_RET_FAILURE	Upon FAILURE.

Function Prototype for PWR_StatusClear()

The PWR_StatusClear function is provided to clear a previously used PWR_Status object created using PWR_StatusCreate, (see page 52) basically allowing reuse of the same structure if multiple calls are executed and examined in sequence. Note, PWR_Status is an opaque handle, its backing definition is determined by the implementor (see 4.1).

```
int PWR_StatusClear(PWR_Status status)
```

Arguments	Description
IN PWR_Status status	The PWR_Status structure the user wishes to clear (reuse).

Return Code(s)	Description
PWR_RET_SUCCESS	Upon SUCCESS.
PWR_RET_FAILURE	Upon FAILURE.

Function Prototype for PWR_ObjAttrGetValues()

The PWR_ObjAttrGetValues function is provided to get the value of multiple specified attributes listed in the PWR_AttrName attrs[] array from a single specified object – **get multiple attribute values from a single object**. The timestamps returned in the PWR_Time ts[] array should accurately represent, and correspond sequentially, with the time each value returned was measured. If the function fails for one or more attributes, the PWR_Status status structure returned can be examined for additional information regarding the failure using PWR_StatusPopError (see page 54).

```
int PWR_ObjAttrGetValues(PWR_Obj object, int count, const PWR_
AttrName attrs[], void* values, PWR_Time ts[], PWR_Status status)
```

Arguments		Description
IN	`PWR_Obj object`	The target object.
IN	`int count`	The number of elements in the `attrs[]`, `*values`, and `ts[]` arrays.
IN	`const PWR_AttrName attrs[]`	The array of target attributes to read. See section 4.5 for a list of available attributes.
OUT	`void* values`	The array of values read, one value for each target attribute. This should point to caller-allocated storage of at least (`count * 8`) bytes. Upon success, the value read for attribute `attrs[i]` will be located at address (`values+(i*8)`).
OUT	`PWR_Time ts[]`	The array of timestamps, one timestamp for each value read. This should point to caller-allocated storage of at least (`count*sizeof(PWR_Time)`). Upon success, the timestamp of the value read for `attrs[i]` will be located at ts[i]. Pass in `NULL` if timestamps are not needed.
OUT	`PWR_Status status`	Upon `PWR_RET_FAILURE`, `status` contains information about each failure that occurred. Pass in `NULL` if failure information is not needed.

Return Code(s)	Description
`PWR_RET_SUCCESS`	Upon SUCCESS, all operations succeeded.
`PWR_RET_FAILURE`	Upon FAILURE, one or more operations failed. Examine `PWR_Status* status` to determine the operations that failed. All other operations succeeded.

Function Prototype for `PWR_ObjAttrSetValues()`

The `PWR_ObjAttrSetValues` function is provided to set the value of multiple specified attributes in the (`PWR_AttrName attrs[]`) array of a specified object – **set multiple attribute values of a single object**. If the function fails for one or more attributes, the `PWR_Status status` structure returned can be examined for additional information regarding the failure using `PWR_StatusPopError` (see page 54).

```
int PWR_ObjAttrSetValues(PWR_Obj object, int count, const PWR_
AttrName attrs[], const void* values, PWR_Status status)
```

Arguments		**Description**
IN	`PWR_Obj object`	The target object.
IN	`int count`	The number of elements in the `attrs[]` and `*values` arrays.
IN	`const PWR_AttrName attrs[]`	The array of target attributes to write. See section 4.5 for a list of available attributes.
IN	`const void* values`	The array of values to write, one value for each target attribute. The value to write to attribute `attrs[i]` is located at address (`values+(i*8)`).
OUT	`PWR_Status status`	Upon `PWR_RET_FAILURE`, `status` contains information about each failure that occurred. Pass in `NULL` if failure information is not needed.

Return Code(s)	**Description**
`PWR_RET_SUCCESS`	Upon SUCCESS, all operations succeeded.
`PWR_RET_FAILURE`	Upon FAILURE, one or more operations failed. Examine `PWR_Status* status` to determine the operations that failed. All other operations succeeded.

Function Prototype for `PWR_ObjAttrIsValid()`

The `PWR_ObjAttrIsValid` function is used to determine if a specified attribute (`PWR_AttrName attr`) is valid for the specified object.

```
int PWR_ObjAttrIsValid(PWR_Obj object, PWR_AttrName attr)
```

Arguments	Description
IN `PWR_Obj object`	The object that the user is acting on.
IN `PWR_AttrName attr`	The attribute the user wishes to confirm is valid for the specified object. See the `PWR_AttrName` type definition in section 4.5.

Return Code(s)	Description
`PWR_RET_SUCCESS`	Upon SUCCESS.
`PWR_RET_FAILURE`	Upon FAILURE.

Function Prototype for `PWR_GrpAttrGetValue()`

The `PWR_GrpAttrGetValue` function is provided to get the value of a single specified attribute (`PWR_AttrName attr`) from all the objects in a specified group (`PWR_Grp group`) – **get a single attribute value from multiple objects**. The timestamps returned in the `PWR_Time ts[]` array should accurately represent, and correspond sequentially, with the time each value returned was measured. If the function fails for one or more attributes, the `PWR_Status status` structure returned can be examined for additional information regarding the failure using `PWR_StatusPopError` (see page 54). `PWR_GrpAttrGetValue` will continue to attempt to gather values for the entire group, even if an error occurs for a subset of the members of that group.

```
int PWR_GrpAttrGetValue(PWR_Grp group, PWR_Attrgame attr, void*
values, PWR_Time ts[], PWR_Status status)
```

Arguments		Description
IN	`PWR_Grp group`	The target group.
IN	`PWR_Attrgame attr`	The target attribute to retrieve (get) from each object in the target group. See section 4.5 for a list of available attributes.
OUT	`void* values`	The array of attribute values retrieved, one value for each object in the target group. This should point to caller-allocated storage of at least (`PWR_GrpGetNumObjs() * 8`) bytes. Upon success, the value retrieved for the object at index `i` within the group will be located at address (`values+(i*8)`).
OUT	`PWR_Time ts[]`	The array of timestamps, one timestamp for each value retrieved. This should point to caller-allocated storage of at least (`PWR_GrpGetNumObjs()*sizeof(PWR_Time)`). Upon success, the timestamp of the value retrieved for the object at index `i` within the group will be located at ts[i]. Pass in `NULL` if timestamps are not needed.
OUT	`PWR_Status status`	Upon `PWR_RET_FAILURE`, `status` contains information about each failure that occurred. Pass in `NULL` if failure information is not needed.

Return Code(s)	Description
PWR_RET_SUCCESS	Upon SUCCESS, all operations succeeded.
PWR_RET_FAILURE	Upon FAILURE, one or more operations failed. Examine `PWR_Status* status` to determine the operations that failed. All other operations succeeded.

Function Prototype for `PWR_GrpAttrSetValue()`

The `PWR_GrpAttrSetValue` function is provided to set the value of a single specified attribute (`PWR_AttrName attr`) of each object in a specified group – **set a single attribute value on multiple objects**. If the function fails for one or more attributes, the `PWR_Status status` structure returned can be examined for additional information regarding the failure using `PWR_StatusPopError` (see page 54). `PWR_GrpAttrSetValue` will continue to attempt to set values for the entire group, even if an error occurs for a subset of the members of that group.

```
int PWR_GrpAttrSetValue(PWR_Grp group, PWR_AttrName attr, const
void* value, PWR_Status status)
```

Arguments		Description
IN	PWR_Grp group	The target group.
IN	PWR_AttrName attr	The target attribute to set for each object in the target group. See section 4.5 for a list of available attributes.
IN	const void* value	The pointer to a single 8 byte attribute value to set for each object in the target group.
OUT	PWR_Status status	Upon `PWR_RET_FAILURE`, `status` contains information about each failure that occurred. Pass in `NULL` if failure information is not needed.

Return Code(s)	Description
PWR_RET_SUCCESS	Upon SUCCESS, all operations succeeded.
PWR_RET_FAILURE	Upon FAILURE, one or more operations failed. Examine PWR_Status* status to determine the operations that failed. All other operations succeeded.

Function Prototype for PWR_GrpAttrGetValues()

The PWR_GrpAttrGetValues function is provided to get the value of multiple specified attributes listed in the PWR_AttrName attrs[] array from each object in a specified group – **get multiple attribute values from multiple objects**. The timestamps returned in the PWR_Time ts[] array should accurately represent, and correspond sequentially, with the time each value returned was measured. If the function fails for one or more attributes, the PWR_Status status structure returned can be examined for additional information regarding the failure using PWR_StatusPopError (see page 54). PWR_GrpAttrGetValues will continue to attempt to gather values for the entire group, even if an error occurs for a subset of the members or attributes requested in the object group.

```
int PWR_GrpAttrGetValues(PWR_Grp group, int count, const PWR_
AttrName attrs[], void* values, PWR_Time ts[], PWR_Status status)
```

Arguments		Description
IN	PWR_Grp group	The target group.
IN	int count	The number of elements in the attrs[] array.
IN	const PWR_AttrName attrs[]	he array specifying the set of target attributes to read for each object in the target group. See section 4.5 for a list of available attributes.
OUT	void* values	The array of attribute values retrieved. This should point to caller-allocated storage of at least (PWR_GrpGetNumObjs()*count*8) bytes. Upon success, the value read for attribute attrs[i] for the object at index j within the group will be located at address (values+(j*count*8)+(i*8)).
Output	PWR_Time ts[]	The array of timestamps, one timestamp for each value retrieved. This should point to caller-allocated storage of at least (PWR_GrpGetNumObjs()*count*sizeof(PWR_Time)). Upon success, the timestamp of the value retrieved for attribute attrs[i] for the object at index j within the group will be located at ts[(j*count)+i]. Pass in NULL if timestamps are not needed.
OUT	PWR_Status status	Upon PWR_RET_FAILURE, status contains information about each failure that occurred. Pass in NULL if failure information is not needed.

Return Code(s)	Description
PWR_RET_SUCCESS	Upon SUCCESS, all operations succeeded.
PWR_RET_FAILURE	Upon FAILURE, one or more operations failed. Examine PWR_Status* status to determine the operations that failed. All other operations succeeded.

Function Prototype for PWR_GrpAttrSetValues()

The PWR_GrpAttrSetValues function is provided to set the value of multiple specified attributes listed in the (PWR_AttrName attrs[]) array of each object in a specified group – **set multiple attribute values on multiple objects**. If the function fails for one or more attributes, the PWR_Status status structure returned can be examined for additional information regarding the failure using PWR_StatusPopError (see page 54). PWR_GrpAttrSetValues will continue to attempt to set values for the entire group and requested attributes, even if an error occurs for a subset of the members or attributes of that object group.

```
int PWR_GrpAttrSetValues(PWR_Grp group, int count, const PWR_
AttrName attrs[], const void* values, PWR_Status status)
```

Arguments		Description
IN	PWR_Grp group	The target group.
IN	int count	The number of elements in the attrs[] and *values arrays.
IN	const PWR_AttrName attrs[]	The array specifying the set of target attributes to set for each object in the target group. See section 4.5 for a list of available at tributes.
IN	const void* values	The array of attribute values to set for each object in the group. The value to write to attribute attrs[i] of each object is located at address (values+(i*8)).
OUT	PWR_Status status	Upon PWR_RET_FAILURE, status contains information about each failure that occurred. Pass in NULL if failure information is not needed.

Return Code(s)	Description
PWR_RET_SUCCESS	Upon SUCCESS, all operations succeeded.
PWR_RET_FAILURE	Upon FAILURE, one or more operations failed. Examine PWR_Status* status to determine the operations that failed. All other operations succeeded.

5.5 Metadata Functions

The metadata functions provide an interface for getting more descriptive information about an object or attribute, such as estimated measurement accuracy or the list of valid values for a given attribute. This information is often useful for getting a better understanding of the meaning of objects and attributes and how to interpret the values read from attributes. While most metadata is read-only information, some metadata is potentially configurable, such as the underlying power sampling rate used to calculate PWR_ATTR_ENERGY values.

Table 5.2 on page 29 lists the available types of metadata. Not all of the metadata items listed will be available for every object and attribute pair. The exact set is dependent on the capabilities of the underlying hardware and Power API implementation. If a requested metadata item is not available a PWR_RET_NO_ATTRIB error is returned at runtime.

The majority of metadata items will require that both an object instance and attribute name pair be specified, but a few may be defined for object instances alone. For example, the metadata strings PWR_MD_NAME, PWR_MD_DESC, and PWR_MD_VENDOR_INFO may be available for individual object instances, with no associated attribute name specified. In these cases, the attribute name requested should be set to PWR_ATTR_NOT_SPECIFIED. One important use case for these informational strings, especially the PWR_MD_VENDOR_INFO string, is for a Power API user to capture these strings with each run to record configuration and provenance information. For example, a user may chose to log the PWR_MD_VENDOR_INFO string for the top-level platform object in the output of each run.

The metadata interface consists of three functions. The PWR_ObjAttrGetMeta and PWR_ObjAttrSetMeta functions allow metadata values to be retrieved and set, respectively. The third function, PWR_MetaValueAtIndex, provides a way to enumerate through an attribute's list of available values. This is useful for attributes that have a small, well-defined set of discrete values (e.g., PWR_ATTR_PSTATE). It is expected that where a set of discrete values can be described in a logical order that the index ordering is from smallest (lowest) to largest (highest) value. The remainder of this section describes the metadata functions in more detail.

Function Prototype for PWR_ObjAttrGetMeta()

The PWR_ObjAttrGetMeta function returns the requested metadata item for the specified object or object and attribute name pair. The caller must allocate enough storage to hold the returned metadata value and pass a pointer to the storage in the value argument. The required size can be determined by consulting the type column of Table 5.2. In the case of string metadata items (i.e., type char *), the required string length can be determined by getting the appropriate length metadata item, which is the original metadata name with the _LEN suffix added. For example, the required string length for the PWR_MD_VENDOR_INFO string can be determined by retrieving the PWR_MD_VENDOR_INFO_LEN metadata item.

```
PWR_ObjAttrGetMeta(PWR_Obj obj, iPWR_AttrName attr, PWR_MetaName
meta, void* value)
```

Arguments		Description
IN	`PWR_Obj obj`	The target object.
IN	`iPWR_AttrName attr`	The target attribute. See the `PWR_AttrName` type definition in Section 4.5 for the list of possible attributes. If object-only metadata is being requested, this argument should be set to `PWR_ATTR_NOT_SPECIFIED`.
IN	`PWR_MetaName meta`	The target metadata item to get. See the `PWR_MetaName` type definition in Section 4.6 for the list of possible metadata items, with detailed descriptions provided in Table 5.2.
OUT	`void* value`	Pointer to the caller allocated storage to hold the value of the requested metadata item. See Table 5.2 for type information.

Return Code(s)	Description
`PWR_RET_SUCCESS`	Upon SUCCESS.
`PWR_RET_NO_ATTRIB`	The attribute specified is not implemented.
`PWR_RET_NO_META`	The metadata specified is not implemented.
`PWR_RET_FAILURE`	Upon FAILURE.

Function Prototype for `PWR_ObjAttrSetMeta()`

The `PWR_ObjAttrSetMeta` function sets the specified metadata item for the target object or object and attribute name pair. The caller must pass a pointer to the new value for the specified metadata item in the `value` argument. The required type for the value can be determined by consulting the type column of Table 5.2. In the case of string metadata items (i.e., type `char *`), the maximum string length can be determined by getting the appropriate length metadata item, which is the original metadata name with the _LEN suffix added. For example, the maximum string length for the PWR_MD_VENDOR_INFO string can be determined by retrieving the PWR_MD_VENDOR_INFO_LEN metadata item.

```
int PWR_ObjAttrSetMeta(PWR_Obj obj, PWR_AttrName attr, PWR_MetaName meta, const void* value)
```

Arguments		Description
IN	PWR_Obj obj	The target object.
IN	PWR_AttrName attr	The target attribute. See the PWR_AttrName type definition in Section 4.5 for the list of possible attributes. If object-only metadatais being set, this argument should be set to PWR_ATTR_NOT_SPECIFIED.
IN	PWR_MetaName meta	The target metadata item to set. See the PWR_MetaName type definition in Section 4.6 for the list of possible metadata items, with detailed descriptions provided in Table 5.2.
IN	const void* value	Pointer to the new value for the metadata item. See Table 5.2 for type information.

Return Code(s)	Description
PWR_RET_NO_ATTRIB	The attribute specified is not implemented.
PWR_RET_NO_META	The metadata specified is not implemented.
PWR_RET_READ_ONLY	The metadata specified is not settable.
PWR_RET_BAD_VALUE	The value specified is not valid.
PWR_RET_FAILURE	Upon FAILURE.

Function Prototype for PWR_MetaValueAtIndex()

The PWR_MetaValueAtIndex function allows the available values for a given attribute to be enumerated. It is assumed that the set of valid values is static and has size equal to the value returned by the PWR_MD_NUM metadata item.

Once the value of PWR_MD_NUM is known, PWR_MetaValueAtIndex() can be called repeatedly with index from 0 to PWR_MD_NUM - 1 to retrieve the list of valid values for the target attribute. Each call will return the value at the specified index as well as a human-readable string representing the value in human readable format. If an attribute is not enumerable, then PWR_MD_NUM will return 0. In general any attribute that does not have a small set of discrete valid values will return 0 when PWR_MD_NUM is requested, to indicate that the attribute is not enumerable.

```
int PWR_MetaValueAtIndex(PWR_Obj obj, PWR_AttrName attr, unsigned
int index, void* value, char* value_str)
```

Arguments		Description
IN	PWR_Obj obj	The target object.
IN	PWR_AttrName attr	The target attribute. See the PWR_AttrName type definition in Section 4.5 for the list of possible attributes.
IN	unsigned int index	The index of the metadata item value to look up. The PWR_MD_NUM metadata item returns the number of possible values, indexed from 0 to PWR_MD_NUM - 1.
OUT	void* value	Pointer to the caller allocated storage to hold the value of the requested metadata item value. See Table 5.2 for type inform ation. The storage must be sized appropriately for the metadata value type. If the value is not required, this argument should be set to NULL.
OUT	char* value_str	Pointer to the caller allocated storage to hold the human-readable C-style NULL-terminated ASCII string representing the metadata item value. The storage passed in must have size in bytes of at least the value returned by the PWR_MD_VALUE_LEN metadata item. If the string representation is not required this argument should be set to NULL.

Return Code(s)	Description
PWR_RET_SUCCESS	Upon SUCCESS.
PWR_RET_NO_ATTRIB	The attribute specified is not implemented.
PWR_RET_BAD_INDEX	The index specified is not valid.
PWR_RET_FAILURE	Upon FAILURE.

5.6 Statistics Functions

The statistics functions provide an interface to generate statistics related to specific attributes of an object or group. The interface allows for generating statistics somewhat real-time or mining historic statistics, assuming that the necessary data is retained. The interface for collecting historic statistics is much more straight forward and can be accomplished with a single call, `PWR_ObjGetStat` for a single object and `PWR_GrpGetStats` for a group of objects. The interface for collecting real-time statistics is designed to interface with hardware or layers of software that require a notification of when information collection should begin and when it can be terminated. The requested statistic can then be mined for this window of time, even while the window remains open. The sequence of calls for mining real-time statistics is as follows. The user creates a statistic object using the `PWR_ObjCreateStat` call when collecting a statistic on a single object or the `PWR_GrpCreateStat` call when the statistic is to be collected on a group of objects. Basically, a tuple of information is provided, an object or group, the attribute (`PWR_ATTR_POWER` for example, see page 24) that the user would like the statistic for and the statistic (`PWR_ATTR_STAT_AVG` for example, see page 29). Notice that the statistic to be collected is part of the required parameters for creating a statistics object, while it is provided at the time of retrieval when collecting historic statistics. The reason for this approach is that the underlying hardware or software layer needs to understand what information to start collecting to support the requested statistic. Buffers are typically a limiting factor in the capabilities that can be supported by an implementation. Requiring an implementation to collect the data necessary for any potential statistic could require a great deal of space. Once a statistics object is created (for an object or a group) the user indicates the beginning of the window by calling `PWR_StatStart`. Once `PWR_StatStart` is called the user can retrieve the statistic information associated with the statistics object by calling `PWR_StatGetValue`, when the statistics object was created for a single object, or `PWR_StatGetValues`, when the statistics object was created for a group of objects. The start time is always the time that the user calls `PWR_StatStart` on the statistics object. The user can call `PWR_StatGetValue` or `PWR_StatGetValues` as many times as they wish prior to calling `PWR_StatStop`. If `PWR_StatStop` has not been called, the stop time is the time the user calls `PWR_StatGetValue` or `PWR_StatGetValues`. Once `PWR_StatStop` has been called the stop time if fixed for that statistics object. Essentially, the implementation at this time has everything it needs to calculate the return value or values for PWR_StatGetValue or PWR_StatGetValues. The user is responsible for checking the start and stop times returned along with the statistics value. The start and stop times may be different for two reasons. In the normal case, the implementation is required to return start and stop times that accurately represent when the actual data was sampled that was used in calculting the statistics value. As such, the returned values could differ from the times set by the real-time statistics functions. In the abnormal case, the start time, and possibly the stop time, could differ more significantly from the times PWR_StatStart and PWR_StatStop were called or the stop time determined by calling either of the PWR_StatGetValue or PWR_StatGetValues functions before PWR_StatStop has been called. If this occurs, due to a resource exhaustion issue for example, the implementation is required to either return a failure or return a statistics value and the accurate time values representing the statistics value returned along with a warning indicating that the time window has been truncated. A truncated time-frame is still required to as closely as possible represent the data collection time the statistic is generated based on. It is then up to the user to determine if the value returned is useful or not. Statistics objects can be re-used by calling PWR_StatClear, which indicates to the implementation that any data retained associated with the statistic object can be released. To begin another statistics window the user repeats the process just outlined. When the user is done with a statistics object they should call PWR_StatDestroy.

Function Prototype for PWR_ObjGetStat()

The PWR_ObjGetStat function is used to retrieve a historic statistic using an object, attribute, statistic tuple. Note that the PWR_ObjGetStat call operates on single objects only, not groups of objects. The PWR_ObjGetStat is a standalone call is used for historic data collection only. To retrieve a statistic from a group of objects, the PWR_GrpGetStats call on page 72 should be used.

```
PWR_ObjGetStat(PWR_Obj object, PWR_AttrName name, PWR_AttrStat
statistic, PWR_TimePeriod* statTime, double* value)
```

Arguments		Description
IN	PWR_Obj object	The object to collect the statistic for (part of the object, attribute statistic triple
IN	PWR_AttrName name	The attribute to act on, see the PWR_AttrName type definition in section 4.5.
IN	PWR_AttrStat statistic	ified attribute, see PWR_AttrStat type definition in section 4.9.
IN/OUT	PWR_TimePeriod* statTime	Time structure that initially must contain the times (start, stop and instant if appropriate) requested by the user (Input) and the times, possibly different, representing the period of the statistic data returned (Output), see page 28.
OUT	double* value	pointer to space (double) to store the statistic

Return Code(s)	Description
PWR_RET_SUCCESS	Upon SUCCESS.
PWR_RET_FAILURE	Upon FAILURE.

Function Prototype for PWR_GrpGetStats()

The PWR_GrpGetStats function is used to retrieve historic statistic for a group of objects. Each object in the group is combined with the attribute and statistic specified to form the object, attribute, statistic tuple. Note that the PWR_GrpGetStats call operates on one or more objects in a group. The PWR_GrpGetStats is a standalone call is used for historic data collection only. To retrieve a statistic from a single object, the PWR_ObjGetStat call on page 71 should be used.

```
int PWR_GrpGetStats(PWR_Grp group, PWR_AttrName name, PWR_
AttrStat statistic, PWR_TimePeriod* statTime, double values[],
PWR_TimePeriod statTimes[])
```

Arguments		Description
IN	PWR_Grp group	The group to collect the statistic for. Each object in the group forms the object, attribute, statistic triple.
IN	PWR_AttrName name	The attribute to act on, see the PWR_AttrName type definition in section 4.5.
IN	PWR_AttrStat statistic	The desired statistic for the specified attribute, see PWR_AttrStat type definition in section 4.9.
IN	PWR_TimePeriod* statTime	Time structure that must contain the times (start, stop and instant if appropriate) requested by the user. Note this is Input only, see page **??**.
OUT	double values[]	Space (of double) allocated by user to store an array of statistic values
OUT	PWR_TimePeriod statTimes[]	Space allocated by user to hold an array of time structures representing the actual times associated with each statistic value returned in values[], see page 28.

Return Code(s)	Description
PWR_RET_SUCCESS	SUCCESS.
PWR_RET_FAILURE	Upon FAILURE.

Function Prototype for PWR_ObjCreateStat()

The PWR_ObjCreateStat function is used to create a statistics object that will be used for real-time statistics gathering operations for a single object. The user specifies the **object**, attribute, statistic tuple that all subsequent requests using the statistics object created will be based on. Note, this call is not used for historic statistic gathering, see PWR_ObjGetStat on page 71 and PWR_GrpGetStats on page 72.

```
int PWR_ObjCreateStat(PWR_Obj object, PWR_AttrName name, PWR_
AttrStat statistic, PWR_Stat* stat)
```

Arguments		Description
IN	PWR_Obj object	The object to act on.
IN	PWR_AttrName name	The attribute to act on, see the PWR_AttrName type definition in section 4.5.
IN	PWR_AttrStat statistic	The desired statistic for the specified attribute, see PWR_AttrStat type definition in section 4.9.
OUT	PWR_Stat* stat	The stat for the object, attribute, statistic triple specified.

Return Code(s)	Description
PWR_RET_SUCCESS	Upon SUCCESS, valid stat is created.
PWR_RET_FAILURE	Upon FAILURE.

Function Prototype for PWR_GrpCreateStat()

The PWR_GrpCreateStat function is used to create a statistics object that will be used for real-time statistics gathering operations for a group of objects. The user specifies the **group**, attribute, statistic tuple that all subsequent requests using the statistics object created will be based on. Note, this call is not used for historic statistic gathering, see PWR_ObjGetStat on page 71 and PWR_GrpGetStats on page 72.

```
int PWR_GrpCreateStat(PWR_Grp group, PWR_AttrName name, PWR_
AttrStat statistic, PWR_Stat* stat)
```

Arguments		Description
IN	PWR_Grp group	The group to act on.
IN	PWR_AttrName name	The attribute to act on, see the PWR_AttrName type definition in section 4.5.
IN	PWR_AttrStat statistic	The desired statistic for the specified attribute, see PWR_AttrStat type definition in section 4.9.
OUT	PWR_Stat* stat	The stat for the group, attribute, statistic triple specified.

Return Code(s)	Description
PWR_RET_SUCCESS	Upon SUCCESS, valid stat is created.
PWR_RET_FAILURE	Upon FAILURE.

Function Prototype for `PWR_StatStart()`

The `PWR_StatStart` function is used to indicate to a device or software layer to start the window of time that the statistic requested will be calculated over. The `PWR_StatStart` function is used for real-time statistics gathering only.

```
int PWR_StatStart(PWR_Stat statObj)
```

Arguments	Description
IN `PWR_Stat statObj`	The statistics object to begin collecting the specified statistic for (specified in `PWR_ObjCreateStat` or `PWR_GrpCreateStat`).

Return Code(s)	Description
PWR_RET_SUCCESS	Upon SUCCESS.
PWR_RET_FAILURE	Upon FAILURE.

Function Prototype for `PWR_StatStop()`

The `PWR_StatStop` function is used to indicate to a device or software layer to stop the window of time that the statistic requested will be calculated over. The `PWR_StatStop` function is used for real-time statistics gathering only.

```
int PWR_StatStop(PWR_Stat statObj)
```

Arguments	Description
IN `PWR_Stat statObj`	The statistics object to stop collecting the specified statistic for (specified in `PWR_ObjCreateStat` or `PWR_GrpCreateStat`).

Return Code(s)	Description
PWR_RET_SUCCESS	Upon SUCCESS.
PWR_RET_FAILURE	Upon FAILURE.

Function Prototype for `PWR_StatClear()`

The `PWR_StatClear` function is used to indicate to a device or software layer to clear or reset the window of time that the statistic requested will be calculated over. The clear effectively restarts the window, so there is no need to call `PWR_StatStart` again. The `PWR_StatClear` function is used for real-time statistics gathering only.

```
int PWR_StatClear(PWR_Stat statObj)
```

Arguments	Description
IN PWR_Stat statObj	The statistics object to clear (effectively reset) for the specified statistic (specified in `PWR_ObjCreateStat` or `PWR_GrpCreateStat`).

Return Code(s)	Description
PWR_RET_SUCCESS	Upon SUCCESS.
PWR_RET_FAILURE	Upon FAILURE.

Function Prototype for `PWR_StatGetValue()`

The `PWR_StatGetValue` function is used to retrieve the statistic and related time stamp information from the statistics object created using `PWR_ObjCreateStat`. Note that the `PWR_StatGetValue` call operates on single objects only, not groups of objects. The start time for the window the statistic is calculated over is set by calling `PWR_StatStart`. The stop time is set by either calling this function, `PWR_StatGetValue`, or set for the statistics object by calling `PWR_StatStop`. Each time `PWR_StatGetValue` is called prior to calling `PWR_StatStop` the time `PWR_StatGetValue` is called is used as the stop time for the statistics calculation. From the specification standpoint, there is no limit to how often `PWR_StatGetValue` can be called. The start, stop and, depending on the statistic requested, the instant time values returned should as accurately as possible represent the time-stamps of the data used in the statistics value returned. The PWR_StatGetValue function is used for real-time statistics gathering only. If a single value return is desired for a group of objects, the PWR_StatGetReduce call on page 78 should be used.

```
int PWR_StatGetValue(PWR_Stat statObj, double* value, PWR_
TimePeriod* statTimes)
```

Arguments	Description
IN PWR_Stat statObj	The statistics object to collect the statistic for (the object, attribute stat triple is specified in PWR_ObjCreateStat).
OUT double* value	pointer to space (double) to store the statistic
OUT PWR_TimePeriod* statTimes	Time structure that contains the timestamps pertinent to the specific statistic value, see page 28.

Return Code(s)	Description
PWR_RET_SUCCESS	Upon SUCCESS.
PWR_RET_FAILURE	Upon FAILURE.
PWR_RET_WARN_TRUNC	When the time window has been truncated by the implementation, start and stop times may differ significantly from those set by the interface.

Function Prototype for PWR_StatGetValues()

The PWR_StatGetValues function is used to retrieve the statistic and related time stamp information from the statistics object(s) created using PWR_GrpCreateStat. Note that the PWR_StatGetValues call operates on one or more objects in the group specified in the PWR_GrpCreateStat call. The start time for the window the statistic is calculated over is set by calling PWR_StatStart. The stop time is set by either calling this function, PWR_StatGetValues, or set for the statistics object by calling PWR_StatStop. Each time PWR_StatGetValues is called prior to calling PWR_StatStop the time PWR_StatGetValues is called is used as the stop time for the statistics calculation. From the specification standpoint, there is no limit to how often PWR_StatGetValues can be called. The start, stop and, depending on the statistic requested, the instant time values for each individual object returned (in the Output PWR_TimePeriod structure) should as accurately as possible represent the time-stamps of the data used in the statistics values returned. The PWR_StatGetValues function is used for real-time statistics gathering only. If a single value return is desired for a group of objects, the PWR_StatGetReduce call on page 78 should be used.

```
int PWR_StatGetValues(PWR_Stat statObj, double values[], PWR_
TimePeriod statTimes[])
```

Arguments		Description
IN	PWR_Stat statObj	The statistics object to collect the statistic for (the group, attribute stat triple is specified in PWR_GrpCreateStat).
OUT	double values[]	Space allocated by user to hold array of values (statistics).
OUT	PWR_TimePeriod statTimes[]	Space allocated by user to hold array of time structures that contains the timestamps pertinent to each specific statistic value, see page **??**

Return Code(s)	Description
PWR_RET_SUCCESS	Upon SUCCESS.
PWR_RET_FAILURE	Upon FAILURE.
PWR_RET_WARN_TRUNC	When the time window has been truncated by the implementation, start and stop times may differ significantly from those set by the interface.

Function Prototype for PWR_StatGetReduce()

The PWR_StatGetReduce function is used to reduce a set of per-object statistics down into a single returned value. The inputs are a PWR_Stat object, and a reduction operation. The reduction operation can be thought of as occurring in two phases. In the first phase, a statistic is calculated for each object associated with the input `PWR_Stat`, one statistic value per object. The objects, target attribute, and desired statistic to calculate are specified when the `PWR_Stat` is created. In the second phase, the set of statistic values calculated in the first phase are combined into a single result value. How this occurs is determined by the reduction operation that was specified by the caller. For example, the `PWR_ATTR_STAT_AVG` reduction operation returns the average of the per-object statistics calculated in the first phase. The start time for the window the statistic is calculated over is set by calling `PWR_StatStart`. The stop time used for the statistics calculated in the first phase are based on the time this function is called, or set for the statistics object from a previous call to `PWR_StatStop`. `PWR_StatGetReduce` is provided such that optimizations may be possible when gathering the statistics of each member in a group of objects. An example of such an operation would be calculating an average, where gathering the values is done through a tree topology overlay network, where averages can be calculated at each parent of multiple children in the tree. Note that the implementation of `PWR_StatGetReduce` can be done in its more simplistic form by calling `PWR_StatGetValues` and performing the required operation on the returned set of values to return the requested reduction operation.

```
int PWR_StatGetReduce(PWR_Stat statObj, PWR_AttrStat reduceOp,
int* index, double* result, PWR_Time* instant)
```

Arguments		Description
IN	PWR_Stat statObj	The statistics object to collect the statistic for (the object group, attribute, stat triple is specified in PWR_GrpCreateStat).
IN	PWR_AttrStat reduceOp	The reduction operation to perform.
OUT	int* index	The index of the object in the statObj's associated object group that provided the reduction result. This value is only set for reduction operations where it makes sense, such as PWR_ATTR_STAT_MIN and PWR_ATTR_STAT_MAX.
OUT	double* result	The result of the reduction operation, which is always a single double value.
OUT	PWR_Time* instant	For statistics where a point in time that the value occured is valid (e.g. max and min), this is the timestamp when that value was observed.

Return Code(s)	Description
PWR_RET_SUCCESS	Upon SUCCESS.
PWR_RET_FAILURE	Upon FAILURE.
PWR_RET_WARN_TRUNC	When the time window has been truncated by the implementation.

Function Prototype for PWR_GrpGetReduce()

The PWR_GrpGetReduce function is used to reduce a set of per-object statistics down into a single returned value. Unlike PWR_StatGetReduce that is used for real time statistics gathering, PWR_GrpGetReduce is meant to gather statistics for historical data. Therefore, this call is much like the PWR_GrpGetStats function, with an added reduction. The inputs are a PWR_Grp object, an attribute, a statistic, a reduction operation and a time period. The reduction operation can be thought of as occurring in two phases. In the first phase, a statistic is calculated for each object associated with the input group, one statistic value per object. The objects, target attribute, and desired statistic to calculate are specified as inputs to this function.. In the second phase, the set of statistic values calculated in the first phase are combined into a single result value. How this occurs is determined by the reduction operation that was specified by the caller. For example, the `PWR_ATTR_STAT_AVG` reduction operation returns the average of the per-object statistics calculated in the first phase. Upon success, the returned `PWR_TimePeriod` structure will have its time fields set to the timestamps that are most closely associated with the result of the reduction operation. For certain reduction operations, some timestamps in the returned `PWR_TimePeriod` may not be valid output. For example, in the case of a averaging reduction, an associated "instant" timestamp is not a useful value. For minimum and maximum operations, the "instant" timestamp is useful and will represent the time at which the maximum or minimum was observed. In all cases the start and stop timestamps in the `PWR_TimePeriod` will represent the time window over which the the value was calculated. `PWR_GrpGetReduce` is provided such that optimizations may be possible when gathering the statistics of each member in a group of objects. An example of such an operation would be calculating an average, where gathering the values is done through a tree topology overlay network, where averages can be calculated at each parent of multiple children in the tree. Note that the implementation of `PWR_GrpGetReduce` can be done in its more simplistic form by calling `PWR_GrpGetStats` and performing the required operation on the returned set of values to return the requested reduction operation.

```
int PWR_GrpGetReduce(PWR_Grp group, PWR_AttrName name, PWR_
AttrStat statistic, PWR_AttrStat reduceOp, PWR_TimePeriod
statTime, int* index, double* result, PWR_TimePeriod* resultTime)
```

Arguments		Description
IN	PWR_Grp group	The group to collect the statistic for. Each object in the group forms the object, attribute, statistic triple.
IN	PWR_AttrName name	The attribute to act on, see the PWR_AttrName type definition in section 4.5.
IN	PWR_AttrStat statistic	The desired statistic for the specified attribute, see PWR_AttrStat type definition in section 4.9.
IN	PWR_AttrStat reduceOp	The reduction operation to perform.
IN	PWR_TimePeriod statTime	Time structure that must contain the times (start, stop and instant if appropriate) requested by the user. Note this is Input only, see page 28.
OUT	int* index	The index of the object in the statObj's associated object group that provided the reduction result. This value is only set for reduction operations where it makes sense, such as PWR_ATTR_STAT_MIN and PWR_ATTR_STAT_MAX.
OUT	double* result	The result of the reduction operation, which is always a single double value.
OUT	PWR_TimePeriod* resultTime	The time period that the results are valid for. Note that this may diverge from the input time period if results for the exact time period are not available. This time period will also contain the instant that the statistic was observed for cases where this makes sense, such as PWR_ATTR_STAT_MIN and PWR_ATTR_STAT_MAX.

Return Code(s)	Description
PWR_RET_SUCCESS	Upon SUCCESS.
PWR_RET_FAILURE	Upon FAILURE.
PWR_RET_WARN_TRUNC	When the time window has been truncated by the implementation.

Function Prototype for PWR_StatDestroy()

The PWR_StatDestroy function is used to destroy (clean up) the statistics pointer created using PWR_ObjCreateStat or PWR_GrpCreateStat.

```
int PWR_StatDestroy(PWR_Stat statObj)
```

Arguments	Description
IN PWR_Stat statObj	The statistics object to destroy (clean up)

Return Code(s)	Description
PWR_RET_SUCCESS	Upon SUCCESS.
PWR_RET_FAILURE	Upon FAILURE.

5.7 Version Functions

The PWR_GetMajorVersion and PWR_GetMinorVersion functions are used to get the major and minor portions of the specification version supported by the implementation. Users can make decisions regarding available functionality based on the version number supported.

Function Prototype for PWR_GetMajorVersion()

The PWR_GetMajorVersion function is used to get the major version number portion of the version number of the specification supported by the implementation.

```
int PWR_GetMajorVersion()
```

Return Code(s)	Description
`int`	Upon SUCCESS, integer representation of major portion of version number
`PWR_RET_FAILURE`	Upon FAILURE

Function Prototype for `PWR_GetMinorVersion()`

The `PWR_GetMinorVersion` function is used to get the minor version portion of the version number of the specification supported by the implementation.

```
int PWR_GetMinorVersion()
```

Return Code(s)	Description
`int`	Upon SUCCESS, integer representation of minor portion of version number.
`PWR_RET_FAILURE`	Upon FAILURE.

5.8 Big List of Attributes

The following is the master list of Attributes available to the user. The attributes valid for specific interfaces are listed in the appropriate section in Chapter 7.

Table 5.1: Complete list of all supported attributes

Attribute, Get/Set, Type	Description
`PWR_ATTR_PstateDesc` . Get/Set . `uint64_t`	The current P-state for the object specified (typically processors but for use with other component types when applicable).
`PWR_ATTR_CstateDesc` . Get/Set . `uint64_t`	The current C-state for the object specified (typically processors but for use with other component types when applicable).

Continued on next page

Table 5.1 – continued from previous page

Attribute, Get/Set, Type	Description
`PWR_ATTR_CstateLimitDesc` . Get/Set . `uint64_t`	The lowest C-state allowed for the object specified (typically processors but for use with other component types when applicable).
`PWR_ATTR_SstateDesc` . Get/Set . `uint64_t`	The current S-state for the object specified (typically processors but for use with other component types when applicable).
`PWR_ATTR_CurrentDesc` . Get . `double`	Discrete current value in amps. The current value should be the value measured as close as possible to the time of the function call.
`PWR_ATTR_VoltageDesc` . Get . `double`	Discrete voltage value in volts. The voltage value should be the value measured as close as possible to the time of the function call.
`PWR_ATTR_PowerDesc` . Get . `double`	Discrete power value in watts. The power value should be the value measured as close as possible to the time of the function call.
`PWR_ATTR_MinPowerDesc` . Get/Set . `double`	Minimum power limit (floor, lower bound) for the specified object in watts.
`PWR_ATTR_MaxPowerDesc` . Get/Set . `double`	Maximum power limit (ceiling, upper bound) for the specified object (as in power cap) in watts.
`PWR_ATTR_FreqDesc` . Get/Set . `double`	The current operating frequency value for the specified object in Hz (cycles per second).
`PWR_ATTR_FreqLimitMinDesc` . Get/Set . `double`	Minimum operating frequency limit for the specified object in Hz (cycles per second).

Continued on next page

Table 5.1 – continued from previous page

Attribute, Get/Set, Type	Description
PWR_ATTR_FreqLimitMaxDesc . Get/Set . double	Maximum operating frequency limit for the specified object in Hz (cycles per second).
PWR_ATTR_EnergyDesc . Get . double	The cumulative energy used by the specified object in joules. Note that two attribute get calls are typically required to obtain the energy consumed by the specified object. Subtracting the energy value obtained from the first call from the energy value obtained from the second call produces the energy used for the object from the timestamp of the first value through the timestamp of the second value.
PWR_ATTR_TempDesc . Get . double	The current temperature value for the specified object in degrees Celsius.
PWR_ATTR_OSIdDesc . Get . double	The operating system ID that corresponds to the object. For example, a runtime system may need to figure out which Power API PWR_OBJ_CORE objects correspond to the cores that it is controlling. This attribute provides a linkage between Power API objects and operating system IDs.
PWR_ATTR_ThrottledIdDesc . Get . double	The cumulative time in nanoseconds that the specified object's performance was purposefully slowed in order to meet some constraint, such as a power cap.

Continued on next page

Table 5.1 – continued from previous page

Attribute, Get/Set, Type	Description
`PWR_ATTR_ThrottledCountIdDesc` . Get . `double`	The cumulative count of the number of times that the specified object's performance was purposefully slowed in order to meet some constraint, such as a power cap.
`PWR_ATTR_GovDesc` . Get . `double`	Power related governor capability exposed through the operating system interface.

5.9 Big List of Metadata

Table 5.2: Complete List of All Metadata Names

Attribute, Get/Set, Type	Description
`PWR_MD_num` . Get . `uint64_t`	Number of values supported. This is only relevant for attributes with a discrete set of values (e.g., `PWR_ATTR_PSTATE`). Other attributes return 0.
`PWR_MD_min` . Get . Same type as attribute	Minimum value supported.
`PWR_MD_max` . Get . Same type as attribute	Maximum value supported.
`PWR_MD_precision` . Get . `uint64_t`	Number of significant digits in values.
`PWR_MD_precision` . Get . `double`	Estimated percent error +/- of measured vs. actual values.

Continued on next page

Table 5.2 – continued from previous page

Attribute, Get/Set, Type	Description
PWR_MD_update_rate . Get/Set . double	Rate values become visible to user, in updates per second. Getting or setting a value at a rate higher than this is not useful.
PWR_MD_sample_rate . Get/Set . double	Rate of underlying sampling, in samples per second. This is only relevant for values derived over time (e.g., PWR_ATTR_ENERGY).
PWR_MD_time_window . Get/Set . PWR_Time	The time window used to calculate the value returned or relevant to an attribute. For example, the "instantaneous" PWR_ATTR_POWER values reported may actually be averaged over a short time window. Power caps are also enforced with respect to a target time window.
PWR_MD_ts_latency . Get . PWR_Time	Estimate of the time required to get or set an attribute. This is useful to estimate completion time for an operation *a priori*. A value of zero should be returned when the get/set is instantaneous.
PWR_MD_ts_accuracy . Get . PWR_Time	Estimated accuracy of returned timestamps, represented as +/- the PWR_Time value returned.
PWR_MD_max_len . Get . uint64_t	The maximum string length that will be returned by the metadata interface. All other string lengths (metadata items ending in _LEN) will be less than or equal to this value. The value of PWR_MD_MAX_LEN will be less than or equal to PWR_MAX_STRING_LEN.

Continued on next page

Table 5.2 – continued from previous page

Attribute, Get/Set, Type	Description
PWR_MD_name_len . Get . uint64_t	Length of the attribute name string, in bytes. This is the buffer length needed to store the string returned when PWR_MD_NAME is requested.
PWR_MD_name . Get . uint64_t	Attribute name string. This is a C-style NULL-terminated ASCII string. This provides a human readable name for the attribute. The string length is given by PWR_MD_NAME_LEN.
PWR_MD_desc_len . Get . uint64_t	Length of the attribute description string, in bytes. This is the buffer length needed to store the string returned when PWR_MD_DESC is requested.
PWR_MD_desc . Get . char *	Attribute description string. This is a C-style NULL-terminated ASCII string. This provides a human readable description of the attribute that is more descriptive than the attribute's name alone. The string length is given by PWR_MD_DESC_LEN.
PWR_MD_value_len . Get . uint64_t	Maximum length of the value strings returned by PWR_MetaValueAtIndex. This can be used to discover the buffer size that needs to be passed to PWR_MetaValueAtIndex via the value_str argument.

Continued on next page

Table 5.2 – continued from previous page

Attribute, Get/Set, Type	Description
`PWR_MD_vendor_info_len` . Get . `uint64_t`	Length of the vendor information string, in bytes. This is the buffer length needed to store the string returned when `PWR_MD_VENDOR_INFO` is requested.
`PWR_MD_vendor_info` . Get . `char *`	Vendor provided information string. This is a C-style NULL-terminated ASCII string. This may be used to convey part numbers, configuration, or other non-standard information. The string length is given by `PWR_MD_VENDOR_INFO_LEN`.
`PWR_MD_measure_method` . Get . `char *`	Denotes the measurement method: an actual measurement (returned value = 0) or a model based estimate (return value = 1). Other values > 1 may be used to denote multiple vendor specific models in the situation where multiple models may exist.

Chapter 6

High-Level (Common) Functions

This chapter includes specifications for High-Level functions that are common for more than one of the Role/System pair interfaces specified in chapter 7. The implementation may choose to selectively provide implementations for these functions, but all should be stubbed out or available. If an implementation is not provided the function should simply return `PWR_RET_NOT_IMPLEMENTED`.

6.1 Report Functions

Report functions are intended to provide a number of Role/System pairs with the ability to produce a range of reports. These particular functions target historic data, typically data that has been recorded in logs or some type of database. These functions are considered High-Level and abstract the object and group concepts found in the Core functions. Information is requested based on higher level concepts such as job, application or user ID. These functions require the user to provide a context which is used for determining whether the calling user can access the requested data.

Function Prototype for `PWR_GetReportByID()`

The `PWR_GetReportByID` function is provided to allow the collection of statistics information based on the ID types defined in `PWR_ID` in Section 4.9. A `PWR_ID` type must be supplied with `char*` pointer pointing to a valid ID for the specified type. The PWR_AttrName, PWR_AttrStat pair determines the statistic that will be reported. For example, the user of this function might desire the maximum power used over a period of time one week prior to the current time. The user would specify the id, id_type, PWR_ATTR_POWER for the attribute and PWR_STAT_MAX for the statistic and populate the start and stop members of the PWR_TimePeriod structure appropriately. The times specified must be prior to the time when the function is called. The function returns the actual start and stop times if they differ from the times the user inputs. The implementation should return the time available time period that most closely matches the requested time period. The implementation determines the supported attribute combinations. The context of the calling user will determine if the user has the necessary privilege to access this information. This functionality assumes the system has a data retention capability exposed to the user.

```
int PWR_GetReportByID(PWR_Cntxt context, const char* id, PWR_ID
id_type, PWR_AttrName name, PWR_AttrStat stat, double* value,
PWR_TimePeriod* ReportTimes)
```

Arguments		Description
IN	PWR_Cntxt context	The calling user's context which can be used to determine data access for individual role/user combinations.
IN	const char* id	The ID that the statistic will be collected for.
IN	PWR_ID id_type	The type of ID used to interpret the ID input.
IN	PWR_AttrName name	The name of the attribute the statistic will be based on.
IN	PWR_AttrStat stat	The desired statistic.
OUT	double* value	Pointer to a double that will contain the statistic.
IN/OUT	PWR_TimePeriod* ReportTimes	The user specified window for the report (start and stop times must be specified).

Return Code(s)	Description
PWR_RET_SUCCESS	Upon SUCCESS
PWR_RET_FAILURE	Upon FAILURE (Function is implemented but call failed)
PWR_RET_NOT_IMPLEMENTED	Indicates that the combination of the attribute statistic pair and ID is not supported by this implementation.

Chapter 7

Role/System Interfaces

This chapter includes the specifications for all of the Role/System pair interfaces depicted in figure 2.1 on page 11. Each interface section first outlines the purpose the interface serves. Core functionality for each interface is exposed through the attribute functions (see section 5.4). Each interface section includes a table of the supported attributes for that interface. The table contains the *suggested* attributes that the implementation should support for each interface. The implementation can choose to implement additional, some subset, or none of the attributes listed for that interface. As previously mentioned, the implementation must implement all attribute functions whether individual attributes are supported or not. If a particular attribute is not supported for that interface the implementation should return `PWR_RET_NOT_IMPLEMENTED`.

In addition to the attribute functions, other Core (Common) functions are included in this specification. Each individual interface section will enumerate the Core (Common) functions that the specification suggests are applicable for that interface (see chapter 5 for details regarding Core (Common) functions). Again, the implementation must implement these functions but may choose not to support them for a particular interface.

Each section also includes the High-Level (Common) functions that are applicable to that section (see chapter 6 for details regarding High-Level (Common) functions). These functions are functions that are applicable to more than one Role/System pair interface.

Finally, individual interface sections may also contain interface specific functions. These are functions that, at the time of their addition to the specification, are specific to one Role/System pair. This does not indicate that the function cannot be supported by an implementation for other Role/System pairs, only that the authors did not recognize a use for other interfaces at the time of addition to the specification.

7.1 Operating System, Hardware Interface

The Operating system/Hardware Interface is intended to be a low level interface that exposes power and energy relevant architecture features of the underlying hardware, such as the ability to measure and control power and energy characteristics of underlying components. In some cases this information will be abstracted for presentation to the application through the Application/Operating System API interface (section 7.3) or the resource manager through the Resource Manager/Operating System API (section 7.5). While we have chosen the term `Operating system` as part of this interface name, we are not strictly implying that all interfaces described in this section should be limited to the domain of the operating system. Additionally, we are not implying that this interface requires specific privileges, although many low level operations require elevated privileges. Portions of the system software stack, like a runtime system, may use many of the interfaces described in this section.

7.1.1 Supported Attributes

A significant amount of functionality for this interface is exposed through the attribute functions (section 5.4). The attribute functions in conjunction with the following attributes (Table 7.1) expose numerous measurement (get) and control (set) capabilities to the operating system.

Table 7.1: Operating System, Hardware - Supported Attributes

Attribute, Get/Set, Type	**Description**
`PWR_ATTR_PstateDesc` . Get/Set . `uint64_t`	The current P-state for the object specified (typically processors but for use with other component types when applicable).

Continued on next page

Table 7.1 – continued from previous page

Attribute, Get/Set, Type	Description
`PWR_ATTR_CstateDesc` . Get/Set . `uint64_t`	The current C-state for the object specified (typically processors but for use with other component types when applicable).
`PWR_ATTR_CstateLimitDesc` . Get/Set . `uint64_t`	The lowest C-state allowed for the object specified (typically processors but for use with other component types when applicable).
`PWR_ATTR_SstateDesc` . Get/Set . `uint64_t`	The current S-state for the object specified (typically processors but for use with other component types when applicable).
`PWR_ATTR_CurrentDesc` . Get . `double`	Discrete current value in amps. The current value should be the value measured as close as possible to the time of the function call.
`PWR_ATTR_VoltageDesc` . Get . `double`	Discrete voltage value in volts. The voltage value should be the value measured as close as possible to the time of the function call.
`PWR_ATTR_PowerDesc` . Get . `double`	Discrete power value in watts. The power value should be the value measured as close as possible to the time of the function call.
`PWR_ATTR_MinPowerDesc` . Get/Set . `double`	Minimum power limit (floor, lower bound) for the specified object in watts.
`PWR_ATTR_MaxPowerDesc` . Get/Set . `double`	Maximum power limit (ceiling, upper bound) for the specified object (as in power cap) in watts.
`PWR_ATTR_FreqDesc` . Get/Set . `double`	The current operating frequency value for the specified object in Hz (cycles per second).

Continued on next page

Table 7.1 – continued from previous page

Attribute, Get/Set, Type	Description
`PWR_ATTR_FreqLimitMinDesc` . Get/Set . `double`	Minimum operating frequency limit for the specified object in Hz (cycles per second).
`PWR_ATTR_FreqLimitMaxDesc` . Get/Set . `double`	Maximum operating frequency limit for the specified object in Hz (cycles per second).
`PWR_ATTR_EnergyDesc` . Get . `double`	The cumulative energy used by the specified object in joules. Note that two attribute get calls are typically required to obtain the energy consumed by the specified object. Subtracting the energy value obtained from the first call from the energy value obtained from the second call produces the energy used for the object from the timestamp of the first value through the timestamp of the second value.
`PWR_ATTR_TempDesc` . Get . `double`	The current temperature value for the specified object in degrees Celsius.
`PWR_ATTR_ThrottledIdDesc` . Get . `double`	The cumulative time in nanoseconds that the specified object's performance was purposefully slowed in order to meet some constraint, such as a power cap.
`PWR_ATTR_ThrottledCountIdDesc` . Get . `double`	The cumulative count of the number of times that the specified object's performance was purposefully slowed in order to meet some constraint, such as a power cap.

Continued on next page

Table 7.1 – continued from previous page

Attribute, Get/Set, Type	Description
PWR_ATTR_OSIdDesc . Get . double	The operating system ID that corresponds to the object. For example, a runtime system may need to figure out which Power API PWR_OBJ_CORE objects correspond to the cores that it is controlling. This attribute provides a linkage between Power API objects and operating system IDs.
PWR_ATTR_GovDesc . Get . double	Power related governor capability exposed through the operating system interface.

7.1.2 Supported Core (Common) Functions

- Hierarchy Navigation Functions - section 5.2
  - ALL
- Group Functions - section 5.3
  - ALL
- Attribute Functions - section 5.4
  - ALL
- Metadata Functions - section 5.5
  - ALL
- Statistics Functions - section 5.6
  - ALL - for real time queries only

7.1.3 Supported High-Level (Common) Functions

7.1.4 Interface Specific Functions

Function Prototype for `PWR_StateTransitDelay()`

The `PWR_StateTransitDelay` function returns the expected latency to transition between two valid states in nanoseconds. It is up to the vendor to provide accurate estimates for hardware. For example, P-state transitions could be given a single latency, even though some transitions might take less time (e.g., high voltage to lower voltage versus low to high). The desired state must be expressed using a `PWR_OperState` structure described in section 4.10 on page 29. This transition time may be a worst case latency time, and may be supplied by the hardware manufacturer (through the BIOS or other reporting mechanism). It is expected that this delay is an estimate of the time required to transition between states, not an estimate of the time that the core is unavailable for use (which may be a shorter interval than the time for the changes to take effect).

```
int PWR_StateTransitDelay(PWR_Ob obj, PWR_OperState start_state,
PWR_OperState end_state, PWR_Time *latency)
```

Arguments		**Description**
IN	`PWR_Ob obj`	The object that the state transition would be applied to.
IN	`PWR_OperState start_state`	The state at the beginning of the transition.
IN	`PWR_OperState end_state`	The state at the end of the transition.
OUT	`PWR_Time *latency`	Pointer to a double that will contain the transition latency in nanoseconds upon return.

Return Code(s)	**Description**
`PWR_RET_SUCCESS`	Upon SUCCESS.
`PWR_RET_FAILURE`	Upon FAILURE.

7.2 Monitor and Control, Hardware Interface

The Monitor and Control/Hardware interface is targeted to support a critical function on HPC platforms (systems monitoring and management) often embodied in Reliability Availability and Serviceability (RAS) systems. RAS systems must evolve to measure and control power and energy relevant aspects of the system and serve this information and capability to administrators (Administrator/Monitor and Control Interface - section 7.7), resource managers (Resource Manager/Monitor and Control Interface - section 7.6), accounting (Accounting/Monitor and Control Interface - section 7.9) and users (User/Monitor and Control Interface - section 7.10). The Monitor and Control Interface serves more other roles than any other system in this specification. The base level functionality that is exposed through this interface is very similar to the Operating System/Hardware Interface (section 7.1) but the functional responsibilities of the role differ considerably. Some of the interfaces described in this specification imply data retention, or database, functionality. The monitor and control software (RAS system) is a prime candidate to serve this purpose. Low level power and energy data can be mined through the interfaces documented in this section and stored in raw or processed form in a database and made available for historic queries by other roles.

7.2.1 Supported Attributes

As in the Operating System/Hardware interface (section 7.1) a significant amount of functionality for this interface is exposed through the attribute functions (section 5.4). The attribute functions in conjunction with the following attributes (Table 7.2) expose numerous measurement (get) and control (set) capabilities to the monitor and control system.

Table 7.2: Monitor and Control, Hardware - Supported Attributes

Attribute, Get/Set, Type	**Description**
`PWR_ATTR_PstateDesc` . Get/Set . `uint64_t`	The current P-state for the object specified (typically processors but for use with other component types when applicable).

Continued on next page

Table 7.2 – continued from previous page

Attribute, Get/Set, Type	Description
PWR_ATTR_CstateDesc . Get/Set . uint64_t	The current C-state for the object specified (typically processors but for use with other component types when applicable).
PWR_ATTR_CstateLimitDesc . Get/Set . uint64_t	The lowest C-state allowed for the object specified (typically processors but for use with other component types when applicable).
PWR_ATTR_SstateDesc . Get/Set . uint64_t	The current S-state for the object specified (typically processors but for use with other component types when applicable).
PWR_ATTR_CurrentDesc . Get . double	Discrete current value in amps. The current value should be the value measured as close as possible to the time of the function call.
PWR_ATTR_VoltageDesc . Get . double	Discrete voltage value in volts. The voltage value should be the value measured as close as possible to the time of the function call.
PWR_ATTR_PowerDesc . Get . double	Discrete power value in watts. The power value should be the value measured as close as possible to the time of the function call.
PWR_ATTR_MinPowerDesc . Get/Set . double	Minimum power limit (floor, lower bound) for the specified object in watts.
PWR_ATTR_MaxPowerDesc . Get/Set . double	Maximum power limit (ceiling, upper bound) for the specified object (as in power cap) in watts.
PWR_ATTR_FreqDesc . Get/Set . double	The current operating frequency value for the specified object in Hz (cycles per second).

Continued on next page

Table 7.2 – continued from previous page

Attribute, Get/Set, Type	Description
`PWR_ATTR_FreqLimitMinDesc` . Get/Set . `double`	Minimum operating frequency limit for the specified object in Hz (cycles per second).
`PWR_ATTR_FreqLimitMaxDesc` . Get/Set . `double`	Maximum operating frequency limit for the specified object in Hz (cycles per second).
`PWR_ATTR_EnergyDesc` . Get . `double`	The cumulative energy used by the specified object in joules. Note that two attribute get calls are typically required to obtain the energy consumed by the specified object. Subtracting the energy value obtained from the first call from the energy value obtained from the second call produces the energy used for the object from the timestamp of the first value through the timestamp of the second value.
`PWR_ATTR_TempDesc` . Get . `double`	The current temperature value for the specified object in degrees Celsius.

7.2.2 Supported Core (Common) Functions

- Hierarchy Navigation Functions - section 5.2
  - ALL
- Group Functions - section 5.3
  - ALL
- Attribute Functions - section 5.4
  - ALL
- Metadata Functions - section 5.5
  - ALL
- Statistics Functions - section 5.6
  - ALL

7.2.3 Supported High-Level (Common) Functions

7.2.4 Interface Specific Functions

7.3 Application, Operating System Interface

The Application/Operating System Interface is intended to expose the appropriate level of information (measurement) and control to the application user or application library. This interface may also provide functionality appropriate for other levels of system software, such as a runtime system. The capabilities included in this interface concentrate on providing abstractions that allow an application or library to provide information that can be used to make intelligent decisions regarding performance, power and energy efficiency.

An important aspect of this interface is accommodating portable application (or library) code. Generalized concepts such as performance and sleep states that hardware can operate in are used rather than architecture specific concepts such as hardware P-States. The operating system, or privileged layer, is responsible for appropriately translating the abstracted information provided by the application layer into the hardware specific details necessary for accomplishing the desired functionality (or not). In essence the operating system, or privileged layer, acts as the hardware translator for the application.

7.3.1 Supported Attributes

A significant amount of functionality for this interface is exposed through the attribute functions (section 5.4). The attributes functions in conjunction with the following attributes (Table 7.3) expose numerous measurement and control capabilities to the application, application libraries or possibly portions of runtime systems.

Table 7.3: Application, Operating System - Supported Attributes

Attribute, Get/Set, Type	**Description**
`PWR_ATTR_PowerDesc` . Get . `double`	Discrete power value in watts. The power value should be the value measured as close as possible to the time of the function call.
`PWR_ATTR_MinPowerDesc` . Get/Set . `double`	Minimum power limit (floor, lower bound) for the specified object in watts.

Continued on next page

Table 7.3 – continued from previous page

Attribute, Get/Set, Type	Description
PWR_ATTR_MaxPowerDesc . Get/Set . double	Maximum power limit (ceiling, upper bound) for the specified object (as in power cap) in watts.
PWR_ATTR_FreqDesc . Get/Set . double	The current operating frequency value for the specified object in Hz (cycles per second).
PWR_ATTR_FreqLimitMinDesc . Get/Set . double	Minimum operating frequency limit for the specified object in Hz (cycles per second).
PWR_ATTR_FreqLimitMaxDesc . Get/Set . double	Maximum operating frequency limit for the specified object in Hz (cycles per second).
PWR_ATTR_EnergyDesc . Get . double	The cumulative energy used by the specified object in joules. Note that two attribute get calls are typically required to obtain the energy consumed by the specified object. Subtracting the energy value obtained from the first call from the energy value obtained from the second call produces the energy used for the object from the timestamp of the first value through the timestamp of the second value.
PWR_ATTR_TempDesc . Get . double	The current temperature value for the specified object in degrees Celsius.

Continued on next page

Table 7.3 – continued from previous page

Attribute, Get/Set, Type	Description
PWR_ATTR_OSIdDesc . Get . double	The operating system ID that corresponds to the object. For example, a runtime system may need to figure out which Power API PWR_OBJ_CORE objects correspond to the cores that it is controlling. This attribute provides a linkage between Power API objects and operating system IDs.
PWR_ATTR_GovDesc . Get . double	Power related governor capability exposed through the operating system interface.

7.3.2 Supported Core (Common) Functions

- Hierarchy Navigation Functions - section 5.2
  - ALL
- Group Functions - section 5.3
  - ALL
- Attribute Functions - section 5.4
  - ALL
- Metadata Functions - section 5.5
  - ALL
- Statistics Functions - section 5.6
  - ALL - for real time queries only

7.3.3 Supported High-Level (Common) Functions

Function Prototype for PWR_AppHintCreate()

The PWR_AppHint* functions are intended to be used by an application, or application library, to supply power relevant hints to the operating system (or a runtime layer). This function creates a tuning hint region-context that can be re-used, and indicates to the OS/runtime that information gathered from previous executions of this particular region can be used to determine effective strategies to improve power/performance efficiency on future runs. The PWR_RegionHint hints are intended to be used by the application layer to indicate that it is entering a `SERIAL, PARALLEL, COMPUTE` (computation intensive) or `COMMUNICATE, I/O or MEM_BOUND` (communication intensive, I/O intensive or memory bound) region. The `DEFAULT` hint types are used for defining regions that may be significant, but the type of region is unknown. The `GLOBAL_LOOP` hint type helps to denote the main computational loop for an application, which allows some runtimes to optimize machine power/performance balance. `PWR_RegionHint` type is described in section 4.11 on page 30. It is intended that these hints may be leveraged to provide some performance or power benefit, for example, a hint may indicate that an intensely parallel region is about to happen, this may motivate the proactive migration of tasks to an accelerator or preemptively speed up cooling fans to proactively deal with the thermal load. `PWR_RegionIntensity`, described in section 4.11 on page 31 can be used for finer-grained hints than are possible with `PWR_RegionHint`. It is intended to allow for more explicit hints as to the intensity of the described region behavior. For example, it can be used to describe the intensity of a memory bound region, which can be utilized by the runtime or operating system in deciding what resources to allocate for a given power budget. `PWR_RegionIntensity` values are useful for all regions except for `GLOBAL_LOOP` regions. It is expected that the implementation will use these hints whenever possible to increase application performance while honoring energy/power targets or increase energy efficiency without incurring significant performance penalties. `PWR_RegionIntensity` may be set to `PWR_REGION_INT_NONE` if it is desirable for the operating system or a runtime to determine the intensity of resource usage dependent on the given hint. `PWR_REGION_INT_NONE` can also be used when the intensity of the described behavior is not known. This parameter may be ignored by the OS. The `hint_region_name` is used to name a region and assign a ID number to that region. All `hint_region_name` values used **must be unique**. If a name is not specified (name input parameter is NULL), then the implementation will assign a unique name to the region. This will create and return a region ID that can be used in calls to `PWR_AppHintStart` to indicate the region that is being entered. This should be accompanied by a `PWR_RegionHint` type as described in section 4.11 on page 30 and a `PWR_RegionIntensity` as described in section 4.11 on page 31. All calls to `PWR_AppHintCreate` should be matched to a call to `PWR_AppHintDestroy` (7.3.3). *Rationale: Giving hint regions human readable names facilitates easier display and debugging of information associated with the region, allowing for performance reports to be generated from the OS/runtime for regions of interest. /End Rationale.*

```
int PWR_AppHintCreate(PWR_Obj obj, const char *name, uint64_t
*region_id, PWR_RegionHint hint)
```

Arguments		Description
IN	PWR_Obj obj	The object that the hint applies to.
IN	const char *name	A name for the region of code to receive the hint.
IN/OUT	uint64_t *region_id	A region identifier created from the region name that can be used in subsequent hint calls.
IN	PWR_RegionHint hint	The hint corresponding to the code (behavioral) region being entered.

Return Code(s)	Description
PWR_ERR_SUCCESS	Upon SUCCESS
PWR_RET_FAILURE	Upon FAILURE
PWR_RET_NOT_IMPLEMENTED	Object does not support the requested operation

Function Prototype for PWR_AppHintDestroy()

This function destroys a tuning hint region that was created with the PWR_AppHintCreate call. For more information on the use of application tuning hints in regions, see 7.3.3. All calls to PWR_AppHintCreate should be matched to a call to PWR_AppHintDestroy (7.3.3).

```
int PWR_AppHintDestroy(uint64_t region_id)
```

Arguments		Description
IN	uint64_t region_id	The region identifier of the region to be destroyed.

Return Code(s)	Description
PWR_ERR_SUCCESS	Upon SUCCESS
PWR_RET_FAILURE	Upon FAILURE
PWR_RET_NOT_IMPLEMENTED	Object does not support the requested operation

Function Prototype for `PWR_AppHintStart()`

The `PWR_AppHint*` functions are intended to be used by an application, or application library, to supply power relevant hints to the operating system (or a runtime layer). It is intentional that many of these hints do not directly imply that a power or energy adjustment will be made. `hint_region_id` values are used to indicate the region ID number supplied from the `PWR_AppHintCreate` function. A given region can only be started one, and requires a matched call to `PWR_AppHintStop()`. Subsequent calls to `PWR_AppHintStart` for a given region ID that has already been started without being stopped are ignored. Tuning hints for multiple regions may be nested, but the OS/runtime is not required to support more than a single region at a time. Therefore nested hints calls result in using the most recently started region and hint. When nested regions are stopped, the parent region's hint is re-applied. Consult your implementation documentation to determine if blending of nested hints are supported (multiple hint regions being applied simultaneously).

```
int PWR_AppHintStart(uint64_t hint_region_id)
```

Arguments	Description
IN `uint64_t hint_region_id`	A region identifier of the region being entered.

Return Code(s)	Description
PWR_ERR_SUCCESS	Upon SUCCESS
PWR_RET_FAILURE	Upon FAILURE
PWR_RET_NOT_IMPLEMENTED	Object does not support the requested operation

Function Prototype for `PWR_AppHintStop()`

The `PWR_AppHint*` functions is intended to be used by an application, or application library, to supply power relevant hints to the operating system (or a runtime layer). This function delineates the termination of a tuning hint region that was started with the `PWR_AppHintStart` call.

```
int PWR_AppHintStop(uint64_t region_id)
```

Arguments		Description
IN	uint64_t region_id	The region identifier of the region that is to be stopped.

Return Code(s)	Description
PWR_ERR_SUCCESS	Upon SUCCESS
PWR_RET_FAILURE	Upon FAILURE
PWR_RET_NOT_IMPLEMENTED	Object does not support the requested operation

Function Prototype for PWR_AppHintProgress()

The PWR_AppHintProgress function is intended to be used by an application, or application library, to indicate progress within a hint region. This can be used by underlying OS/runtimes to determine if adjustments made to the system based on the hint information are appropriate and facilitate further tuning. While use of this function is not required in order to use hints for code regions its use in encouraged as it may provide increased efficiency/performance from the OS/runtime. This function call may be ignored by the OS or runtime if they do not support hint region tuning.

```
int PWR_AppHintProgress(uint64_t region_id, double progress_
fraction)
```

Arguments		Description
IN	uint64_t region_id	A region identifier corresponding to the region making progress.
IN	double progress_fraction	A value representing what fraction of the region/computation is complete as of this call to PWR_AppHintProgress.

Return Code(s)	Description
PWR_ERR_SUCCESS	Upon SUCCESS
PWR_RET_FAILURE	Upon FAILURE
PWR_RET_NOT_IMPLEMENTED	Object does not support the requested operation

Function Prototype for `PWR_SetSleepStateLimit()`

`PWR_SetSleepStateLimit` allows the application to request that, when possible, the OS restrict the deepest sleep state (e.g. C-state) that the hardware can enter. It is important to note that this function does not place the object in a sleep state, it only suggests to the Operating System (or privileged layer) that it limit the deepest possible sleep state that the object can enter. The operating system or hardware are responsible for determining when hardware should be put to sleep. This is not required to be honored by the OS or HW, but serves as a hint to the OS as to the latency that can be tolerated when transitioning between sleep and active states. As the application cannot typically control the entry of hardware into sleep states this function is meant to provide a method for an application to express its latency tolerance in an environment where resources may be put into sleep states without the application's knowledge. Applications calling `PWR_SetSleepStateLimit` are expected to make use of the `PWR_WakeUpLatency` call on page 111 to provide information needed to determine the desired sleep state level. Sleep states must conform to the `PWR_SleepState` type in section 4.11 on page 31.

```
int PWR_SetSleepStateLimit(PWR_Obj obj, PWR_SleepState state)
```

Arguments	Description
IN `PWR_Obj obj`	The object to set the sleep state on.
IN `PWR_SleepState state`	The sleep state to set as the maximum deepest sleep allowed.

Return Code(s)	Description
`PWR_RET_SUCCESS`	Upon SUCCESS
`PWR_RET_FAILURE`	Upon FAILURE
`PWR_RET_NOT_IMPLEMENTED`	Object does not support the requested operation

Function Prototype for `PWR_WakeUpLatency()`

The `PWR_WakeUpLatency` function returns a value in nanoseconds that corresponds to the time required to resume normal operation when transitioning out of a given sleep state. If the supplied `PWR_Obj` does not support sleeping or the requested sleep state is not available then the function may return PWR_RET_FAILURE. *Advice to users: This function is useful when determining what sleep states can be exploited when knowledge of the length of time that certain operations (most likely remote ones) can be expected to take. Use of this function is intended to be paired with the* ***SetSleepStateLimit*** *function. Although users cannot use this function to place hardware into a sleep state, when used in conjunction with* ***SetSleepStateLimit*** *it can be used to suggest to an actor placing the hardware in a sleep state which state may be the most desirable. End of Advice to users.*

```
int PWR_WakeUpLatency(PWR_Obj obj, PWR_SleepState state, PWR_
Time* latency)
```

Arguments		Description
IN	PWR_Obj obj	The object to query for latency.
IN	PWR_SleepState state	The sleep state to transition out of.
OUT	PWR_Time* latency	The latency of the transition in nanoseconds.

Return Code(s)	Description
PWR_RET_SUCCESS	Upon SUCCESS
PWR_RET_FAILURE	Upon FAILURE
PWR_RET_NOT_IMPLEMENTED	Object does not support the requested operation

Function Prototype for PWR_RecommendSleepState()

This is a convenience function for cases in which an application's maximum tolerable latency is known for a given region and a deepest possible sleep state for use with the SetSleepStateLimit function is desired. Calling RecommendSleepState with the known latency will return the sleep state that has the closest latency to the desired value without exceeding it. Returned sleep states from this function conform to the PWR_SleepState type in section 4.11 on page 31.

```
 PWR_RecommendSleepState(PWR_Obj obj, PWR_Time latency, PWR_
SleepState* state)
```

Arguments		Description
IN	PWR_Obj obj	The object to set the sleep state on.
IN	PWR_Time latency	The amount of latency tolerable to the application in nanoseconds.
OUT	PWR_SleepState* state	The deepest sleep state recommended to be used as a limit.

Return Code(s)	Description
PWR_RET_SUCCESS	Upon SUCCESS
PWR_RET_FAILURE	Upon FAILURE
PWR_RET_NOT_IMPLEMENTED	Object does not support the requested operation

Function Prototype for `PWR_SetPerfState()`

The `PWR_SetPerfState` function is used to request that an object change its performance level. The operating system is responsible for translating the abstracted `PWR_PerfState` value into an appropriate hardware-specific performance level (e.g. a CPU P-State). Setting the performance state of an object is not guaranteed to result in the requested change. The operating system may choose to ignore it or the hardware may not honor the request. The user should not expect that once a performance state has been set that it will not change in the future. Multiple actors may also set the performance state, including in some cases, remote actors.

```
int PWR_SetPerfState(PWR_Obj obj, PWR_PerfState state)
```

Arguments		Description
IN	PWR_Obj obj	The object to set the performance state on.
IN	PWR_PerfState state	The performance state to set the object to.

Return Code(s)	Description
PWR_RET_SUCCESS	Upon SUCCESS
PWR_RET_FAILURE	Upon FAILURE
PWR_RET_NOT_IMPLEMENTED	Object does not support the requested operation

Function Prototype for `PWR_GetPerfState()`

The `PWR_GetPerfState` function returns the performance state for any given object. The value that is returned is an abstracted value based on the real hardware state of the object that is mapped to the closest `PWR_PerfState` value. Objects must return `PWR_RET_FAILURE` if they do not support operating in different states.

```
int PWR_GetPerfState(PWR_Obj obj, PWR_PerfState* state)
```

Arguments		Description
IN	`PWR_Obj obj`	The object to get the current performance state of.
OUT	`PWR_PerfState* state`	performance state of the object.

Return Code(s)	Description
`PWR_RET_SUCCESS`	Upon SUCCESS
`PWR_RET_FAILURE`	Upon FAILURE
`PWR_RET_NOT_IMPLEMENTED`	Object does not support the requested operation

Function Prototype for `PWR_GetSleepState()`

The `PWR_GetSleepState` function returns the current sleep state for any given object.

```
int PWR_GetSleepState(PWR_Obj obj, PWR_PerfState* state)
```

Arguments		Description
IN	`PWR_Obj obj`	The object to get the current sleep state of.
OUT	`PWR_PerfState* state`	The sleep state of the object.

Return Code(s)	Description
`PWR_RET_SUCCESS`	SUCCESS
`PWR_RET_FAILURE`	FAILURE
`PWR_RET_NOT_IMPLEMENTED`	Object does not support the requested operation.

7.4 User, Resource Manager Interface

The User/Resource Manger Interface is intended to support access to power and energy related information, specifically pertaining to jobs, relevant to an HPC user. This interface is similar to the User/Monitor and Control Interface (section 7.10) but in this case assumes that the Resource Manager has a data retention capability (database) available to query energy and statistics information based on job or user Id. The availability of this information is implementation dependent. Alternatively, if the Resource Manager does not have a database capability, the same interfaces are available to the user role through the User/Monitor and Control System Interface (section 7.10 which may provide this functionality.

7.4.1 Supported Attributes

The Power API specification does not currently recommend that any of the attributes be exposed to the user role. The implementation is free to expose any attribute they determine is useful to the user role without violating the specification.

7.4.2 Supported Core (Common) Functions

- Hierarchy Navigation Functions - section 5.2
  - ALL
- Group Functions - section 5.3
  - ALL
- Metadata Functions - section 5.5
  - ALL
- Statistics Functions - section 5.6
  - ALL - for historic queries only

7.4.3 Supported High-Level (Common) Functions

- Report Functions - section 6.1
  - ALL

7.4.4 Interface Specific Functions

7.5 Resource Manager, Operating System Interface

The Resource Manager/Operating System Interface is intended to access both low level and abstracted information from the operating system. Similar or additional information may be available from the monitor and control system (section 7.6) depending on the implementation. The resource manager is in a somewhat unique position of providing a range of functionality depending on the specific implementation. The resource manager role includes functionality such as batch schedulers and allocators as well as potential portions of tightly integrated runtime and launch systems. The resource manager may require fairly low level measurement information to make decisions and potentially store historic information for consumption by the user role (for example). The resource manager may also play a very large role in controlling power and energy pertinent functionally on both a application and platform basis in response to facility restrictions (power capping or energy aware scheduling for example).

7.5.1 Supported Attributes

A significant amount of functionality for this interface is exposed through the attribute functions (section 5.4). The attribute functions in conjunction with the following attributes (Table 7.4) expose numerous measurement (get) and control (set) capabilities to the resource manager.

Table 7.4: Resource Manager, Operating System - Supported Attributes

Attribute, Get/Set, Type	**Description**
`PWR_ATTR_PstateDesc` . Get/Set . `uint64_t`	The current P-state for the object specified (typically processors but for use with other component types when applicable).
`PWR_ATTR_CstateDesc` . Get/Set . `uint64_t`	The current C-state for the object specified (typically processors but for use with other component types when applicable).

Continued on next page

Table 7.4 – continued from previous page

Attribute, Get/Set, Type	Description
PWR_ATTR_CstateLimitDesc . Get/Set . uint64_t	The lowest C-state allowed for the object specified (typically processors but for use with other component types when applicable).
PWR_ATTR_SstateDesc . Get/Set . uint64_t	The current S-state for the object specified (typically processors but for use with other component types when applicable).
PWR_ATTR_PowerDesc . Get . double	Discrete power value in watts. The power value should be the value measured as close as possible to the time of the function call.
PWR_ATTR_MinPowerDesc . Get/Set . double	Minimum power limit (floor, lower bound) for the specified object in watts.
PWR_ATTR_MaxPowerDesc . Get/Set . double	Maximum power limit (ceiling, upper bound) for the specified object (as in power cap) in watts.
PWR_ATTR_FreqDesc . Get/Set . double	The current operating frequency value for the specified object in Hz (cycles per second).
PWR_ATTR_FreqLimitMinDesc . Get/Set . double	Minimum operating frequency limit for the specified object in Hz (cycles per second).
PWR_ATTR_FreqLimitMaxDesc . Get/Set . double	Maximum operating frequency limit for the specified object in Hz (cycles per second).

Continued on next page

Table 7.4 – continued from previous page

Attribute, Get/Set, Type	Description
PWR_ATTR_EnergyDesc . Get . double	The cumulative energy used by the specified object in joules. Note that two attribute get calls are typically required to obtain the energy consumed by the specified object. Subtracting the energy value obtained from the first call from the energy value obtained from the second call produces the energy used for the object from the timestamp of the first value through the timestamp of the second value.
PWR_ATTR_TempDesc . Get . double	The current temperature value for the specified object in degrees Celsius.

7.5.2 Supported Core (Common) Functions

- Hierarchy Navigation Functions - section 5.2
  - ALL
- Group Functions - section 5.3
  - ALL
- Attribute Functions - section 5.4
  - ALL
- Metadata Functions - section 5.5
  - ALL
- Statistics Functions - section 5.6
  - ALL

7.5.3 Supported High-Level (Common) Functions

7.5.4 Interface Specific Functions

7.6 Resource Manager, Monitor and Control Interface

The Resource Manager/Monitor and Control Interface is intended to access both low level and abstracted information from the monitor and control system (if available), much like the Resource Manager/Operating System Interface (section 7.5). The resource manager is in a somewhat unique position of providing a range of functionality depending on the specific implementation. The resource manager role includes functionality such as batch schedulers and allocators as well as potential portions of tightly integrated runtime and launch systems. The resource manager may require fairly low level measurement information to make decisions and potentially store historic information for consumption by the user role (for example). In contrast to the Resource Manager/Operating System Interface (section 7.5) this interface includes the capability to mine information from the Monitor and Control system in situations where the Resource Manager does not retain historic data itself. The resource manager may also play a very large role in controlling power and energy pertinent functionally on both a application and platform basis in response to facility restrictions (power capping or energy aware scheduling for example).

7.6.1 Supported Attributes

A significant amount of functionality for this interface is exposed through the attribute functions (section 5.4). The attribute functions in conjunction with the following attributes (Table 7.5) expose numerous measurement (get) and control (set) capabilities to the resource manager.

Table 7.5: Resource Manager, Monitor and Control - Supported Attributes

Attribute, Get/Set, Type	Description
`PWR_ATTR_PstateDesc` . Get/Set . `uint64_t`	The current P-state for the object specified (typically processors but for use with other component types when applicable).

Continued on next page

Table 7.5 – continued from previous page

Attribute, Get/Set, Type	Description
`PWR_ATTR_CstateDesc` . Get/Set . `uint64_t`	The current C-state for the object specified (typically processors but for use with other component types when applicable).
`PWR_ATTR_CstateLimitDesc` . Get/Set . `uint64_t`	The lowest C-state allowed for the object specified (typically processors but for use with other component types when applicable).
`PWR_ATTR_SstateDesc` . Get/Set . `uint64_t`	The current S-state for the object specified (typically processors but for use with other component types when applicable).
`PWR_ATTR_PowerDesc` . Get . `double`	Discrete power value in watts. The power value should be the value measured as close as possible to the time of the function call.
`PWR_ATTR_MinPowerDesc` . Get/Set . `double`	Minimum power limit (floor, lower bound) for the specified object in watts.
`PWR_ATTR_MaxPowerDesc` . Get/Set . `double`	Maximum power limit (ceiling, upper bound) for the specified object (as in power cap) in watts.
`PWR_ATTR_FreqDesc` . Get/Set . `double`	The current operating frequency value for the specified object in Hz (cycles per second).
`PWR_ATTR_FreqLimitMinDesc` . Get/Set . `double`	Minimum operating frequency limit for the specified object in Hz (cycles per second).
`PWR_ATTR_FreqLimitMaxDesc` . Get/Set . `double`	Maximum operating frequency limit for the specified object in Hz (cycles per second).

Continued on next page

Table 7.5 – continued from previous page

Attribute, Get/Set, Type	Description
`PWR_ATTR_EnergyDesc` . Get . `double`	The cumulative energy used by the specified object in joules. Note that two attribute get calls are typically required to obtain the energy consumed by the specified object. Subtracting the energy value obtained from the first call from the energy value obtained from the second call produces the energy used for the object from the timestamp of the first value through the timestamp of the second value.
`PWR_ATTR_TempDesc` . Get . `double`	The current temperature value for the specified object in degrees Celsius.

7.6.2 Supported Core (Common) Functions

- Hierarchy Navigation Functions - section 5.2
  - ALL
- Group Functions - section 5.3
  - ALL
- Attribute Functions - section 5.4
  - ALL
- Metadata Functions - section 5.5
  - ALL
- Statistics Functions - section 5.6
  - ALL

7.6.3 Supported High-Level (Common) Functions

- Report Functions - section 6.1
  - ALL

7.6.4 Interface Specific Functions

7.7 Administrator, Monitor and Control Interface

The Administrator/Monitor and Control Interface is intended to expose administrator level measurement and control capabilities to the administrator role for the HPC platform. This interface assumes that the administrator role has elevated privileges. Additionally, the administrator is assumed to have access to all user role functionality documented in sections 7.10 and 7.4. A full complement of access to low level information is exposed through the attribute interface and other core level functions.

7.7.1 Supported Attributes

A significant amount of functionality for this interface is exposed through the attribute functions (section 5.4). The attribute functions in conjunction with the following attributes (Table 7.6) expose numerous measurement (get) and control (set) capabilities to the administrator role.

Table 7.6: Monitor and Control, Hardware - Supported Attributes

Attribute, Get/Set, Type	Description
`PWR_ATTR_PstateDesc` . Get/Set . `uint64_t`	The current P-state for the object specified (typically processors but for use with other component types when applicable).
`PWR_ATTR_CstateDesc` . Get/Set . `uint64_t`	The current C-state for the object specified (typically processors but for use with other component types when applicable).
`PWR_ATTR_CstateLimitDesc` . Get/Set . `uint64_t`	The lowest C-state allowed for the object specified (typically processors but for use with other component types when applicable).
`PWR_ATTR_SstateDesc` . Get/Set . `uint64_t`	The current S-state for the object specified (typically processors but for use with other component types when applicable).

Continued on next page

Table 7.6 – continued from previous page

Attribute, Get/Set, Type	Description
`PWR_ATTR_CurrentDesc` . Get . `double`	Discrete current value in amps. The current value should be the value measured as close as possible to the time of the function call.
`PWR_ATTR_VoltageDesc` . Get . `double`	Discrete voltage value in volts. The voltage value should be the value measured as close as possible to the time of the function call.
`PWR_ATTR_PowerDesc` . Get . `double`	Discrete power value in watts. The power value should be the value measured as close as possible to the time of the function call.
`PWR_ATTR_MinPowerDesc` . Get/Set . `double`	Minimum power limit (floor, lower bound) for the specified object in watts.
`PWR_ATTR_MaxPowerDesc` . Get/Set . `double`	Maximum power limit (ceiling, upper bound) for the specified object (as in power cap) in watts.
`PWR_ATTR_FreqDesc` . Get/Set . `double`	The current operating frequency value for the specified object in Hz (cycles per second).
`PWR_ATTR_FreqLimitMinDesc` . Get/Set . `double`	Minimum operating frequency limit for the specified object in Hz (cycles per second).
`PWR_ATTR_FreqLimitMaxDesc` . Get/Set . `double`	Maximum operating frequency limit for the specified object in Hz (cycles per second).

Continued on next page

Table 7.6 – continued from previous page

Attribute, Get/Set, Type	Description
`PWR_ATTR_EnergyDesc` . Get . `double`	The cumulative energy used by the specified object in joules. Note that two attribute get calls are typically required to obtain the energy consumed by the specified object. Subtracting the energy value obtained from the first call from the energy value obtained from the second call produces the energy used for the object from the timestamp of the first value through the timestamp of the second value.
`PWR_ATTR_TempDesc` . Get . `double`	The current temperature value for the specified object in degrees Celsius.

7.7.2 Supported Core (Common) Functions

- Hierarchy Navigation Functions - section 5.2
  - ALL
- Group Functions - section 5.3
  - ALL
- Attribute Functions - section 5.4
  - ALL
- Metadata Functions - section 5.5
  - ALL
- Statistics Functions - section 5.6
  - ALL

7.7.3 Supported High-Level (Common) Functions

- Report Functions - section 6.1
  - ALL

7.7.4 Interface Specific Functions

7.8 HPCS Manager, Resource Manager Interface

The HPCS Manager/Resource Manager Interface is intended to provide the necessary functionality for the HPCS Manager to implement policy via the Resource Manager. Policy information such as power caps (minimums or maximums), per user energy limits and traditional policies like node hours and priorities will all play a role in energy aware platform scheduling.

7.8.1 Supported Attributes

The Power API specification does not currently recommend that any of the attributes be exposed to the HPCS Manager role. The implementation is free to expose any attribute they determine is useful to the user role without violating the specification.

7.8.2 Supported Core (Common) Functions

7.8.3 Supported High-Level (Common) Functions

7.8.4 Interface Specific Functions

7.9 Accounting, Monitor and Control Interface

The Accounting/Monitor and Control Interface is intended to support access to power and energy related information regarding users, jobs and platform details to the accounting role. The accounting role differs from the user role in part by the elevated permissions this role will typically have. The accounting role includes interfaces to expose both a low-level interface via the attribute interface and higher level energy and statistics information through interface specific functions. The availability of historic information, critical to much of the accounting role, is dependent on the availability of the information in the Monitor and Control System which is implementation specific.

7.9.1 Supported Attributes

A significant amount of functionality for this interface is exposed through the attribute functions (section 5.4). The attribute functions in conjunction with the following attributes (Table 7.7) expose numerous measurement (get) and control (set) capabilities to the accounting role.

Table 7.7: Accounting, Monitor and Control System - Supported Attributes

Attribute, Get/Set, Type	Description
`PWR_ATTR_PstateDesc` . Get/Set . `uint64_t`	The current P-state for the object specified (typically processors but for use with other component types when applicable).
`PWR_ATTR_CstateDesc` . Get/Set . `uint64_t`	The current C-state for the object specified (typically processors but for use with other component types when applicable).
`PWR_ATTR_CstateLimitDesc` . Get/Set . `uint64_t`	The lowest C-state allowed for the object specified (typically processors but for use with other component types when applicable).

Continued on next page

Table 7.7 – continued from previous page

Attribute, Get/Set, Type	Description
`PWR_ATTR_SstateDesc` . Get/Set . `uint64_t`	The current S-state for the object specified (typically processors but for use with other component types when applicable).
`PWR_ATTR_CurrentDesc` . Get . `double`	Discrete current value in amps. The current value should be the value measured as close as possible to the time of the function call.
`PWR_ATTR_VoltageDesc` . Get . `double`	Discrete voltage value in volts. The voltage value should be the value measured as close as possible to the time of the function call.
`PWR_ATTR_PowerDesc` . Get . `double`	Discrete power value in watts. The power value should be the value measured as close as possible to the time of the function call.
`PWR_ATTR_MinPowerDesc` . Get/Set . `double`	Minimum power limit (floor, lower bound) for the specified object in watts.
`PWR_ATTR_MaxPowerDesc` . Get/Set . `double`	Maximum power limit (ceiling, upper bound) for the specified object (as in power cap) in watts.
`PWR_ATTR_FreqDesc` . Get/Set . `double`	The current operating frequency value for the specified object in Hz (cycles per second).
`PWR_ATTR_FreqLimitMinDesc` . Get/Set . `double`	Minimum operating frequency limit for the specified object in Hz (cycles per second).
`PWR_ATTR_FreqLimitMaxDesc` . Get/Set . `double`	Maximum operating frequency limit for the specified object in Hz (cycles per second).

Continued on next page

Table 7.7 – continued from previous page

Attribute, Get/Set, Type	Description
`PWR_ATTR_EnergyDesc` . Get . `double`	The cumulative energy used by the specified object in joules. Note that two attribute get calls are typically required to obtain the energy consumed by the specified object. Subtracting the energy value obtained from the first call from the energy value obtained from the second call produces the energy used for the object from the timestamp of the first value through the timestamp of the second value.
`PWR_ATTR_TempDesc` . Get . `double`	The current temperature value for the specified object in degrees Celsius.

7.9.2 Supported Core (Common) Functions

- Hierarchy Navigation Functions - section 5.2
  - ALL
- Group Functions - section 5.3
  - ALL
- Attribute Functions - section 5.4
  - ALL
- Metadata Functions - section 5.5
  - ALL
- Statistics Functions - section 5.6
  - ALL

7.9.3 Supported High-Level (Common) Functions

- Report Functions - section 6.1
  - ALL

7.9.4 Interface Specific Functions

7.10 User, Monitor and Control Interface

The User/Monitor and Control Interface is intended to support access to power and energy information relevant to an HPC user. This interface is similar to the User/Resource Manager Interface (section 7.4) but exposes more low level information to the user through the Monitor and Control system, assuming the user has permission to access the information. The low level information exposed to the user role through this interface is primarily to support fine grained application analysis when available. The ability to mine energy and other statistics information based on job Id and user Id, included in this interface, assumes that a data retention capability is implemented in the Monitor and Control system. This is of course implementation dependent. Alternatively, if the Monitor and Control system does not have a database capability, the same interfaces are available to the user role through the User/Resource Manager Interface (section 7.4 which may provide this functionality.

7.10.1 Supported Attributes

A significant amount of functionality for this interface is exposed through the attribute functions (section 5.4). The attribute functions in conjunction with the following attributes (Table 7.8) expose numerous measurement (get) and control (set) capabilities to the user role.

Table 7.8: User, Monitor and Control - Supported Attributes

Attribute, Get/Set, Type	**Description**
`PWR_ATTR_PstateDesc` . Get/Set . `uint64_t`	The current P-state for the object specified (typically processors but for use with other component types when applicable).
`PWR_ATTR_CstateDesc` . Get/Set . `uint64_t`	The current C-state for the object specified (typically processors but for use with other component types when applicable).

Continued on next page

Table 7.8 – continued from previous page

Attribute, Get/Set, Type	Description
`PWR_ATTR_SstateDesc` . Get/Set . `uint64_t`	The current S-state for the object specified (typically processors but for use with other component types when applicable).
`PWR_ATTR_CurrentDesc` . Get . `double`	Discrete current value in amps. The current value should be the value measured as close as possible to the time of the function call.
`PWR_ATTR_VoltageDesc` . Get . `double`	Discrete voltage value in volts. The voltage value should be the value measured as close as possible to the time of the function call.
`PWR_ATTR_PowerDesc` . Get . `double`	Discrete power value in watts. The power value should be the value measured as close as possible to the time of the function call.
`PWR_ATTR_MinPowerDesc` . Get/Set . `double`	Minimum power limit (floor, lower bound) for the specified object in watts.
`PWR_ATTR_MaxPowerDesc` . Get/Set . `double`	Maximum power limit (ceiling, upper bound) for the specified object (as in power cap) in watts.
`PWR_ATTR_FreqDesc` . Get/Set . `double`	The current operating frequency value for the specified object in Hz (cycles per second).
`PWR_ATTR_FreqLimitMinDesc` . Get/Set . `double`	Minimum operating frequency limit for the specified object in Hz (cycles per second).
`PWR_ATTR_FreqLimitMaxDesc` . Get/Set . `double`	Maximum operating frequency limit for the specified object in Hz (cycles per second).

Continued on next page

Table 7.8 – continued from previous page

Attribute, Get/Set, Type	Description
PWR_ATTR_EnergyDesc . Get . double	The cumulative energy used by the specified object in joules. Note that two attribute get calls are typically required to obtain the energy consumed by the specified object. Subtracting the energy value obtained from the first call from the energy value obtained from the second call produces the energy used for the object from the timestamp of the first value through the timestamp of the second value.
PWR_ATTR_TempDesc . Get . double	The current temperature value for the specified object in degrees Celsius.

7.10.2 Supported Core (Common) Functions

- Hierarchy Navigation Functions - section 5.2
  - ALL
- Group Functions - section 5.3
  - ALL
- Attribute Functions - section 5.4
  - ALL
- Metadata Functions - section 5.5
  - ALL
- Statistics Functions - section 5.6
  - ALL

7.10.3 Supported High-Level (Common) Functions

- Report Functions - section 6.1
  - ALL

7.10.4 Interface Specific Functions

Chapter 8

Conclusion

The case for an HPC-community-adopted power API specification is compelling. The demand for computational cycles continues to increase, as does the expense to power the cycles. Hardware vendors are providing interfaces to power data and controls so that software can monitor usage and even control it. To maximize utilization of these ”knobs”, a portable interface layer allows multiple software products to code to a generic layer which can be translated by the individual hardware vendors. With this need in mind, the Power API defined herein sets out to address the following tenets.

Very wide scope from facility to hardware component This specification is not just limited to the hardware interfaces. The information from the hardware is the enabler for this API. However, the information is needed at many levels, from many different viewpoints. In [10] we identified a discrete set of unique actors (a.k.a. users, which can be software components) communicating via the API. In turn, these actors have interfaces with one or more systems within the scope of the API. The actor/system combinations represent the variety of viewpoints. For example, a batch job scheduler is more likely concerned about overall system and/or node power information, not the draw of a specific processor core or memory controller.

Portability for software calling the API By grouping the function calls by actor/system combination, we attempted to strike a balance between a totally non-intuitive, but generic get/put interface and one that is overly prescriptive by focusing on pre-identified and specific software packages. In addition to the actor/system calls, there is a set of calls to build the system “diagram” without having to rely on configuration files from a specific system type.

Flexibility for implementer of an API As this is a new area, the specification provides interfaces that are adaptable as hardware power technology evolves. The API is not based on any existing software-specific API. We can envision ways that interfaces such as RAPL, DVFS, NVML, BGQT/EMON, ACPI, the PAPI power interface, OpenMPI's hwloc package, etc., etc. can become implementations for certain actor/system interfaces.

We strived to create a portable, implementable interface for power-aware computing. We welcome all suggestions and comments.

Bibliography

[1] R. Bertran, Y. Sugawara, H. M. Jacobson, A. Buyuktosunoglu, and P. Bose. Application-level power and performance characterization and optimization on IBM Blue Gene/Q systems. In *IBM Journal of Research and Development*, volume 57, 2013.

[2] Grady Booch, Ivar Jacobson, and James Rumbaugh. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[3] M. Brocanelli, Sen Li, Xiaorui Wang, and Wei Zhang. Joint management of data centers and electric vehicles for maximized regulation profits. In *Green Computing Conference (IGCC), 2013 International*, pages 1–10, June 2013.

[4] Hao Chen, Can Hankendi, Michael C. Caramanis, and Ayse K. Coskun. Dynamic Server Power Capping for Enabling Data Center Participation in Power Markets. In *Proceedings of the International Conference on Computer-Aided Design*, ICCAD '13, pages 122–129, Piscataway, NJ, USA, 2013. IEEE Press.

[5] Yuan Chen, D. Gmach, C. Hyser, Zhikui Wang, C. Bash, C. Hoover, and S. Singhal. Integrated management of application performance, power and cooling in data centers. In *Network Operations and Management Symposium (NOMS), 2010 IEEE*, pages 615–622, April 2010.

[6] Yiannis Georgiou. Energy Accounting and Control on HPC clusters, November 2013. `http://perso.ens-lyon.fr/laurent.lefevre/greendayslille/greendayslille_Yiannis_Georgiou.pdf`.

[7] Yiannis Georgiou, Thomas Cadeau, David Glesser, Danny Auble, Morris Jette, and Matthieu Hautreux. Energy Accounting and Control with

SLURM Resource and Job Management System. In Mainak Chatterjee, Jian-Nong Cao, Kishore Kothapalli, and Sergio Rajsbaum, editors, *ICDCN*, volume 8314 of *Lecture Notes in Computer Science*, pages 96–118. Springer, 2014.

[8] Ryan E Grant, Michael Levenhagen, Stephen L Olivier, David DeBonis, Kevin T Pedretti, and James H Laros III. Standardizing power monitoring and control at exascale. *Computer*, 49(10):38–46, 2016.

[9] Ivar Jacobson. *Object Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1992.

[10] James H Laros, Suzanne M Kelly, Steven Hammond, Ryan Elmore, and Kristin Munch. Power/Energy Use Cases for High Performance Computing. Internal SAND Report SAND2013-10789. `https://cfwebprod.sandia.gov/cfdocs/CompResearch/docs/UseCase-powapi.pdf`.

[11] Zhenhua Liu, Yuan Chen, Cullen Bash, Adam Wierman, Daniel Gmach, Zhikui Wang, Manish Marwah, and Chris Hyser. Renewable and Cooling Aware Workload Management for Sustainable Data Centers. *SIGMETRICS Perform. Eval. Rev.*, 40(1):175–186, June 2012.

[12] Bryan Mills, Ryan E. Grant, Kurt B. Ferreira, and Rolf Riesen. Evaluating Energy Savings for Checkpoint/Restart. In *Proceedings of the 1st International Workshop on Energy Efficient Supercomputing*, E2SC '13, pages 6:1–6:8, New York, NY, USA, 2013. ACM.

[13] D.K. Newsom, S.F. Azari, A. Anbar, and T. El-Ghazawi. Locality-aware power optimization and measurement methodology for PGAS workloads on SMP clusters. In *Green Computing Conference (IGCC), 2013 International*, pages 1–10, June 2013.

[14] J.U. Patel, S.J. Guercio, A.E. Bruno, M.D. Jones, and T.R. Furlani. Implementing green technologies and practices in a high performance computing center. In *Green Computing Conference (IGCC), 2013 International*, pages 1–8, June 2013.

[15] P.V. Ramakrishna, G. Kaushik, K.L. Sudhakar, G. Thiagarajan, and A. Sivasubramaniam. Online system for energy assessment in large facilities - Methodology amp; A real-world case study. In *Green Computing Conference (IGCC), 2013 International*, pages 1–9, June 2013.

[16] Geri Schneider and Jason Winters. *Apply Use Cases: A Practical Guide, Second Edition.* Addison-Wesley, 2001.

[17] Hayk Shoukourian, Torsten Wilde, Axel Auweter, and Arndt Bode. Monitoring power data: A first step towards a unified energy efficiency evaluation toolset for hpc data centers. *Environmental Modeling & Software*, 2013.

[18] Tiffany Trader. Green Power Management Deep Dive. *Green Computing Report*, June 2013. `http://www.greencomputingreport.com/gcr/2013-06-05/green_power_management_deep_dive.html`.

[19] Abhinav Vishnu, Shuaiwen Song, Andres Marquez, Kevin Barker, Darren Kerbyson, Kirk Cameron, and Pavan Balaji. Designing energy efficient communication runtime systems: a view from PGAS models. *The Journal of Supercomputing*, 63(3):691–709, 2013.

[20] Sean Wallace, Venkatram Vishwanath, Susan Coghlan, John Tramm, Zhiling Lan, and Michael E. Papka. Application Power Profiling on IBM Blue Gene/Q. In *Proceedings of 2013 International Conference on Cluster Computing*. IEEE, 2013.

[21] V.M. Weaver, M. Johnson, K. Kasichayanula, J. Ralph, P. Luszczek, D. Terpstra, and S. Moore. Measuring Energy and Power with PAPI. In *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*, pages 262–268, Sept 2012.

[22] Xu Yang, Zhou Zhou, Sean Wallace, Zhiling Lan, Wei Tang, Susan Coghlan, and Michael E. Papka. Integrating Dynamic Pricing of Electricity into Energy Aware Scheduling for HPC Systems. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 60:1–60:11, New York, NY, USA, 2013. ACM.

Appendices

Appendix A

Topics Under Consideration for Future Versions

The following topics are either currently in active discussion or are planned to be addressed in future versions of the specification. In some cases it will be necessary to solicit additional feedback from the community to ensure we properly address the issue in future versions.

- **Support for Generic Notifications between Nodes** - Working on capability to send generic notifications between PowerAPI objects. This is expected to be useful for compatibility with other community projects and interoperability with job management.
- **Better Support for Frequency Scaling** - Frequency scaling features are not universal between hardware implementations. It is desirable to expose all of the possible behaviors of frequency scaling and we are working towards better descriptive solutions to represent these behaviors in the most accurate manner possible.
- **Coexistence of Implementations** - One of the driving questions for this future work is - how does one implementation interface with another? It is possible, even likely that an implementor will focus on implementing a portion or portions of the specification. This begs the question of how does implementation A interact with implementation B? Further, what role does the specification play in driving this interaction? We intend to work closely with the community to sort out this issue and document the appropriate guidance in the next version of the specification.

- **Language Bindings** - Some roles, system administrator for example, more commonly interface with the platform through shells, shell scripting or other interpretive languages like Perl or Python. We will investigate adding some or all of these capabilities, via specification and possibly prototypes, in future versions of the standard.
  - The next version of the specification will include a complete Python specification of all existing functions modified appropriately for the Python language
- **User Guide** - The addition of a user guide could provide additional useful information to both users and implementors. The addition of a users guide will be considered and if realized will accompany subsequent releases of the specification.
- **Hypothetical System Example** - We are considering creating a hypothetical system example to use to discuss and clarify concepts and higher level use cases. This will likely be included in the User Guide.
- **Required versus Optional, or Quality of Implementation** - We plan to clarify and document more precisely what portions of the specification are required to be implemented, what portions are optional and the definition of a quality implementation. This topic is complicated by the fact that implementors are free to implement portions of the specification.
  - Some progress has been made on this topic for version 1.1 but additional work is required.
- **Policies** - Security policies, priority of operations and privileges need to be further vetted and specified when appropriate. This topic has a large amount of intersection with the *Coexistence of Implementations* topic and will be considered jointly.
- **Unit Tests** - Development of a unit test infrastructure is under consideration, possibly to be associated with our prototype which will be released open source at a later date. Unit tests might also be a way for the implementation community to assure interaction between implementations of portions of the specification that will be required to work together.
- **User Supplied Functions** - We intend to investigate adding the ability for a user to supply a function for the purposes of generating a statistic, for example.

- **Multiple Platform Support** - Currently the specification only considers operation on a single platform. There is nothing preventing supporting multiple platforms and exposing multiple platforms in a single context in future versions. This will be considered for the next release in conjunction with the Coexistence of Implementation issue.
- **Generation Counter** - We intend to consider the addition of a generation counter capability to be used in conjunction with counters that have the potential for roll over. The generation counter could be used to inform the user that this has taken place. This concept likely has additional utility which is what will be explored for future releases of the specification. Target: 1.X - Implementation should handle overflow internally
- **Time Conversion/Overflow** - Time conversion convenience functions are being considered to convert between `PWR_Time` values and POSIX-compatible time representations. Included in this will be methods of detecting overflow during time value arithmetic.
- **Context Refresh** - We are considering adding the ability to refresh a context int he case of a long lived context such as one that is used by a persistent daemon. Yet to be resolved is what happens to existing pointers, more specifically what happens when the user has a pointer to an object that no longer exists after the refresh, or if this can happen.
- **Enhanced Support for ACPI 5.0** - Collaborative Processor Performance Control and Continuous Performance Control are currently not supported. Support will require new attributes and some function calls to allow for the flexible mechanisms provided in the ACPI 5.0 specification to allow expression of desired performance on a sliding, abstract unit-less scale. ACPI 5.0 also supports gathering statistics about the delivery of given performance values and the time spent in certain states, which we intend to address. We anticipate adding this support alongside the P-state and C-state functionality already in the Power API in a future version of the specification.
- **User/Resource Manager Interfaces** - Work needs to be done in this area but is best accomplished in collaboration with resource manager, work load manager experts. We hope to include standard interfaces for the user to query this system in future versions of the specification.
  - Work has begun to develop general report and information mining capabilities

- **HPCS Manager to Resource Manager Interface** - This interface clearly needs some work. Again it seems that this would benefit greatly from collaborative efforts.
  - Work has begun to develop general report and information mining capabilities

Appendix B

Change Log

The following list contains changes to the community Power API specification version 1.0

- First release of community Power API
- PWR_ObjGetSizeOfName - New function added that allows the user to get the size of buffer required to successfully call PWR_ObjGetName for a given object.
- Features to support Power Stack

Appendix C

Alternative Programming Language Bindings: Python

Acceptance of the Python language is growing within the HPCS System and Power Management communities for its ease of scripting and its interoperability with existing other applications and frameworks.

Python differs from C in many ways, including that it is an object-oriented language and implements garbage collection. Because of these differences, Python implementations of the API have some fundamental differences with the way they are used. The functionality is the same as the C API specification, but the "style" with which it is used is different in several respects, as described in the remainder of this appendix.

C.1 Introduction

The general structure of the C API specification is followed throughout this appendix. Differences between the Python language bindings and the C API specification are clearly noted. The subsections of this Python specification mirror the C API specification and are similarly labeled, e.g. section C.3.1 corresponds to 4.1 in the C API.

Python PEP-8 Standard Compliance

In general this Python API specification follows the PEP8 guidelines regarding coding standards and naming conventions. These guidelines are followed unless they create undue differences with how the C API specification defines the names of various entities. An example of this is the C function: `PWR_CntxtGetEntryPoint()`. In this Python Specification, `GetEntryPoint()` is an instance method of the `pwr.Cntxt` class. PEP8 suggests method and function names start with a lower case letter, e.g. `getEntryPoint()`; but for consistency with the C API, this Python specification uses `GetEntryPoint()`. A brief summary of the PEP8 naming conventions follows:

- Packages and Modules have short names with all lowercase.
- Classes are named with "CapWords" names, also known as "CamelCase". The first letter is capitalized.
- Exceptions follow the same rules as classes with the additional rule that "Error" be appended, such as "CamelCaseError".
- Global Variables, Method and Function names, and Instance Variables are named with "lower_case_with_underscores" or "mixedCase" (with the first character being lower case).
- Indentation and spacing are flexible within Python. See the PEP8 standard for an at-length discussion. C function parameters are aligned with the opening delimiter so Python methods and functions and their invocations are indented that way whenever possible.
- Non-Public or external variables are named with a "_leadingUnderscore". Virtual methods for which a child class method is required may or may not have a leading underscore depending on whether that method or its derivatives is meant to be publicly exposed. This document only defines the public methods available via the "opaque" API
- Constants are named with "ALL_CAPS".
- Keyword collisions are handled by appending an underscore to the colliding keyword, such as "open_".
- Always use "self" as the first argument for instance methods, and use "cls" as the first argument for class methods.

Implementation Note: Throughout this appendix, module function and class method names begin with an upper-case character.

C.2 Theory Of Operation

C.2.1 Overview

This section loosely follows the same tack as the C API chapter of the same name (chapter 3 on page 12). In particular, it discusses any remarkable caveats or differences between the Python Power API bindings and the C Power API specification. Because the intended audience of this section includes those with limited experience with Python, seasoned Python developers are forewarned of some pedantic explanations of Python behavior.

Python Version Agnosticism

This Python specification aims to be agnostic with respect to Python versioning where possible. It is currently designed to include version 2.7 and on. In particular, considerations needed to include version 2.7 are made with respect to integer typing (section C.2.1 on page 150) and enumerations (section C.2.1 on page 148). Enumerations as explained later in this section follow a C-style syntax instead of the "enum" data type available as of Python 3.4.[1]

The Python import statement

Throughout this appendix the "`pwr.`" prefix is used on API class methods, functions, and variables so that they are identifiable in the document. The example below illustrates a typical usage of an Python module implementation of the API imported as `pwr`:

```
import <your.implementation.library.path> as pwr
myPwrCntxt = pwr.Cntxt(...)
```

Memory Management and Garbage Collection

The C API specification is very precise in defining the enumerations, structures, and entry-points that are to be used to access the power measurement and control functionality exposed by the specification. In the pass-by-value C language, "objects" are precisely defined structures that are allocated (and freed) by the user of the API or the implementation of the API. The functions defined by the C API use opaque handles to identify or point to these objects to represent them and pass them around from place to place. Refer to the C API "Theory of Operation" section 3 on page 12 for the language independent information.

[1]https://docs.python.org/3/library/enum.html

Python is an object-oriented language, and uses "pass-by-object-reference" to reference and transport data. Python treats everything as an object, including references to other objects, however, it has no native concept of pointers to locations in memory. Because of this, Python uses a garbage collection mechanism to clean up previously instantiated objects which have become completely de-referenced.

For example, the C API call `PWR_CntxtInit()` returns a handle "myPwrCntxt" that may identify or point to a location in memory containing the context structure:

```
/* Example C code */
PWR_Cntxt myPwrCntxt;
rc = PWR_CntxtInit(PWR_CNTXT_DEFAULT, PWR_ROLE_RM, "foo", &myPwrCntxt)
    ;
```

The memory region pointed to by myPwrCntxt must be returned to the heap once the application is done using it. This explicit memory management is some of what necessitates the use of clean-up functions such as `PWR_CntxtDestroy(myPwrCntxt)`.

With Python, "myPwrCntxt" is a reference to an instance of the `pwr.Cntxt` object class. Python counts the number of references to its objects, and once that count goes to "0", the object is garbage collected (freed). De-referencing an object such as "myPwrCntxt" can be implicitly performed by returning from the method or function in which it is being used, or explicitly performed by calling "del":

```
import <your.implementation.library.path> as pwr
def someFunction():
    myPwrCntxt = pwr.Cntxt(pwr.CntxtType.DEFAULT, pwr.Role.RM, "foo")
    localEPObj = myPwrCntxt.GetEntryPoint()
    ...
    # Once this function returns, localEPObj is out of its scope,
    which
    # causes Python garbage collection to free its memory. myPwrCntxt
    is
    # not garbage collected as it is needed/returned to the caller.
    return myPwrCntxt

def anotherFunction():
    LocalPwrCntxt = someFunction()
    ...
    # Explicitly releasing the LocalPwrCntxt object returned from
    # someFunction() by calling del.
    del LocalPwrCntxt
    ...  # More to be done, but no further need for the LocalPwrCntxt
    object.
```

If myPwrCntxt is being used elsewhere, such as being passed via the return statement in a particular method or function, it is not freed (garbage collected) until all possible usage is out of scope. This automatic garbage collection eliminates the necessity to explicitly free the memory used by "myPwrCntxt".

Encapsulation - Methods are part of the data classes

Another feature of Python is its object-oriented approach to defining data and the methods that act on that data. Since much of the C API specification is written with collections of functions using a few handles or pointers to identify common data structures as parameters, Python's object-oriented approach fits well with the existing C API. In Python, classes are defined, and those classes contain methods appropriate for the particular class. For instance in C, `PWR_CntxtInit()` returns an opaque handle, "myPwrCntxt". That handle must be passed into functions as a parameter for that function to know what data to act on:

```
/* Example C code: */
PWR_Cntxt myPwrCntxt;
PWR_Obj   myPwrObj;

rc = PWR_CntxtInit(PWR_CNTXT_DEFAULT, PWR_ROLE_RM, "foo", &myPwrCntxt)
    ;
if (rc != PWR_RET_SUCCESS)
  exit(-rc);
rc = PWR_CntxtGetEntryPoint(myPwrCntxt, &myPwrObj);
if (rc != PWR_RET_SUCCESS)
  exit(-rc);
```

Note: The example above adds error handling that validates the return code of the two C API functions called. This causes the C example to exit if one of the calls fail, resulting in the same "exit on error" behavior that the Python example that follows will exhibit. More on Python error handling in C.2.1 on page 151.

In this Python API, `myPwrCntxt` is an object that is an instance of the `pwr.Cntxt` class and that object contains methods for acting on that object:

```
myPwrCntxt = pwr.Cntxt(pwr.CntxtType.DEFAULT, pwr.Role.RM, "foo")
myPwrObj = myPwrCntxt.GetEntryPoint()
```

Descriptions and many more examples of class object and methods can be found in section C.4 starting on page 170.

Enumerations

The C API relies heavily on C-style enumerations which, amongst other things, enforce strict type-checking, and can provide automatic incrementing of the enumeration's value. Python 2.7 has no direct support for enumerations. This Python API specification defines support for enumerations so that they match in style and functionality with the defined enumerations in the C API. This enables strict type-checking and automatic incrementing of the enumeration's value definitions:

```
# Example Colors Enumeration (not part of the specification)
class Colors(_EnumerationClass):
    pass
Colors("Blue")
Colors("Green")
Colors("Red")
Colors("Yellow")
# Vendor implementations can add more entries here or look at
# the object-oriented "Extending Existing Enumerations" description
    below.
Colors("NUM_COLORS")
Colors("INVALID", -1)
Colors("NOT_SPECIFIED", -2)
```

```
# Example Sizes Enumeration (not part of the specification)
class Sizes(_EnumerationClass):
    pass
Sizes("Small")
Sizes("Medium")
Sizes("Large")
# Vendor implementations can add more entries here or look at
# the object-oriented "Extending Existing Enumerations" description
    below.
Sizes("NUM_SIZES")
Sizes("INVALID", -1)
Sizes("NOT_SPECIFIED", -2)
```

Implementation Note: Defining a class such as Colors with only the do-nothing `pass` statement imposes strict type-checking against the Colors class instead of the generic (and externally defined) `_Enumeration` class. The subsequent "constructors" being called to define the specific enumerations such as Colors("Green") actually add the definition into the Colors class itself, as opposed to instantiating the Colors class. This is implemented using Python "MetaClasses".

Enumerations Extension Enumerations can be extended in two ways: (1) By adding more entries before the `NUM_classname` entry, as shown in the example above, or (2) by extending a defined `EnumerationClass` in an object-oriented way, as illustrated here:

```
# Likely in some other vendor specific file...
Colors("Black", Colors.NUM_COLORS)
Colors("NUM_COLORS")  # to reset NUM_COLORS
#
Sizes("X-Large", Sizes.NUM_SIZES)
Sizes("NUM_SIZES")  # to reset NUM_SIZES
```

The code examples above show how a vendor implementation can extend a defined `EnumerationClass` in this specification.

Enumerations Usage The following examples show Python code using some of the enumerations defined in section C.3.3 starting on page 159:

```
# Using enumerations as parameters
myPwrCntxt = pwr.Cntxt(pwr.CntxtType.DEFAULT, pwr.Role.RM, "foo")

# Return the value of an enumeration
enumVal = int(pwr.Role.RM)  # Gives a value of 4 to enumVal

# Return the name of the enumeration:
enumStr = str(pwr.Role.RM)  # Gives "RM" to enumStr

# Strict type-checking can be enforced
class Cntxt():
    def __init__(self, cntxtType, cntxtRole, cntxtName):
    ...
    if not isinstance(cntxtType, CntxtType):
            raise PwrError( ReturnCode.BAD_VALUE,
                "{0:s}: Invalid context type.".format(self.__class__.
    __name__))
    ...
```

Numeric Types

Python represents integer, unlimited length integer, and floating point numbers with its respective "int", "long" and "float" built-in types. A Python "float" is equivalent to a C double. Unlike Python 3.0, Python 2.7 still distinguishes an "int" type from an unlimited-length "long". Python implementations of this API shall use long for all integer types.

Time Entities

The C API specification in section 4.8 on page 27 describes encapsulating a 64-bit integer representing a time value in nanoseconds. Python represents a time value in its `time` module using a floating point number representing time in seconds. Since Python time values are a floating point values (e.g., 13.434582349 seconds) no precision is lost representing time this way. Conversion back and forth between the C style integer representation and Python's default floating point format can be done as follows:

```
# Convert Python's floating point "seconds" value to a long
# integer representing "nanoseconds":
timeIntNs = long(timeFloatSec * 1000000000.0)

# Convert a long integer representing a count of "nanoseconds" to a
    Python
# "float" value containing a possible fractional value of "seconds":
timeFloatSec = float(timeIntNs) / 1000000000.0
```

The preferred Python implementation for handling time in this API is described in section C.3.8 on page 166.

Error Handling

Error handling in the C API specification generally involves checking for a non-zero integer return-code value. Python supports two methodologies for robust error handling: (1) exceptions and (2) the ability of a method or function to return multiple values (in the form of a tuple). Returning multiple values enables the return of an error code alongside of an expected (or unexpected) result. This Python API specification uses Python exceptions for things that are generally seen as fatal errors. Some functions in the API that "get" or "set" values use per-value return codes to continue the operation in the face of non-fatal errors.

Exceptions give the ability to funnel any errors in a Python method or function through one or more error tracking regions wrapped by Python `try/except` clauses. Errors that occur while instantiating objects, i.e. when the class's `__new__()` and `__init__()` methods are invoked, need to be handled using Python exceptions. This is primarily because some of the Python library initialization methods raise exceptions themselves and a class `__init__()` method does not return any value at all. For these reasons, this Python specification uses the Exception methodology.

In this specification, some methods return arrays of measurement information that may contain error information for individual measurements. These methods may encounter an error reading an attribute on a particular object in the group, but the rest of the attributes from the operation should not be thrown away. For this type of functionality exceptions are only raised in the case of a fatal error, and per-element access errors are handled with error status information in results data structures.

For "Get" methods that return or yield multiple results from multiple operations such as the `AttrGetValue(s)` methods and the `pwr.Stat.GetValue(s)` methods, the `pwr.ReturnCode` value is included with the measurement data for any particular operation. If the return code value is set to `pwr.ReturnCode.SUCCESS`, then the measurement data is valid. Otherwise the data is invalid, and the return code is set to be something besides `pwr.ReturnCode.SUCCESS`. See C.4.4 on page 180 for more details.

Similarly, "Set" operations will yield failure details when errors occur. If there are no errors, nothing will be generated. See C.4.4 information returned.

Class PwrError An example is shown below of the definition of the `Exception` class that is used throughout this Python API specification. See section C.3.7 on page 164 for supported API ReturnCodes. All exceptions generated by implementation of this specification should use and check for this Exception class. Other system-level exceptions that may occur should be accounted for but are not described in this document. Please refer to system-level Python documentation for details on these other exceptions.

```
class PwrError(EnvironmentError):
    def __init__(self, returnCode, errorMsg = None):
        # returnCode      --> e.errno
        # str(returnCode) --> e.strerror
        # errorMsg        --> e.errmsg
        ...
```

Below is an example of the `pwr.PwrError` exception:

```
import <your.implementation.library.path> as pwr

def someOptionalMethod():
    # Is not implemented
    raise PwrError( ReturnCode.NOT_IMPLEMENTED,
        "{0:s}: someOptionalMethod not implemented!".format(
            self.__class__.__name__))

def ExampleOfExceptionHandling():
    ...
    defaultResult = None
    try:
        theResult = myPwrObj.someOptionalMethod()
        theResult += myPwrObj.SomeOtherMethod()

    except PwrError as e:
        print "ERROR!"
        print e.errno     # "-2"
        print e.strerror  # "NOT_IMPLEMENTED"
        print e.errmsg    # "someOptionalMethod not implemented!"
        return defaultResult
        # "raise" can also be called here.
    else:
        return theResult
```

Multiple Return Values Python seemingly has the capability to return multiple values from a method or function. These multiple values are actually contained within a single object. This follows from the fact that everything is an object in Python, including tuples, which is an object that contains a collection of other objects:

```
def SomeMethodOrFunction():
    return 1, 2
    # or "return (1, 2)"
```

The method or function can be called two ways: either returning the elements of the tuple or the entire tuple as one indexed object (an array). Here the tuple's elements are returned:

```
retval1, retval2 = SomeMethodOrFunction()
# or "(retval1, retval2) = SomeMethodOrFunction(arg)"
print str(retval1)  # prints "1"
print str(retval2)  # prints "2"
```

Here everything is treated as the singular-indexed tuple object:

```
retval = SomeMethodOrFunction()
print str(retval[0])  # prints "1"
print str(retval[1])  # prints "2"
```

In the case where a method or function returns a tuple of several arrays that relate one-to-one with each other element-wise, they can be re-arranged to be an array of tuples. This arrangement which may be more programmatically simple to iterate over, among other things. Below is an example of how to convert a tuple of arrays to an array of tuples using the standard built-in Python function, `zip`:

```
def SomeGroupMethodOrFunction(arg):
    return [1,2,3], [4,5,6]
array1, array2 = SomeGroupMethodOrFunction(arg)
print array1         # prints "[1,2,3]"
print array2         # prints "[4,5,6]"
combinedArray = zip(array1, array2)
print combinedArray  # prints [(1,4),(2,5),(3,6)]
```

Iterators and Generators

Python has a special `yield` statement which allows a method or function to return or "generate" a result without exiting from the logic flow of that method or function. This allows an "Iterator" method/function to loop and collect or act upon the particular objects that a "Generator" method/function may yield. This technique can be used to avoid the creation of very long, memory intensive lists of objects such as those represented by a `pwr.Grp` object.

In this Python specification, "Generators" are exposed as part of the API. They complement the normal, list-giving methods as defined in the C API, but do not replace them. They are prefixed with the name "Generate". For example, the method `GenerateChildren()` complements the `GetChildren` method documented in C.4.2 on page 174). The following are some example functions illustrating the use of iterators and generators versus lists:

```
# This function creates a list of objects and returns it to the
    calling function.
# The entire list is created in memory before it is returned to the
    caller.
def ObjectLister(numObjs):
    objList = []
    for objNum in numObjs:
        newObj = ExampleObj()
        objList.append(newObj)
    return objList

# This function generates objects on the fly, and yields each object
    it creates
# to the calling function one at a time. The code flows back into this
     generator
# on each iteration of the calling functions for loop.
def ObjectGenerator(numObjs):
    for objNum in numObjs:
        newObj = ExampleObj()
        yield newObj

def ObjectConsumer():
    for someObj in ObjectGenerator(1000000):
        # ObjectGenerator has yielded another object for this function
     to use.
        useObjectOnce(someObj)  # Each individual "someObj" is garbage
     collected

    # Here ObjectLister() creates/returns the entire list all at once.
    for someObj in ObjectLister(1000000):
        # This function iterates over the list, and then when it is
    done with the
        # entire list, the list and its objects get garbage collected.
        useObjectOnce(someObj)
```

Shortcuts using Properties

Python offers the ability to attach "properties" to class methods and to overload operators so that simple "set" and "get" methods and operators on a class can appear and behave like standard class-variables. These shortcuts make for simpler and more concise code. Throughout this Appendix, these shortcuts will be mentioned for the various functions, methods and operators for which they are available.

```
# The standard way...
myEntryPoint = cntxt.GetEntryPoint()
myAttr = myPwrObj.AttrGetValue(pwr.AttrName.TEMP)
myAttrMeta = myPwrObj.AttrGetMeta(pwr.AttrName.TEMP, pwr.MetaName.
    SAMPLE_RATE)
unionGroup = myPwrGrp.Union(someOtherPwrGrp)

# With properties...
myEntryPoint = cntxt.entrypoint
myAttr = myPwrObj.TEMP.value
myAttrMeta = myPwrObj.TEMP.SAMPLE_RATE
unionGroup = myPwrGrp | someOtherPwrGrp
```

C.2.2 Power API Initialization

Initialization is accomplished by instantiating the `pwr.Cntxt` class which returns a `pwr.Cntxt` object:

```
import <your.implementation.library.path> as pwr
myPwrCntxt = pwr.Cntxt(CtxtType,  # pwr.CntxtType
                       Role,      # pwr.Role
                       Name)      # Python str
```

C.2.3 Roles

All of the same roles in the C API specification (section 3.3 on page 13) should be supported by valid Python implementations.

C.2.4 System Description

All of the object types in the C API "System Description" (section 3.4 on page 14) are represented by a Python base class, `pwr.Obj`. The `pwr.Obj` base class supports functionality such as: obtaining an object's parent, obtaining its children, getting the object's type, and navigating the object tree. Power object type specific functionality is represented in child classes of the base `pwr.Obj` class.

The C API on page 14, states that a variety of object types are to be defined, but not necessarily used or supported. These are: "Platform", "Cabinet", "Chassis", "Board", "Node", "Socket", "Power Plane", "Core", "Memory", and "NIC". In the C API specification, opaque handles, which can be pointers, are used to point to these various abstracted objects. However, in Python, separate child classes to the `pwr.Obj` class are created to represent these various types of objects. These object types are represented by respectively named child classes such as `pwr.ObjPlatform`, `pwr.ObjCabinet`, and `pwr.ObjBoard`.

C.2.5 Attributes

Each Python `pwr.Obj` object has two sets of associated attributes, “Global” and “Explicit”. Refer to the C API “Attributes” section 3.5 on page 18 for language independent details. Global attributes are guaranteed to exist on every type of `pwr.Obj`.

Global attributes are accessed through methods defined in the base `pwr.Obj` class, such as `GetName`, `GetType`, `GetParent`, `GetChildren/GenerateChildren`, e.g.:

```
myType = myPwrObj.GetType()
myName = myPwrObj.GetName()
myParent = myPwrObj.GetParent()
myChildren = myPwrObj.GetChildren()
```

The `GetEntryPoint` context method is accessed via the `pwr.Cntxt` class and returns the entry point object of the context’s system description:

```
import <your.implementation.library.path> as pwr
myPwrCntxt = pwr.Cntxt(pwr.CntxtType.DEFAULT, pwr.Role.MC, "
    MonitorAndControl")
myPwrObj = myPwrCntxt.GetEntryPoint()
```

Explicit attributes are attributes that may be unique to one or more `pwr.Obj` object types. They are accessed via the attribute interface. For details on Python attribute methods, see section C.4.4 on page 180:

```
attrName = pwr.AttrName.POWER
measurement1 = myPwrObj.AttrGetValue(attrName)
measurement2 = myPwrObj.AttrGetValue(pwr.AttrName.VOLTAGE)
```

The attribute interface is preferred over explicit methods so that additional API methods are not necessary to expand functionality for a particular object type.

C.2.6 Metadata

Metadata are supported as detailed in section 3.6 on page 19 of the C API.

To access metadata for a particular object attribute:

```
attrName  = pwr.AttrName.PSTATE
metaName  = pwr.MetaName.MIN
minPstate = myPwrObj.AttrGetMeta(attrName, metaName)
maxPstate = myPwrObj.AttrGetMeta(pwr.AttrName.PSTATE, pwr.MetaName.MAX
    )
```

C.2.7 Thread Safety

There is no difference in the way threading is to be handled versus what is described in the C API. Please refer to the C API specification (section 3.7 on page 19) for a discussion on threading and multiprocessing concerns.

C.3 Type Definitions

This chapter lists the enumerations and classes associated with the Python version of the Power API and mirrors the naming and numbering used in the C API specification (found in chapter 4 starting on page 20). The enumerations listed in this section are required to exist, but not all enumerated values are required to be supported by any specific Python implementation of the Power API. Some of the enumerations are meant to be expanded, while some are not. Each of the sections below discuss what compliant Python versions of these enumerations and structures (classes) look like and whether they can be expanded upon.

C.3.1 Opaque Types

The opaque types described in 4.1 on page 20 are represented as Python base classes with the exception of the `PWR_Status` structure. The `PWR_Status` structure is used for returning error status on functions that perform multiple operations. Python implementations handle these errors differently and do not need to use the `PWR_Status` structure.

```
class Cntxt(...):
    ...
class Grp(...):
    ...
class Obj(...):
    ...
class Stat(...):
    ...
```

These opaque abstract classes are meant to be overloaded by specific implementations of this Python API. The discussion of the object type child classes of the `pwr.Obj` class in section C.2.4 on page 156 illustrates this.

C.3.2 Globally Relevant Definitions

The Python bindings support all of the C API's global definitions (see page 20), such as the version number definitions that are useful in the Python API. Definitions like the maximum length of text-strings, are not useful since Python automatically handles allocation and garbage collection for strings. PWR_MAJOR_VERSION and PWR_MINOR_VERSION are exposed through the `pwr.GetMajorVersion()` and `pwr.GetMinorVersion()` functions defined in section C.4.7 on page 193).

C.3.3 Context Relevant Type Definitions

The Python bindings support the necessary power contexts needed for implementation and follows the design of the C API as defined on page 21. Power contexts use the Python enumeration scheme described in C.2.1 on starting on page 148 and contain a single "Default" power context enumeration. The default context type carries with it the default capabilities of the API. For vendor, platform, and model-specific capabilities, implementors can add new context types.

Enumeration Class CntxtType

All implementations must support the `"DEFAULT"` context. The corresponding C API enumeration is on page 22. The following is the enumeration for the default context type:

```
class CntxtType(_EnumerationClass):
    pass
CntxtType("DEFAULT")
```

Add the following to extend the enumeration for context types for a new, non-default vendor. (see "Enumerations Extension" C.2.1 on page 149):

```
CntxtType("VENDORNAME")
```

Enumeration Class Role

Default roles are defined by the `pwr.Role` enumeration. All contexts support one or more of these roles. See page 22 for the C API `enum` definition.

```
class Role(_EnumerationClass):
    pass
Role("APP")     # Application
Role("MC")      # Monitor and Control
Role("OS")      # Operating System
Role("USER")    # User
Role("RM")      # Resource Manager
Role("ADMIN")   # Administrator
Role("MGR")     # HPCS Manager
Role("ACC")     # Accounting
#
# Vendor implementations SHALL NOT add roles!
#
Role("NUM_ROLES")
Role("INVALID", -1)
Role("NOT_SPECIFIED", -2)
```

C.3.4 Object Relevant Type Definitions

Enumeration Class ObjType

All implementations of the Power API are required to have the following object types enumerated. Implementations may add object types to these defaults, but must do so using the methods described in C.2.1 on page 149. The corresponding C API enumeration is on page 23:

```
class ObjType(_EnumerationClass):
    pass
ObjType("PLATFORM")
ObjType("CABINET")
ObjType("CHASSIS")
ObjType("BOARD")
ObjType("NODE")
ObjType("SOCKET")
ObjType("CORE")
ObjType("POWER_PLANE")
ObjType("MEM")
ObjType("NIC")
# Vendor implementations can add more entries here or look at
# the object-oriented "Enumerations Extension" description previously.
ObjType("NUM_OBJ_TYPES")
ObjType("INVALID", -1)
ObjType("NOT_SPECIFIED", -2)
```

C.3.5 Attribute Relevant Type Definitions

Enumeration Class AttrName

The following default attributes must be enumerated in any implementation of this API. If more attributes are desired to be added for a particular implementation, see the methods described in C.2.1 on page 149. The corresponding C API enumeration is on page 24:

```
class AttrName(_EnumerationClass):
    pass
AttrName("PSTATE")          # Python long
AttrName("CSTATE")          # Python long
AttrName("CSTATE_LIMIT")    # Python long
AttrName("SSTATE")          # Python long
AttrName("CURRENT")         # Python float, amps
AttrName("VOLTAGE")         # Python float, volts
AttrName("POWER")           # Python float, watts
AttrName("POWER_LIMIT_MIN") # Python float, watts
AttrName("POWER_LIMIT_MAX") # Python float, watts
AttrName("FREQ")            # Python float, Hz
AttrName("FREQ_LIMIT_MIN")  # Python float, Hz
AttrName("FREQ_LIMIT_MAX")  # Python float, Hz
AttrName("ENERGY")          # Python float, joules
AttrName("TEMP")            # Python float, degrees Celsius
AttrName("OS_ID")           # Python long
AttrName("THROTTLED_TIME")  # Python long
AttrName("THROTTLED_COUNT") # Python long
# Vendor implementations can add more entries here or look at
# the object-oriented "Enumerations Extension" description previously.
AttrName("NUM_ATTR_NAMES")
AttrName("INVALID", -1)
AttrName("NOT_SPECIFIED", -2)
```

Built-in Support for AttrDataType

In the C API specification a `pwr.AttrDataType` enumeration is defined. This is to ease the type checking of data coming from various power attributes. In Python, object "typing" is built-in such that this enumeration becomes redundant and not meaningful.

There are two basic data types found to represent the various values that may be used with any valid Python API methods; "long" and "float". They may be type-checked as follows:

```
val = SomeMethodOrFunction()
if not isinstance(val, float):
    raise pwr.PwrError(pwr.ReturnCode.BAD_VALUE, "Bad temperature
    value returned!")
```

Class AttrAccessError

In the Python API, the `PWR_AttrAccessError` (along with the `PWR_Stat` structure), have been replaced by the functionality of *measurement* named tuples and lists of *measurement* named tuples. For details on these named tuples, please refer to the discussion on `AttrGetValue` and `AttrSetValue` starting at C.4.4 on page 180.

C.3.6 Metadata Relevant Type Definitions

Enumeration Class MetaName

The default implementation/context must at least have these Metadata names enumerated. Additional metadata names may be defined using the methods described in C.2.1 on page 149. The corresponding C API enumeration is on page 26:

```
class MetaName(_EnumerationClass):
    pass
MetaName("NUM")              # Python long
MetaName("MIN")              # Python long or Float (depending on attr
    . type)
MetaName("MAX")              # Python long or Float (depending on attr
    . type)
MetaName("PRECISION")        # Python long
MetaName("ACCURACY")         # Python Float
MetaName("UPDATE_RATE")      # Python Float
MetaName("SAMPLE_RATE")      # Python Float
MetaName("TIME_WINDOW")      # pwr.Time object
MetaName("TS_LATENCY")       # pwr.Time object
MetaName("TS_ACCURACY")      # pwr.Time object
MetaName("MAX_LEN")          # Python long (max length of any metadata
     string)
MetaName("NAME_LEN")         # Python long (max length of NAME)
MetaName("NAME")             # Python String
MetaName("DESC_LEN")         # Python long (max length of DESC)
MetaName("DESC")             # Python String
MetaName("VALUE_LEN")        # Python long (max length of meta value
    at index)
MetaName("VENDOR_INFO_LEN")  # Python long (max length of VENDOR_INFO
MetaName("VENDOR_INFO")      # Python String
MetaName("MEASURE_METHOD")   # Python long (0/1 depending on real/
    model meas.)
# Vendor implementations can add more entries here or look at
# the object-oriented "Enumerations Extension" description previously.
MetaName("NUM_META_NAMES")
MetaName("INVALID", -1)
MetaName("NOT_SPECIFIED", -2)
```

Implementation Note: The "LEN"-related definitions above are not useful in any Python implementation but are included for consistency with the C API specification.

C.3.7 Error Return Definitions

Enumeration Class ReturnCode

The following error definitions are required to be defined for every implementation of the API. New return code definitions may be added at the end of this list. The corresponding C API enumeration is on page 26.

```
class ReturnCode(_EnumerationClass):
    negative = True
ReturnCode("WARN_NO_GRP_BY_NAME",  5)
ReturnCode("WARN_NO_OBJ_BY_NAME",  4)
ReturnCode("WARN_NO_CHILDREN",     3)
ReturnCode("WARN_NO_PARENT",       2)
ReturnCode("WARN_NOT_OPTIMIZED",   1)
#
ReturnCode("SUCCESS")            #  0
ReturnCode("FAILURE")            # -1
ReturnCode("NOT_IMPLEMENTED")    # -2
ReturnCode("EMPTY")              # -3
ReturnCode("INVALID")            # -4
ReturnCode("LENGTH")             # -5
ReturnCode("NO_ATTRIB")          # -6
ReturnCode("NO_META")            # -7
ReturnCode("READ_ONLY")          # -8
ReturnCode("BAD_VALUE")          # -9
ReturnCode("BAD_INDEX")          # -10
ReturnCode("OPT_NOT_ATTEMPTED")  # -11
ReturnCode("NO_PERM")            # -12
ReturnCode("OUT_OF_RANGE")       # -13
ReturnCode("NO_OBJ_AT_INDEX")    # -14
```

C.3.8 Time Related Definitions

In the C API, `uint64` (unsigned 64-bit integer) values are used to represent a value in nanoseconds. Native Python time values are stored in floating point format. A full description of why Python implementations of the Power API needs some extra features is given in section C.2.1 on page 151. The preferred implementation for handling time in Python implementations of the Power API is as follows:

Class Time

```
class Time(long):
    def __init__(self, timeVal):
        # convert to ns if timeVal is a float (seconds) value
        ...
# To access:
now = pwr.Time():    # A generic, opaque time value
nowNs = long(now)    # Converts and returns the time value to a long
    integer
                     #   representing a number of nanoseconds.
nowSec = float(now)  # Converts and returns the time value to a
    floating point
                     # value representing a number of seconds.
```

A set of definitions is used for defining what a time value set to `None` means:

```
PWR_TIME_UNINIT  = None  # Time value was never initialized
PWR_TIME_UNKNOWN = None  # Time value was never recorded
```

Class TimePeriod

The `PWR_TimePeriod` struct in C on page 28, is represented in Python by a class:

```

class TimePeriod():
    def __init__(self,
                 start = TIME_UNINIT,
                 stop = TIME_UNINIT,
                 instant = TIME_UNINIT):
        self._start = Time(start)
        self._stop  = Time(stop)
        self._instant = Time(instant)
    ...
```

To access the various data items of the pwr.TimePeriod class instance, “myTimePeriod”:

```
# Get (read) access
startTimeSec = float(myTimePeriod.start)  # as Seconds     (float
    value)
stopTimeNs = long(myTimePeriod.stop)      # as Nanoseconds (long value
    )
instantTime = myTimePeriod.instant        # as pwr.Time    (pwr.Time
    object)
# Set (write) access
myTimePeriod.start = startTimeNs          # (a long value)
myTimePeriod.stop = stopTimeSec           # (a float value)
myTimePeriod.instant = PWR_TIME_UNINIT    # (None)
```

C.3.9 Statistics Relevant Type Definitions

For background on the overall support for statistics in the Power API refer to section 4.9 on page 28, and the "Statistics Functions" in section 5.6 starting on page 70.

Enumeration Class AttrStat

The following class `AttrStat(_EnumerationClass)` includes the list of currently-defined statistics potentially available to the user of an implementation. Additional Statistics operations may be vendor-defined using the methods described in C.2.1 on page 149. See section 4.9 on page 29 for the C API enumeration:

```
class AttrStat(_EnumerationClass):
    pass
AttrStat("MIN")
AttrStat("MAX")
AttrStat("AVG")
AttrStat("STDEV")
AttrStat("CV")
AttrStat("SUM")
# Vendor implementations can add more entries here or look at
# the object-oriented "Enumerations Extension" description previously.
AttrStat("NUM_ATTR_STATS")
AttrStat("INVALID", -1)
AttrStat("NOT_SPECIFIED", -2)
```

Enumeration Class ID

The C API definition for the PWR_ID enumeration is on page 29. Python implementations use ID-enumerated types in support of the method GetReportByID in section C.5.1. Vendor specific additions to this enumeration class can be added using the methods described in C.2.1 on page 149. The corresponding C API enumeration is on page 29:

```
class ID(_EnumerationClass):
    pass
ID("USER")
ID("JOB")
ID("RUN")
#
ID("NUM_IDS")
ID("INVALID", -1)
ID("NOT_SPECIFIED", -2)
```

C.3.10 OS Hardware Type Definitions

Class OperState

A Python class represents the C API `PWR_OperState` structure as follows:

```
class OperState():
    def __init__(self, cStateNum, pStateNum):
        self.c_state_num = cStateNum
        self.p_state_num = pStateNum
    ...
```

To access the various data items of the `pwr.OperState` class instance, `myOpState`:

```
# Get (read) access
cState = myOpState.c_state_num
pState = myOpState.p_state_num
# Set (write) access
myOpState.c_state_num = pwr.SleepState.SHALLOW
myOpState.p_state_num = pwr.PerfState.FASTEST
```

C.3.11 Application OS Interface Type Definitions

Enumeration Class RegionHint

Please see page 30 for the C API description of this enumeration.

```
class RegionHint(_EnumerationClass):
    pass
RegionHint("DEFAULT")
RegionHint("SERIAL")
RegionHint("PARALLEL")
RegionHint("COMPUTE")
RegionHint("COMMUNICATE")
RegionHint("IO")
RegionHint("MEM_BOUND")
# Vendor implementations can add more entries here or look at
# the object-oriented "Enumerations Extension" description previously.
RegionHint("NUM_REGION_HINTS")
RegionHint("INVALID", -1)
RegionHint("NOT_SPECIFIED", -2)
```

Enumeration Class RegionIntensity

Please see page 31 for the C API description of this enumeration.

```
class RegionIntensity(_EnumerationClass):
    pass
RegionIntensity("HIGHEST")
RegionIntensity("HIGH")
RegionIntensity("MEDIUM")
RegionIntensity("LOW")
RegionIntensity("LOWEST")
RegionIntensity("NONE")
# Vendor implementations can add more entries here or look at
# the object-oriented "Enumerations Extension" description previously.
RegionIntensity("NUM_REGION_INTENSITIES")
RegionIntensity("INVALID", -1)
RegionIntensity("NOT_SPECIFIED", -2)
```

Enumeration Class SleepState

Please see page 31 for the C API description of this enumeration.

```
class SleepState(_EnumerationClass):
    pass
SleepState("NO")
SleepState("SHALLOW")
SleepState("MEDIUM")
SleepState("DEEP")
SleepState("DEEPEST")
# Vendor implementations can add more entries here or look at
# the object-oriented "Enumerations Extension" description previously.
SleepState("NUM_SLEEP_STATES")
SleepState("INVALID", -1)
SleepState("NOT_SPECIFIED", -2)
```

Enumeration Class PerfState

Please see page 32 for the C API description of this enumeration.

```
class PerfState(_EnumerationClass):
    pass
PerfState("FASTEST")
PerfState("FAST")
PerfState("MEDIUM")
PerfState("SLOW")
PerfState("SLOWEST")
# Vendor implementations can add more entries here or look at
# the object-oriented "Enumerations Extension" description previously.
PerfState("NUM_PERF_STATES")
PerfState("INVALID", -1)
PerfState("NOT_SPECIFIED", -2)
```

C.4 Core (Common) Interface Methods

Core Interface Methods fall into the following categories:

- **Initialization**
- **Navigation**
- **Group**
- **Attribute**
- **Metadata**
- **Statistics**

For background information on the methods described in this section, please refer to the C API function descriptions starting at page 33. Many of the Interface methods defined are implemented as class instance constructor methods in Python versions of this API. Because of the garbage collection capabilities of Python, some of the "Destroy" methods are not needed. Those differences are noted in the following sections. If an error occurs instantiating an object, a `pwr.PwrError` exception is raised.

C.4.1 Initialization

Method Cntxt

A context is an instance of the `pwr.Cntxt` class:

```
class Cntxt():
    def __init__(self, cntxtType, cntxtRole, cntxtName):
        ...
    def GetEntryPoint():
        ...
```

Note: the `try/except` clause has been added for example purposes, but is not included in all the code examples throughout this document. See general discussion about Python Error Handling in section C.2.1 on page 151.

To instantiate a default power context for a user role:

```
try:
    myPwrCntxt = pwr.Cntxt(pwr.CntxtType.DEFAULT, pwr.Role.RM, "
    Default")
except pwr.PwrError as e:
    print str(e.errno)
    print e.errmsg
    print e.strerror
#
# Where:
#   cntxtType is a pwr.CntxtType
#   pwrRole is a pwr.Role type
#   cntxtName is a Python str
# Returns:
#   myPwrCntxt is a pwr.Cntxt context
# This method raises a pwr.PwrError exception when something goes
    wrong.
# The possible exception errors are:
#   pwr.ReturnCode.FAILURE
```

Method CntxtDestroy NOT_IMPLEMENTED

Because Python implements garbage collection, there is no need to de-initialize or destroy a context, and a `CntxtDestroy` method need not be implemented.

C.4.2 Hierarchy Navigation Methods

Method GetEntryPoint

Once a context has been established, the entry point in the object tree can be queried. Each context has its own entry point. Calling the `GetEntryPoint` method on the users context returns the context specific entry point.

```
myPwrObj = myPwrCntxt.GetEntryPoint()
myPwrObj = myPwrCntxt.entrypoint       # Shortcut
#
# Returns:
#   pwr.Obj object or None
# This method raises a pwr.PwrError exception when something goes
   wrong.
# The possible exception errors are:
#   pwr.ReturnCode.FAILURE
```

Once the entry point object is obtained, it can be queried to get its Type, Name, Parent, and Children. These query methods return either an object or a `pwr.Grp` (in the case of `GetChildren()`), or either `None` or an empty `pwr.Grp` if the object(s) are non-existent. Not having a parent or any children is not considered an error condition. Note that the Python handling of non-existent parents and children is different than how these conditions are handled in the C API, where a non-zero int is returned. In Python returning an empty group or `None` enables code to handle hierarchy navigation more naturally then if an exception was to be raised. The `GetChildren()` method has a generator method, `GenerateChildren()`.

Method GetType

This method returns the `pwr.ObjType` of an object.

```
objType = myPwrObj.GetType()
objType = myPwrObj.objType    # Shortcut
#
# Returns:
#   pwr.ObjType or pwr.ObjType.INVALID upon failure
# This method raises a pwr.PwrError exception when something goes
    wrong.
# The possible exception errors are:
#   pwr.ReturnCode.FAILURE
```

Method GetName

This method returns the name of an object.

```
objName = myPwrObj.GetName()
objName = myPwrObj.name        # Shortcut
#
# Returns:
#   String containing myPwrObj's name
# This method raises a pwr.PwrError exception when something goes
    wrong,
#   such as if myPwrObj does not actually represent a pwr.Obj instance
    .
# The possible exception errors are:
#   pwr.ReturnCode.FAILURE
```

Method GetParent

Note that unlike the corresponding C API function on page 38 the Python `GetParent` Method returns `None` when the base object has no parent. This allows for handling this condition in Python without needing a `try/except` block.

```
objParent = myPwrObj.GetParent()
objParent = myPwrObj.parent       # Shortcut
#
# Returns:
#   pwr.Obj type or None if there is no parent.
# This method raises a pwr.PwrError exception when something goes
    wrong.
# The possible exception errors are:
#   pwr.ReturnCode.FAILURE
```

Method GetChildren and GenerateChildren

Note that unlike the corresponding C API function on page 39, the Python `GetChildren` method returns an empty group when the base object has no children. This allows for handling this condition in Python without needing a `try/except` block.

```
# Return a pwr.Grp of the children:
objChildrenGrp = myPwrObj.GetChildren()  # Returns a PwrGrp Group.
objChildrenGrp = myPwrObj.children       # Shortcut
# Generator of pwr.Obj children. Yields pwr.Obj members of the group.
for childPwrObj in myPwrObj.GenerateChildren():
    # Iterate on childPwrObj...
#
# Returns:
#   objChildrenGrp is a pwr.Grp containing pwr.Obj
#    type objects of children. An empty pwr.Grp may be returned
#    when there are no children.
# This method raises a pwr.PwrError exception when something goes
    wrong.
# The possible exception errors are:
#   pwr.ReturnCode.FAILURE
```

Method GetObjByName

A `pwr.Obj` object can be obtained using its name. Because the the naming system used for this method may be vendor-specific, this method is necessarily vendor implementation-specific and should not be considered generally portable. Vendor-specific details should be documented by the API implementor/vendor.

```
namedPwrObj = myPwrCntxt.GetObjByName(objName)
#
# Where:
#   objName is a Python string containing the power object's name
# Returns:
#   namedPowerObj is a pwr.Obj or None upon failure
# This method raises a pwr.PwrError exception when something goes
    wrong.
# The possible exception errors are:
#   pwr.ReturnCode.FAILURE
#   pwr.ReturnCode.NOT_IMPLEMENTED
```

Implementation Note: Object names are vendor-implementation-dependent and are not defined in this API. If the name of an object or group is not supported, a `pwr.PwrError` error code with `pwr.ReturnCode.NOT_IMPLEMENTED` is returned.

C.4.3 Group Methods

All Power API groups are associated with a context, therefore the group creation and retrieval methods are encapsulated as `pwr.Cntxt` class methods. See page 41 for the C API's full text description of Group operations.

Method GrpCreate

If a `pwr.PwrError` does not get raised during creation of this group, an empty `pwr.Grp` group is returned. No specific "Destroy" method is needed for any `pwr.Grp` groups. Python's garbage collection handles the clean up of `pwr.Grp` groups that are no longer referenced.

```
myPwrGrp = myPwrCntxt.GrpCreate()
#
# This method raises a pwr.PwrError exception when something goes
    wrong.
# The possible exception errors are:
#    pwr.ReturnCode.FAILURE
```

A Power API Group is an encapsulated Python list of pwr.Obj objects. This encapsulation offers strict type-checking over that of standard Python lists, but gives inheritance of all the power of Python lists to the `pwr.Grp`:

```
myGroup = myPwrCntxt.GrpCreate()
someOtherList = [1,2,3]
print isinstance(myGroup, list)           # Prints: "True"
print isinstance(myGroup, pwr.Grp)        # Prints: "True"
print isinstance(someOtherList, list)     # Prints: "True"
print isinstance(someOtherList, pwr.Grp)  # Prints: "False"
```

Method iter(Grp)

The following standard Python function generates an iterator over the objects in a `pwr.Grp`. See section C.2.1 on page 154 for more background on "generators":

```
for pwrObj in  iter(myPwrGrp):
    # Iterate on pwrObj...
#
# This method raises a pwr.PwrError exception when something goes
    wrong.
# The possible exception errors are:
#   pwr.ReturnCode.FAILURE
```

Method AddObj

This method adds a `pwr.Obj` to a group. As noted in the C API description on page 42 attempting to add an object that is already in a group is not allowed and will result in no insertion. The following shows examples of adding a `pwr.Obj` to a group.

```
myPwrGrp.AddObj(pwrObj)
myPwrGrp = myPwrGrp + pwrObj         # Shortcut
myPwrGrp = myPwrGrp + [pwrObj, ...]  # Shortcut
myPwrGrp += pwrObj                   # Shortcut
myPwrGrp += [pwrObj, ...]            # Shortcut
#
# Where:
#   pwrObj is a pwr.Obj object
# This method raises a pwr.PwrError exception when something goes
    wrong.
# The possible exception errors are:
#   pwr.ReturnCode.FAILURE
```

Method RemoveObj

This removes a `pwr.Obj` from the group.

```
myPwrGrp.RemoveObj(pwrObj)
myPwrGrp = myPwrGrp - pwrObj          # Shortcut
myPwrGrp = myPwrGrp - [pwrObj, ...]   # Shortcut
myPwrGrp -= pwrObj                    # Shortcut
myPwrGrp -= [pwrObj, ...]             # Shortcut
#
# Where:
#   pwrObj is a pwr.Obj object
# This method raises a pwr.PwrError exception when something goes
    wrong.
# The possible exception errors are:
#   pwr.ReturnCode.FAILURE
```

Method GetNumObjs

The following returns the number of objects in a group:

```
myPwrGrpNumObjs = myPwrGrp.GetNumObjs()
myPwrGrpNumObjs = len(myPwrGrp)          # Shortcut
#
# Returns:
#   myPwrGrpNumObjs is an integer
# This method raises a pwr.PwrError exception when something goes
    wrong.
# The possible exception errors are:
#   pwr.ReturnCode.FAILURE
```

Method GetObjByIndx NOT_IMPLEMENTED

In Python API implementations, there is no need for a Power API method to index a group's objects. Python's built-in list class, which forms the foundation of a `pwr.Grp` has all the necessary indexing and iteration methods needed. The C API's `PWR_GrpGetObjByIndx()` function, is documented in 5.3 on page 43.

```
# Python's built-in iterator
for pwrObj in iter(myPwrGrp):
    print pwrObj.GetName()

# Trick:  To index an item in a group:
pwrObj3 = list(iter(myPwrGrp))[3]
```

Method Duplicate

The following duplicates the `myPwrGrp` group creating the new `duplicateGrp`:

```
duplicateGrp = myPwrGrp.Duplicate()
duplicateGrp = pwr.Grp(myPwrGrp)     # Shortcut: copy constructor.
#
# This method raises a pwr.PwrError exception when something goes
    wrong.
# The possible exception errors are:
#   pwr.ReturnCode.FAILURE
```

Method Union and GenerateUnion

The following example creates a new group `unionGrp` containing all the objects that exist in either or both of the `myPwrGroup` and the `someOtherPwrGrp` group. The associated `GenerateUnion()` method is also shown:

```
unionGrp = myPwrGrp.Union(someOtherPwrGrp)
unionGrp = myPwrGrp | someOtherPwrGrp       # Shortcut
unionGrp |= someOtherPwrGrp                 # Shortcut
# Generator of pwr.Objs:
for pwrObj in myPwrGrp.GenerateUnion(someOtherPwrGrp):
    # Iterate on pwrObj...
#
# Where:
#   someOtherPwrGroup is a pwr.Grp object to merge with
# This method raises a pwr.PwrError exception when something goes
    wrong.
# The possible exception errors are:
#   pwr.ReturnCode.FAILURE
```

Method Intersection and GenerateIntersection

The following creates a new group containing only objects that exist in both the `myPwrGroup` and `someOtherPwrGrp` groups. The associated `GenerateIntersection()` method is also shown:

```
intersectionGrp = myPwrGrp.Intersection(someOtherPwrGrp)
intersectionGrp = myPwrGrp & someOtherPwrGrp                # Shortcut
intersectionGrp &= someOtherPwrGrp                          # Shortcut
# Generator of pwr.Objs:
for pwrObj in myPwrGrp.GenerateIntersection(someOtherPwrGrp):
    # Iterate on pwrObj...
#
# Where:
#   someOtherPwrGroup is a pwr.Grp object to merge with
# This method raises a pwr.PwrError exception when something goes
    wrong.
# The possible exception errors are:
#   pwr.ReturnCode.FAILURE
```

Method Difference and GenerateDifference

The following creates a new group containing all the objects of the current group or another but do not exist in both groups. The associated `GenerateDifference()` method is also shown:

```
differenceGrp = myPwrGrp.Difference(someOtherPwrGrp)
differenceGrp = myPwrGrp - someOtherGrp                 # Shortcut
differenceGrp -= someOtherGrp                           # Shortcut
# Generator of pwr.Objs:
for pwrObj in myPwrGrp.GenerateDifference(someOtherPwrGrp):
    # Iterate on pwrObj...
#
# Where:
#   someOtherPwrGroup is a pwr.Grp object to merge with
# This method raises a pwr.PwrError exception when something goes
    wrong.
# The possible exception errors are:
#   pwr.ReturnCode.FAILURE
```

Method SymDifference

The following creates new group containing members in the current group or another but not members that are in both groups, that is, the symmetric difference of the current group and another. This can be implemented as the `Union()` minus the `Intersection()` of two groups.

```
symDifferenceGrp = myPwrGrp.SymDifference(someOtherPwrGrp)
symDifferenceGrp = myPwrGrp ^ someOtherGrp             # Shortcut
symDifferenceGrp ^= someOtherGrp                       # Shortcut

#
# Where:
#   someOtherPwrGroup is a pwr.Grp object to merge with
# This method raises a pwr.PwrError exception when something goes
    wrong.
# The possible exception errors are:
#   pwr.ReturnCode.FAILURE
#   pwr.ReturnCode.BAD_VALUE
```

Method GetGrpByName

For general details see section 5.3 on page 48. As noted in that description, valid group names are vendor-specific. Use of this function should be considered non-portable. Vendor-specific details should be documented by the API implementor/vendor. An example of getting a group by name follows:

```
groupName = "vendor_supported_group_name_string"
myPwrGrp  = myPwrCntxt.GetGrpByName(groupName)
#
# Where:
#   groupName: vendor specific string designating group name
# Returns:
#   myPwrGrp is a pwr.Grp object or None if none found
# This method raises a pwr.PwrError exception when something goes
    wrong.
# The possible exception errors are:
#   pwr.ReturnCode.FAILURE
```

C.4.4 Attribute Methods

Method pwr.Obj.AttrGetValue

The `pwr.Obj AttrGetValue` method returns a Python named tuple describing a measurement. A measurement is a `namedtuple` type from the `collections` standard Python library module. Its contents can best be described by listing the definition of the named tuple and providing an example of how to access its members:

```
# Definition of the named tuple used to contain a measurement:
InfoFromGet = collections.namedtuple("InfoFromGet",
                                     "attr value obj timestamp rc")
```

```
# To return a single measurement:
attrName = pwr.AttrName.TEMP
measInfo = myPwrObj.AttrGetValue(attrName)

# Access the results:
measurementAttr = measInfo.attr
measurementValue = measInfo.value
measurementPwrObj = measInfo.obj
measurementTime = measInfo.timestamp
measurementError = measInfo.rc
#
# When a general failure occurs, a pwr.PwrError exception is raised.
# The possible exception errors are:
#   pwr.ReturnCode.FAILURE
#   pwr.ReturnCode.BAD_VALUE
```

Method pwr.Obj.AttrSetValue

Similarly, there is an attribute "Set" method for the object, which is capable of setting the value of one or more attributes on that object. This method uses a named tuple similar to the one defined in C.4.4 on page 180 to feed a list of one or more attribute-value pairs to the attribute "Set" methods. A definition of this named tuple and an example of how to create it follow for providing an input to the Attribute "Set" methods:

```
# Definition of the named tuple used to contain a setting:
InfoForSet = collections.namedtuple("InfoForSet", "attr value")
```

Another named tuple definition is used to extract any error information that the attribute operation(s) may yield:

```
# Definition of the named tuple used to contain error information
ErrorFromSet = collections.namedtuple("ErrorFromSet", "attr obj rc")
```

In the below example, the Attribute "Set" method, along with the named tuples for setting it and error handling is shown:

```
# To set a single attribute and handle any error that may occur:
setting = InfoForSet(attr=pwr.AttrName.CSTATE, value=3)
# The for-loop will catch any possible ErrorFromSet named tuples
# that the Set operation may yield.
for setError in myPwrObj.AttrSetValue(setting):
    errorAttribute = setError.attr
    errorPwrObj = setError.obj
    errorReturnCode = setError.rc
    # Process error here...
#
# In case of general failures, the possible exception errors are:
#   pwr.ReturnCode.FAILURE
#   pwr.ReturnCode.BAD_VALUE
```

Method pwr.Obj.AttrGetValues

The `pwr.Obj AttrGetValues` method returns a list containing Python measurement named tuples. Returning a list keeps consistency with the `pwr.Grp` AttrGetValue() and `pwr.Grp` AttrGetValues() methods which return the results from multiple measurements or queries as items in a list. For each of the AttrGetValue(s) methods there is a generator method which returns a memory-efficient iterator as opposed to a Python list. Please refer to C.4.4 on page 180 for details.

```
# To return a measurement for each attribute in the list:
attrList = [pwr.AttrName.TEMP, pwr.AttrName.VOLTAGE]
measList = myPwrObj.AttrGetValues(attrList)
for measInfo in measList:
    # Access the results:
    measurementAttr = measInfo.attr
    measurementValue = measInfo.value
    measurementPwrObj = measInfo.obj
    measurementTime = measInfo.timestamp
    measurementError = measInfo.rc

# To iterate on the results yielded by the generator method:
for measInfo in myPwrObj.AttrGenerateValues(attrList):
    # Access the results:
    measurementAttr = measInfo.attr
    measurementValue = measInfo.value  # etc.
```

Method pwr.Obj.AttrSetValues

The `AttrSetValues` method sets the values for multiple attributes on a `pwr.Obj`, yielding any errors that may have occurred. Please refer to C.4.4 on page 181 for details.

```
# To set multiple attributes and handle any errors that may occur:
setting1 = InfoForSet(attr=pwr.AttrName.CSTATE, value=3)
setting2 = InfoForSet(attr=pwr.AttrName.PSTATE, value=2)
settingList = [setting1, setting2]
for setError in myPwrObj.AttrSetValues(settingList):
    errorAttribute = setError.attr
    errorPwrObj = setError.obj
    errorReturnCode = setError.rc
    # Process error here...
```

Method pwr.Obj.AttrIsValid

To determine the validity of an attribute on a particular `pwr.Obj` object:

```
pwrAttr = pwr.AttrName.ENERGY
attrGood = myPwrObj.AttrIsValid(pwrAttr)
attrGood = myPwrObj.ENERGY.isvalid       # Shortcut
#
# Where:
#   pwrAttr is a pwr.AttrName type
# Returns:
#   True or False
#
```

Method pwr.Grp.AttrGetValue

The `pwr.Grp` `AttrGetValue` method returns a list containing Python named tuples containing the resulting measurements of an attribute across a `pwr.Grp`. Returning a list keeps consistency with the `pwr.Obj` AttrGetValues() and `pwr.Grp` AttrGetValues() methods which return the results from multiple measurements or queries as items in a list. The InfoFromGet() named tuple is used in the same way as with the pwrObj AttrGetValue(s) methods for containing the "measurement" information. Please refer to C.4.4 on page 180 for details. For this method there is a generator method which returns a memory-efficient iterator as opposed to a Python list.

```
# Return measurements for the given attribute for all group members
measList = myPwrGrp.AttrGetValue(pwr.AttrName.TEMP)
for measInfo in measList:
    # Access the results:
    measurementAttr = measInfo.attr
    measurementValue = measInfo.value
    measurementPwrObj = measInfo.obj
    measurementTime = measInfo.timestamp
    measurementError = measInfo.rc

# To iterate on the results yielded by the generator method:
for measInfo in myPwrGrp.AttrGenerateValues(pwr.AttrName.TEMP):
    # Access the results:
    measurementAttr = measInfo.attr
    measurementValue = measInfo.value  # etc.
#
# When a failure occurs, a pwr.PwrError exception is raised.
# The possible exception errors are:
#   pwr.ReturnCode.FAILURE
#   pwr.ReturnCode.BAD_VALUE
```

Method pwr.Grp.AttrSetValue

The `pwr.Grp AttrSetValue` method sets the value of an attribute on all the `pwr.Obj` objects across a `pwr.Grp`. The named tuple definitions InfoForSet() and ErrorFromSet() are used in the same way as with the pwrObj AttrSetValue(s) methods for extraction and construction of the "settings" and error named tuples. Please refer to C.4.4 on page 180 and C.4.4 on page 181 for details.

```
# Set a single attribute for all objects in a group
# and handle any errors that may occur:
setting = pwr.InfoForSet(attr=pwr.AttrName.CSTATE, value=3)
for setError in myPwrGrp.AttrSetValue(setting):
    errorAttribute = setError.attr
    errorPwrObj = setError.obj
    errorReturnCode = setError.rc
    # Process error on particular object
#
# In case of general failures, the possible exception errors are:
#   pwr.ReturnCode.FAILURE
#   pwr.ReturnCode.BAD_VALUE
```

Method pwr.Grp.AttrGetValues

The `pwr.Grp.AttrGetValues()` method returns a list containing Python measurement named tuples. Returning a list maintains consistency with the `pwr.Obj.AttrGetValues()` and `pwr.Grp.AttrGetValue()` methods which return the results from multiple measurements or queries as items in a list. For each of the AttrGetValue(s) methods, there is a generator method which returns a memory-efficient iterator as opposed to a Python list:

```
# To return a list of measurements of a list of attributes across the
# pwr.Obj members of a pwr.Grp, and access the results:
attrList = [pwr.AttrName.TEMP, pwr.AttrName.POWER]
measList = myPwrGrp.AttrGetValues(attrList)
for measInfo in measList:
    # Access the results:
    measurementAttr = measInfo.attr
    measurementValue = measInfo.value
    measurementPwrObj = measInfo.obj
    measurementTime = measInfo.timestamp
    measurementError = measInfo.rc

# To iterate on the results yielded by the generator method:
for measInfo in myPwrGrp.AttrGenerateValues(attrList):
    # Access the results:
    measurementAttr = measInfo.attr
    measurementValue = measInfo.value  # etc.
```

Method pwr.Grp.AttrSetValues

This method sets values for multiple attributes on all the objects in a group.

```
# To set multiple attributes across all objects of a group
# and handle any errors that may occur:
setting1 = pwr.InfoForSet(attr=pwr.AttrName.CSTATE, value=3)
setting2 = pwr.InfoForSet(attr=pwr.AttrName.PSTATE, value=2)
settingList = [setting1, setting2]
for setError in myPwrGrp.AttrSetValues(settingList):
    errorAttribute = setError.attr
    errorPwrObj = setError.obj
    errorReturnCode = setError.rc
    # Process error on particular attr for particular object.
```

C.4.5 Metadata Methods

The C API metadata functions (see page 64) are represented in Python API implementations as class methods to the `pwr.Obj` object.

Method AttrGetMeta

This method returns a metadata value associated with a `pwr.Obj` attribute.

```
attrName = pwr.AttrName.TEMP
metaName = pwr.MetaName.MAX
metaValue = myPwrObj.AttrGetMeta(attrName, metaName)
metaValue = myPwrObj.TEMP.MAX    # Shortcut
#
# Where:
#   attrName is the pwr.AttrName attribute to get the meta info for
#   metaName is the pwr.MetaName meta information to set
# Returns:
#   metaValue: the meta information requested
# This method raises a pwr.PwrError exception when something goes
    wrong.
# The possible exception errors are:
#   pwr.ReturnCode.FAILURE
#   pwr.ReturnCode.NO_ATTRIB
#   pwr.ReturnCode.NO_META
```

Method AttrSetMeta

This method writes a metadata value to the `pwr.Obj`'s attribute's metadata.

```
attrName = pwr.AttrName.CSTATE
metaName = pwr.MetaName.SAMPLE_RATE
myPwrObj.AttrSetMeta(attrName, metaName, 100)
myPwrObj.CSTATE.SAMPLE_RATE = 100   # Shortcut
#
# Where:
#   attrName is the pwr.AttrName attribute to get the meta info for
#   metaName is the pwr.MetaName meta information to set
#   metaValue: the meta information to set
# This method raises a pwr.PwrError exception when something goes
    wrong.
# The possible exception errors are:
#   pwr.ReturnCode.FAILURE
#   pwr.ReturnCode.NO_ATTRIB
#   pwr.ReturnCode.NO_META
#   pwr.ReturnCode.READ_ONLY
#   pwr.ReturnCode.BAD_VALUE
```

Method GetMetaValueAtIndex

This method returns a two-item tuple with the metadata value and a string representation of that value.

```
attrName = pwr.AttrName.CSTATE
metaValue, metaString = myPwrObj.GetMetaValueAtIndex(attrName, 1)
metaValue, metaString = myPwrObj.CSTATE[1]   # Shortcut
#
# Where:
#   attrName is a pwr.AttrName type
#   index is the index of the meta data item
# Returns:
#   metaValue is the meta information requested
#   metaString is the string version of the meta information
# This method raises a pwr.PwrError exception when something goes
    wrong.
# The possible exception errors are:
#   pwr.ReturnCode.FAILURE
#   pwr.ReturnCode.NO_ATTRIB
#   pwr.ReturnCode.BAD_INDEX
```

C.4.6 Statistics Methods

Statistics are applied either to Python `pwr.Obj` or a `pwr.Grp` objects. Because of this, the various statistics methods are either encapsulated by the `pwr.Obj` or the `pwr.Grp` classes. See section 5.6 starting on page 70 for C API documentation on statistics.

Method pwr.Obj.GetStat

Return a named tuple describing the requested historic statistic. Refer to C.4.4 on page 180 for details of the `InfoFromGet()` named tuple to access the information returned. The C API equivalent of this method is documented in section 5.6 on page 71.

```
# To return a single historic statistic:
attrName = pwr.AttrName.POWER
attrStat = pwr.AttrStat.AVG
endTime = Time(time.time())                                  #
    current time.
timePeriod = timePeriod(start=(endTime-3600.0), end=endTime)  # one
    hour.
statInfo = myPwrObj.GetStat(attrName, attrStat, timePeriod)
# Where:
#     attrName is a pwr.AttrName attribute name
#     attrStat is the pwr.AttrStat statistic to gather
#     timePeriod is the desired time of the statistic
# To access the results:
statisticValue = statInfo.value
statisticTimePeriod = statInfo.timestamp
statisticErrorCode = statInfo.rc
#
# When a general failure occurs, a pwr.PwrError exception is raised.
# The possible exception errors are:
#   pwr.ReturnCode.FAILURE
#   pwr.ReturnCode.BAD_VALUE
```

Method pwr.Grp.GetStats

This method returns a list containing Python named tuples describing historic statistics across the objects of a `pwr.Grp`. The C API equivalent of this method is documented in section 5.6 on page 72.

```
# To return historic statistics over the objects of a pwr.Grp:
attrName = pwr.AttrName.POWER
attrStat = pwr.AttrStat.AVG
endTime = Time(time.time())                                      #
    current time.
timePeriod = timePeriod(start=(endTime-3600.0), end=endTime)  # one
    hour.
statList = myPwrGrp.GetStats(attrName, attrStat, timePeriod)
# Where:
#     attrName is a pwr.AttrName attribute name
#     attrStat is the pwr.AttrStat statistic to gather
#     timePeriod: is the desired TimePeriod of the statistic, or None
# To access the results:
for statInfo in statList:
    # Access the results:
    statisticValue = statInfo.value
    statisticPwrObj = statInfo.obj
    statisticTimePeriod = statInfo.timestamp
    statisticErrorCode = statInfo.rc
    # Process statistic...
```

Class Stat

A `pwr.Stat` instance provides real-time statistics functionality and may be associated with a `pwr.Obj` or `pwr.Grp` object.

Method pwr.Obj.CreateStat

This method creates a `pwr.Obj.Stat` object:

```
attrName  = pwr.AttrName.POWER
attrStat  = pwr.AttrStat.AVG
myPwrStat = myPwrObj.CreateStat(attrName, attrStat)
#
# Where:
#   attrName is the pwr.AttrName attribute to get the statistics for
#   attrAttrStat is a pwr.AttrStat object
# Returns:
#   myPwrStat : a pwr.Stat object
# This method raises a pwr.PwrError exception when something goes
    wrong.
# The possible exception errors are:
#   pwr.ReturnCode.FAILURE
```

Method pwr.Grp.CreateStat

This method creates a `pwr.Grp.Stat` object.

```
attrName = pwr.AttrName.TEMP
attrStat = pwr.AttrStat.MAX
myGrpPwrStat = myPwrGrp.CreateStat(attrName, attrStat)
#
# Where:
#   attrName is the pwr.AttrName attribute to get the statistics for
#   attrAttrStat is a pwr.AttrStat object
# Returns:
#   myGrpPwrStat: a pwr.Stat object
# This method raises a pwr.PwrError exception when something goes
    wrong.
# The possible exception errors are:
#   pwr.ReturnCode.FAILURE
```

Method pwr.Stat.Start

This method starts the collection of real-time statistics on either the `pwr.Obj.Stat` or `pwr.Grp.Stat` object:

```
myPwrStat.Start()  # Start gathering real-time stats
#
# This method raises a pwr.PwrError exception when something goes
    wrong.
# The possible exception errors are:
#   pwr.ReturnCode.FAILURE
```

Method pwr.Stat.Stop

This method stops the collection of real-time statistics on either the `pwr.Obj.Stat` or `pwr.Grp.Stat` object:

```
myPwrStat.Stop()  # Stop gathering real-time stats
#
# this method raises a pwr.PwrError exception when something goes
    wrong.
# The possible exception errors are:
#   pwr.ReturnCode.FAILURE
```

Method pwr.Stat.Clear

This method resets the collection of real-time statistics on either the `pwr.Obj.Stat` or `pwr.Grp.Stat` object:

```
myPwrStat.Clear()
#
# This method raises a pwr.PwrError exception when something goes
    wrong.
# The possible exception errors are:
#   pwr.ReturnCode.FAILURE
```

Method pwr.Stat.GetValue

This method returns a named tuple describing the requested real-time statistic. Refer to C.4.4 on page 180 for details of the `InfoFromGet()` named tuple to access the information returned. The C API equivalent of this method is documented in section 5.6 on page 76.

```
# To return a single real-time statistic:
myObjPwrStat = myPwrObj.CreateStat(pwr.AttrName.TEMP, pwr.AttrStat.MAX
    )
myObjPwrStat.Start()  # Start gathering real-time stats
# (Do something useful...)
myObjPwrStat.Stop()   # Stop gathering real-time stats
statInfo = myObjPwrStat.GetValue()
# To access the results:
statisticValue = statInfo.value
statisticTimePeriod = statInfo.timestamp
statisticErrorCode = statInfo.rc
#
# When a general failure occurs, a pwr.PwrError exception is raised.
# The possible exception errors are:
#   pwr.ReturnCode.FAILURE
#   pwr.ReturnCode.BAD_VALUE
```

Method pwr.Stat.GetValues

This method returns a list containing Python named tuples describing real-time statistics across the objects of the `pwr.Grp` referenced in this `pwr.Stat` object. The C API equivalent of this method is documented in section 5.6 on page 77.

```
# To return a real-time statistic across the objects of a pwr.Grp:
myGrpPwrStat = myPwrGrp.CreateStat(pwr.AttrName.TEMP, pwr.AttrStat.MAX
    )
myGrpPwrStat.Start()  # Start gathering real-time stats
# (Do something useful...)
myGrpPwrStat.Stop()  # Stop gathering real-time stats
# Collect the statistics:
statList = myGrpPwrStat.GetValues()
for statInfo in statList:
    # Access the results:
    statisticValue = statInfo.value
    statisticPwrObj = statInfo.obj
    statisticTimePeriod = statInfo.timestamp
    statisticErrorCode = statInfo.rc
    # Process statistic...
#
# When a general failure occurs, a pwr.PwrError exception is raised.
# The possible exception errors are:
#   pwr.ReturnCode.FAILURE
#   pwr.ReturnCode.BAD_VALUE
```

Method pwr.Stat.GetReduce

A reduction of a real-time statistic can be retrieved with the `pwr.Stat.GetReduce()` method. For a description of the reduction operation refer to the C API description of `PWR_StatGetReduce()` on page 78.

```
# Get a reduction of real-time attribute values
reduceOp = pwr.AttrStat.AVG
reduceInfo = myGrpPwrStat.GetReduce(reduceOp)
reduceValue = reduceInfo.value
reduceTimePeriod = reduceInfo.timestamp
reduceErrorCode = reduceInfo.rc
# Where:
#     reduceOp: AttrStat reduction operation to get
# Returns:
#     A named tuple of the attr, value, obj (group), timestamp, and rc
    .
# This method will raise a PwrError exception when something goes
    wrong.
# The possible exception errors are:
#     ReturnCode.FAILURE
```

Method pwr.Grp.GetReduce

A reduction of a historic statistic can be retrieved with the `pwr.Grp.GetReduce()` method. For a description of the reduction operation refer to the C API description of `PWR_GrpGetReduce()` on page 80.

```
# Get a reduction of historic attribute values
import time
attrName = pwr.AttrName.TEMP
attrStat = pwr.AttrStat.MAX
reduceOp = pwr.AttrStat.AVG
endTime = Time(time.time())                                  #
    current time.
timePeriod = timePeriod(start=(endTime-3600.0), end=endTime)  # 1 hour
     period
reduceInfo = myPwrGrp.GetReduce(attrName, attrStat, reduceOp,
    timePeriod)
reduceValue = reduceInfo.value
reduceTimePeriod = reduceInfo.timestamp
reduceErrorCode = reduceInfo.rc
# Where:
#     attrName: AttrName attribute for which to gather statistics
#     attrStat: AttrStat historic statistic to gather
#     reduceOp: AttrStat reduction operation over the objects in the
    group.
#     timePeriod: pwr.TimePeriod period to get statistic.
# Returns:
#     A named tuple of the attr, value, obj (group), timestamp, and rc
    .
# This method will raise a PwrError exception when something goes
    wrong.
# The possible exception errors are:
#     ReturnCode.FAILURE
```

Method pwr.Stat.Destroy NOT_IMPLEMENTED

There is no need for a "Destroy" method due to Python's garbage collection implementation.

C.4.7 Version Functions

The top-level Version functions are not associated with any object. They return an integer detailing a particular segment of the version of the API.

There also is an included `version` variable available to obtain a string version of the major/minor version number of the API

Method GetMajorVersion

```
majorVersion = pwr.GetMajorVersion()
majorVersion = pwr.majorVersion      # Shortcut
versionStr = pwr.version             # Gives "1.2" as example.
#
# This method raises a pwr.PwrError exception when something goes
    wrong.
# The possible exception errors are:
#   pwr.ReturnCode.FAILURE
```

Method GetMinorVersion

```
minorVersion = pwr.GetMinorVersion()
minorVersion = pwr.minorVersion      # Shortcut
#
# These methods raises a pwr.PwrError exception when something goes
    wrong.
# The possible exception errors are:
#   pwr.ReturnCode.FAILURE
```

C.4.8 Big List of Attributes

The list of attributes for the default context is the same as the C API section 5.8 starting on page 84, with the attributes enumerated as defined in C.3.5 on page 161.

C.4.9 Big List of Metadata

The list of metadata names for the default context is the same as in the C API section 5.9 on page 87, with the attributes enumerated as defined in C.3.6 on page 163.

C.5 High Level (Common) Methods

C.5.1 Report Methods

Method pwr.Cntxt.GetReportByID

Please see the C API specification (section 6.1 on page 91) for a verbose description of the `GetReportByID()` method. Also, please see C.3.9 on page 168 and/or vendor implementation specific documentation for valid IDs. To collect statistics for an `idStr` and `idType` combination, the `GetReportByID()` method may be used.

```
statValue, timePeriod = myPwrCntxt.GetReportByID(idStr, idType,
    attrName,
                                                  pwrAttrStat,
    pwrTimePeriod)
#
# Where:
#   idstr is a string ID for the report to be generated.
#   idType is a pwr.ID type used to interpret the idstr ID
#   attrName is a pwr.AttrName type
#   pwrAttrStat is a pwr.AttrStat object
#   pwrTimePeriod is a pwr.TimePeriod object
# Returns:
#   statValue is the requested statistic
#   timePeriod is the pwr.TimePeriod of the statistic
# This method raises a pwr.PwrError exception when something goes
    wrong.
# The possible exception errors are:
#   pwr.ReturnCode.FAILURE
#   pwr.ReturnCode.NOT_IMPLEMENTED
#
```

C.6 Interfaces

In general, this section defines various combinations of default attributes and the roles/interfaces that use them. Mostly, this information directly maps to the definitions in the C API specification. However, there are some methods described in this section that are discussed because of the differences between their C and Python implementations.

C.6.1 Operating System, Hardware Interface

These methods are related to `pwr.Obj` objects of type `pwr.ObjType.NODE`, so they are encapsulated in the `pwr.Obj` class.

Method StateTransitDelay

This method returns the transition delay (`pwr.Time`) given the callers startState and endState input parameters. See section C.3.10 on page 168 for details on the `pwr.OperState` class):

```
startState = myPwrObj.GetPerfState()
startState = myPwrObj.perfstate         # Shortcut
endState = pwr.OperState(pwr.SleepState.SHALLOW, pwr.PerfState.FASTEST
    )
latencyPowerTime = myPwrObj.StateTransitDelay(startState, endState)
#
# Where:
#   startState is a pwr.OperState state
#   endState   is a pwr.OperState state
# Returns:
#   latencyPowerTime is the pwr.Time latency
# This method raises a pwr.PwrError exception when something goes
    wrong.
# The possible exception errors are:
#   pwr.ReturnCode.FAILURE
```

Note that the example above uses the method `GetPerfState` documented on page 199.

C.6.2 Monitor and Control, Hardware Interface

(No new methods described here)

C.6.3 Application, Operating System Interface

Method AppTuningHint

This method supplies power hints to a power object, using the hints enumerated by the `pwr.RegionHint` and `pwr.RegionIntensity` enumerations.

```
pwrRegionHint = pwr.RegionHint.MEM_BOUND      # see enum RegionHint
pwrRegionIntensity = pwr.RegionIntensity.LOW  # see enum
    RegionIntensity
myPwrObj.AppTuningHint(pwrRegionHint, pwrRegionIntensity)
#
# Where:
#   pwrRegionHint is a pwr.RegionHint type
#   pwrRegionIntensity is a pwr.RegionIntensity level
# This method raises a pwr.PwrError exception when something goes
    wrong.
# The possible exception errors are:
#   pwr.ReturnCode.FAILURE
#   pwr.ReturnCode.NOT_IMPLEMENTED
```

Method SetSleepStateLimit

This method sets the sleep-state limit from the enumeration for a power object.

```
pwrSleepState = pwr.SleepState.NO  # see enum SleepState
myPwrObj.SetSleepStateLimit(pwrSleepState)
myPwrObj.sleepstate = pwrSleepState         # Shortcut
#
# Where:
#   pwrSleepState is a pwr.SleepState type
# This method raises a pwr.PwrError exception when something goes
    wrong.
# The possible exception errors are:
#   pwr.ReturnCode.FAILURE
#   pwr.ReturnCode.NOT_IMPLEMENTED
```

Method WakeUpLatency

This method gets the wake up latency for the sleep-state (`DEEPEST` in this example) to transition from, returning the (`pwr.Time`) latency of the transition.

```
latencyPwrTime = myPwrObj.WakeUpLatency(pwr.SleepState.DEEPEST)
#
# Where:
#   pwrSleepState is a pwr.SleepState type
# Returns:
#   latencyPwrTime a pwr.Time object representing the latency time
# This method raises a pwr.PwrError exception when something goes
    wrong.
# The possible exception errors are:
#   pwr.ReturnCode.FAILURE
#   pwr.ReturnCode.NOT_IMPLEMENTED
```

Method RecommendSleepState

This method recommends a sleep-state for a power object, returning the deepest sleep-state (`pwr.SleepState`) to be used as a limit.

```
pwrSleepState = myPwrObj.RecommendSleepState(latencyPwrTime)
#
# Where:
#   latencyPwrTime is a pwr.Time latency value
# Returns:
#   pwrSleepState is a pwr.SleepState recommendation type
# This method raises a pwr.PwrError exception when something goes
    wrong.
# The possible exception errors are:
#   pwr.ReturnCode.FAILURE
#   pwr.ReturnCode.NOT_IMPLEMENTED
```

Method SetPerfState

This method requests that a power object performance level be set to a desired (`FASTEST` in this example) `pwr.PerfState` level.

```
myPwrObj.SetPerfState(pwr.PerfState.FASTEST)
myPwrObj.perfstate = pwr.PerfState.FASTEST     # Shortcut
#
# Where:
#   pwrPerfState is the requested pwr.PerfState type
# This method raises a pwr.PwrError exception when something goes
    wrong.
# The possible exception errors are:
#   pwr.ReturnCode.FAILURE
#   pwr.ReturnCode.NOT_IMPLEMENTED
```

Method GetPerfState

This method retrieves the performance level of a power object, returning the current performance state, `pwr.PerfState` object.

```
pwrPerfState = myPwrObj.GetPerfState()
pwrPerfState = myPwrObj.perfstate       # Shortcut
#
# Returns:
#   pwrPerfState is the returned pwr.PerfState type
# This method raises a pwr.PwrError exception when something goes
    wrong.
# The possible exception errors are:
#   pwr.ReturnCode.FAILURE
#   pwr.ReturnCode.NOT_IMPLEMENTED
```

Method GetSleepState

Similarly for retrieving the sleep-state of a power object, the `GetSleepState` method may be used.

```
pwrSleepState = myPwrObj.GetSleepState()
pwrSleepState = myPwrObj.sleepstate       # Shortcut
#
# Returns:
#   pwrSleepState is the returned pwr.SleepState type
# This method raises a pwr.PwrError exception when something goes
    wrong.
# The possible exception errors are:
#   pwr.ReturnCode.FAILURE
#   pwr.ReturnCode.NOT_IMPLEMENTED
```

C.6.4 User, Resource Manager Interface

(No new methods described here)

C.6.5 Resource Manager, Operating System Interface

(No new methods described here)

C.6.6 Resource Manager, Monitor and Control Interface

(No new methods described here)

C.6.7 Administrator, Monitor and Control Interface

(No new methods described here)

C.6.8 HPCS Manager, Resource Manager Interface

(No new methods described here)

C.6.9 Accounting, Monitor and Control Interface

(No new methods described here)

C.6.10 User Monitor and Control Interface

(No new methods described here)

C.7 Conclusion

This concludes the Python-specific appendix to the Power API specification.

Chapter 9

Index

Index