Service Availability™ Forum Hardware Platform Interface

Specification SAI-HPI-B.01.01

The Service Availability™ solution is high-availability and more; it is the delivery of ultra-dependable communication services on demand and without interruption.

The Service Availability™ Forum Hardware Platform Interface Specification may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Service Availability™ Forum Specification License Agreement

The Service Availability™ Forum Specification (the "Specification") found at the URL http://www.saforum.org (the "Site") is generally made available by the Service Availability Forum (the "Licensor") for use in developing products that are compatible with the standards provided in the Specification. The terms and conditions which govern the use of the Specification are set forth in this agreement (this "Agreement").

IMPORTANT –the following are excerpts of the terms and conditions provided in the agreement. please see the Licensor Site for a complete version of the agreement.

LICENSE GRANT. Subject to the terms and conditions of this Agreement, Licensor hereby grants you a non-exclusive, worldwide, non-transferable, revocable, fully-paid and royalty free license to:

a. reproduce copies of the Specification to the extent necessary to study and understand the Specification and to create products that are compatible with the Specification;
b. distribute copies of the Specification to your fellow employees who are working on a project or product development for which this Specification is useful; and
c. distribute portions of the Specification as part of your own documentation for a product you have built which complies with the Specification.

DISTRIBUTION. If you are distributing any portion of the Specification in accordance with Section 1(c), your documentation must clearly and conspicuously include the following statements:

a. Title to and ownership of the Specification (and any portion thereof) remain with Licensor.
b. The Specification is provided "As Is." Licensor makes no warranties, including any implied warranties, regarding the Specification (and any portion thereof) by Licensor.
c. Licensor shall not be liable for any direct, consequential, special, or indirect damages (including, without limitation, lost profits) arising from or relating to the Specification (or any portion thereof).
d. The terms and conditions for use of the Specification are provided on the Licensor Site.

RESTRICTION. Except as expressly permitted under License Grant, you may not (a) modify, adapt, alter, translate, or create derivative works of the Specification, (b) combine the Specification (or any portion thereof) with another document, (c) sublicense, lease, rent, loan, distribute, or otherwise transfer the Specification to any third party, or (d) copy the Specification for any purpose.

NO OTHER LICENSE. Except as expressly set forth in this Agreement, no license or right is granted to you, by implication, estoppel, or otherwise, under any patents, copyrights, trade secrets, or other intellectual property by virtue of your entering into this Agreement, downloading the Specification, using the Specification, or building products complying with the Specification.

OWNERSHIP OF SPECIFICATION AND COPYRIGHTS. The Specification and all worldwide copyrights therein are the exclusive property of Licensor. You may not remove, obscure, or alter any copyright or other proprietary rights notices that are in or on the copy of the Specification you download. You must reproduce all such notices on all copies of the Specification you make. Licensor may make changes to the Specification, or to items referenced therein, at any time without notice. Licensor is not obligated to support or update the Specification.

WARRANTY DISCLAIMER. THE SPECIFICATION IS PROVIDED "AS IS." LICENSOR DISCLAIMS ALL WARRANTIES, WHETHER EXPRESS, IMPLIED, OR STATUTORY, INCLUDING ANY WARRANTY OF MERCHANTABILITY, NONINFRINGEMENT OF THIRD-PARTY RIGHTS, FITNESS FOR ANY PARTICULAR PURPOSE, OR title. Without limiting the generality of the foregoing, nothing in this Agreement will be construed as giving rise to a warranty or representation by Licensor that implementation of the Specification will not infringe the intellectual property rights of others.

LIMITATION OF LIABILITY. To the maximum extent allowed under applicable law, LICENSOR DISCLAIMS ALL LIABILITY AND DAMAGES, INCLUDING DIRECT, INDIRECT, CONSEQUENTIAL, SPECIAL, AND INCIDENTAL DAMAGES, ARISING FROM OR RELATING TO THIS AGREEMENT, THE USE OF THE SPECIFICATION OR ANY PRODUCT MANUFACTURED IN ACCORDANCE WITH THE SPECIFICATION, WHETHER BASED ON CONTRACT, ESTOPPEL, TORT, negligence, STRICT LIABILITY, OR OTHER THEORY. NOTWITHSTANDING ANYTHING TO THE CONTRARY, LICENSOR'S TOTAL LIABILITY TO YOU ARISING FROM OR RELATING TO THIS AGREEMENT OR THE USE OF THE SPECIFICATION OR ANY PRODUCT MANUFACTURED IN ACCORDANCE WITH THE SPECIFICATION WILL NOT EXCEED ONE HUNDRED DOLLARS ($100). YOU UNDERSTAND AND AGREE THAT LICENSOR IS PROVIDING THE SPECIFICATION TO YOU AT NO CHARGE AND, ACCORDINGLY, THIS LIMITATION OF LICENSOR'S LIABILITY IS FAIR, REASONABLE, AND AN ESSENTIAL TERM OF THIS AGREEMENT.

TERMINATION OF THIS AGREEMENT. LICENSOR MAY TERMINATE THIS AGREEMENT, EFFECTIVE IMMEDIATELY UPON WRITTEN NOTICE TO YOU, IF YOU COMMIT A MATERIAL BREACH OF THIS AGREEMENT AND DO NOT CURE THE BREACH WITHIN TEN (10) DAYS AFTER RECEIVING WRITTEN NOTICE THEREOF FROM LICENSOR. UPON TERMINATION, YOU WILL IMMEDIATELY CEASE ALL USE OF THE SPECIFICATION AND, AT LICENSOR'S OPTION, DESTROY OR RETURN TO LICENSOR ALL COPIES OF THE SPECIFICATION AND CERTIFY IN WRITING THAT ALL COPIES OF THE SPECIFICATION HAVE BEEN RETURNED OR DESTROYED. PARTS OF THE SPECIFICATION THAT ARE INCLUDED IN YOUR PRODUCT DOCUMENTATION PURSUANT TO SECTION 1 PRIOR TO THE TERMINATION DATE WILL BE EXEMPT FROM THIS RETURN OR DESTRUCTION REQUIREMENT.

ASSIGNMENT. You may not assign, delegate, or otherwise transfer any right or obligation under this Agreement to any third party without the prior written consent of Licensor. Any purported assignment, delegation, or transfer without such consent will be null and void.

Contents

Figures

Tables

Revision History

Revision Number	Revision Date	Changes
Service Availability™ Forum Submission Draft	4/24/02	Original RFP Release.
UCMIv1.0	5/31/02	Initial Release.
SAI-HPI-A.00.80	8/2/02	Draft for Service Availability™ Forum comment.
SAI-HPI-A.00.99	8/29/02	Draft for final review and vote.
SAI-HPI-A.00.995	9/18/02	Final draft version.
SAI-HPI-A.01.01	9/25/02	Published Final Version.
SAI-HPI-B.00.86	2/2/04	Draft for Service Availability™ Forum comment.
SAI-HPI-B.00.88	2/18/04	Draft for final review and vote.
SAI-HPI-B.01.01	3/17/04	Published Final Version.

Terms and Definitions

Term	Definition
AdvancedTCA™	Advanced Telecommunications Computing Architecture.
DAT	Domain Alarm Table (qv).
Domain	A domain is a grouping of zero or more resources plus a set of associated services and capabilities.
Domain Alarm Table (DAT)	A table, provided by an HPI implementation, of fault conditions currently present in a domain.
Domain Reference Table (DRT)	A table, provided by an HPI implementation, of additional domains related to the current domain.
DRT	Domain Reference Table (qv).
Entity	An entity is a physical hardware component of the system.
False	The term “False” refers to the value of zero, also defined as the symbol SAHPI_FALSE.
Field Replaceable Unit (FRU)	A FRU is an entity that may be removed from or added to the system, while the system is live. A FRU may follow either the full hot swap model or the simplified hot swap model.
FRU	Field Replaceable Unit (qv).
Hardware Platform Interface (HPI)	The HPI is the open, industry standard interface between middleware or other application software and the hardware platform management infrastructure, for systems serving higher levels of Service Availability.
HPI	Hardware Platform Interface (qv).
HPI User	Within this specification, the term “HPI User” is used to indicate any software unit that is making use of the HPI, such as an operating system, high-availability middleware, application software, etc.
IDR	Inventory Data Repository.
IPMI	Intelligent Platform Management Interface.
Management Instrument	A “Management Instrument” is a sensor, a control, a watchdog timer, an inventory data repository, or an annunciator.

Term	Definition
Process	A process is created by the operating system and contains information about program resources and program execution state. A process can have multiple threads, which execute within the same address space. **Note:** While this term is based on POSIX terminology, it is not meant to limit HPI implementations entirely to POSIX operating systems. The intent of HPI is to remain operating system neutral. Operating systems that support different terminology or concepts should make the appropriate translations, as necessary.
RDR	Resource Data Record – a record that defines the management instruments (sensors, controls, watchdog timers, inventory data repositories, or annunciators) associated with a resource.
Resource	A Resource is the logical representation of the platform management capabilities of one or more entities that share a common management accessibility. The HPI provides accessibility to platform management capabilities by making "resources" accessible through the interface.
Resource Presence Table (RPT)	A table, provided by an HPI implementation, of all resources currently present in a domain.
RPT	Resource Presence Table (qv).
SAF	Service Availability™ Forum.
Session	A session is the context of a series of accesses by an HPI User to a domain.
Thread	A thread exists within a process and uses the process resources, but has its own independent flow of control as long as the parent process exists and the operating system supports it.
True	The term “True” refers to any non-zero value. While the symbol SAHPI_TRUE is defined, any non-zero value is considered “True”, even if not equal to SAHPI_TRUE.

1 Document Introduction

1.1 Document Overview

This document is organized into the following sections:

- Chapter 1, “Document Introduction” – This chapter provides an overview of how this document is organized.
- Chapter 2, “Hardware Platform Interface Overview” – This chapter provides a basic overview of the HPI specification. The overview includes a brief discussion of market and architectural contexts.
- Chapter 3, “The HPI Model” – This chapter introduces the key concepts upon which HPI is based.
- Chapter 4, “API Conventions” – This chapter provides conventions for API use, as well as a detailed description of HPI return code usage.
- Chapter 5, “General Functions” – This chapter describes general functions, such as version checking.
- Chapter 6, “Domain Functions” – This chapter describes functions that apply to entire domains. Discovery, session management, eventing, and Domain Event Logs are also discussed.
- Chapter 7, “Resource Functions” – This chapter contains information about functions that apply to individual resources. Functions are defined that:
  - provide detailed information about individual resources
  - manage hot swap of individual resources
  - discover, monitor, and control a variety of management instruments associated with the resources
- Chapter 8, “Data Type Definitions” – This chapter provides basic HPI data type definitions.
- Appendix A, “Usage Descriptions” – This appendix contains detailed usage descriptions of various functions.

1.2 Summary of Changes in SAI-HPI-B.01.01

- Dramatically improved the usability of sensor representations by getting rid of RAW formats, consolidating many of the sensor structures, and reworking the sensor enables, based on usage models from HPI Users. This provides a significant simplification to the functions and data representations for sensors, while retaining all of the original flexibility.
- Updated the entire Inventory Data Repository concept, making it easier to use and adding the ability to support AdvancedTCA™-based systems.
- Bounded and clarified the notion of HPI domains, making it easier for new HPI Users and HPI Implementers to understand the intent of the authors with respect to domain modeling.
- Enumerated error behaviors and return codes for each API.
- Eliminated `saHpiInitialize()` and `saHpiFinalize()` functions. Implementations are expected to automatically handle initialize/finalize operations as needed when a session is first opened, or all sessions are closed.
- Removed the concept of Entity Schemas because the functionality provided in the A.01.01 specification did not meet any useful purposes.
- Provided support for reporting event queue overflows.
- Formalized the relationships between domains, identifying multiple domains as either "peers" or having a parent/child relationship. These relationships are defined in a new domain-based table, the Domain Reference Table.
- Added support for "automatic" or "manual" mode operations on controls.

- Provided support for reporting failed resources.
- Replaced the "Significant Asserted State" processing in event subscriptions with a new domain-based table, the Domain Alarm Table, to report current alarm conditions in the domain.
- Added a new management instrument type, the "Annunciator" to provide an abstract interface to annunciation hardware.
- A number of small clarifications and issues have also been resolved.

1.3 References

The following documents contain information that is relevant to this Specification:

- IPMI -- *Intelligent Platform Management Interface*, Version 1.5, document Revision 1.0, 2/21/2001; Intel Corporation, Hewlett Packard, NEC, Dell; http://developer.intel.com/design/servers/ipmi/spec.htm
- PICMG 2.1 -- *Compact PCI Hot Swap Specification*, PICMG 2.1, R1.0, August 3, 1998; PCI Industrial Computer Manufacturers Group
- PICMG 3.0 – AdvancedTCA™ Base Specification, PICMG 3.0, R 1.0, December 30, 2002; PCI Industrial Computer Manufacturers Group

1.4 How to Provide Feedback on this Specification

If you have a question or comment about this specification, you may submit feedback online at http://www.saforum.org/specification/feedback.

If you would like to sign up to receive information updates on the Forum or Specification you may register at http://www.saforum.org/mailinglist.

1.5 How to Join the Service Availability™ Forum

The Promoter Members of the Forum require that all organizations wishing to participate in the forum complete a membership application. Once completed, a representative of the Service Availability™ Forum will contact you to discuss your membership in the Group.

1.5.1 Membership Application

The Service Availability™ Forum Membership Application can be completed online at http://www.saforum.org/join/apply.

Information requests may be submitted online at http://www.saforum.org/about/contact_us. Information requests are generally responded to within three business days.

1.5.2 Member Companies

An active list of the Service Availability™ Forum member companies can be viewed online at http://www.saforum.org/about/companies.

1.5.3 Press Materials

The Service Availability™ Forum has available a variety of downloadable resource materials, including the Forum Press Kit, graphics, and press contact information. Visit this area often for the latest press releases from the Service Availability™ Forum and its member companies.

The Service Availability™ Forum Press Area is located at http://www.saforum.org/press.

2 Hardware Platform Interface Overview

2.1 Overview

The SAF Hardware Platform Interface (HPI) specifies a generic mechanism to monitor and control highly available systems. The ability to monitor and control these systems is provided through a consistent, platform independent set of programmatic interfaces. The HPI specification provides data structures and functional definitions that can be used to interact with manageable subsets of a platform or system.

The HPI allows applications and middleware ("HPI User") to access and manage hardware components via a standardized interface. Its primary goal is to allow for portability of HPI User code across a variety of hardware platforms.

2.2 Market Context

Traditionally, Telecommunications Equipment Manufacturers (TEMs) provided internally-developed, vertically-integrated solutions to achieve the goal of "five nines", or 99.999% system availability (about 5 minutes of downtime per year). Such systems are extremely expensive and difficult to develop, and they tend to rely on proprietary hardware or software configurations to achieve the required stability. However, with the evolution of communications networks toward multi-service, packet-switched data, the concept of "system availability" has broadened into an emphasis on "service availability". In this context, service availability is a customer-centric approach to meeting the same "five nines" demands of legacy telecommunications equipment, but in a platform-neutral, standards-compliant distributed computing environment. In this environment "the system" providing a critical service may actually be highly distributed and comprised of several individual, heterogeneous, cooperating platforms.

As a result of this shift towards highly available "services" (rather than "systems") TEMs must now produce equipment that provides more functionality, exhibits higher levels of availability, and allows much greater interoperability in heterogeneous environments than previous monolithic "systems". In addition, these devices must be developed and deployed in substantially less time.

These factors combined with difficult market conditions have driven the industry to out-source solution development and integration programs to an emerging industry for telecom building blocks. This new industry consists of hardware and software suppliers delivering various components or subsystems (building blocks) which must be easily integrated into a complete "platform", "system", "network element", or "solution". The resulting composite device must be compatible with a distributed system architecture that provides "five nines" of availability for critical services.

2.2.1 Building Block Integration and Portability

Regardless of the source, manufacturer, or integrator of these building blocks, each component of the system must be seamlessly compatible with all other components. Additionally, each building block, must present to the external environment a minimal collection of capabilities as well as a non-platform-specific means of interacting with these capabilities.

In short, each building block must:

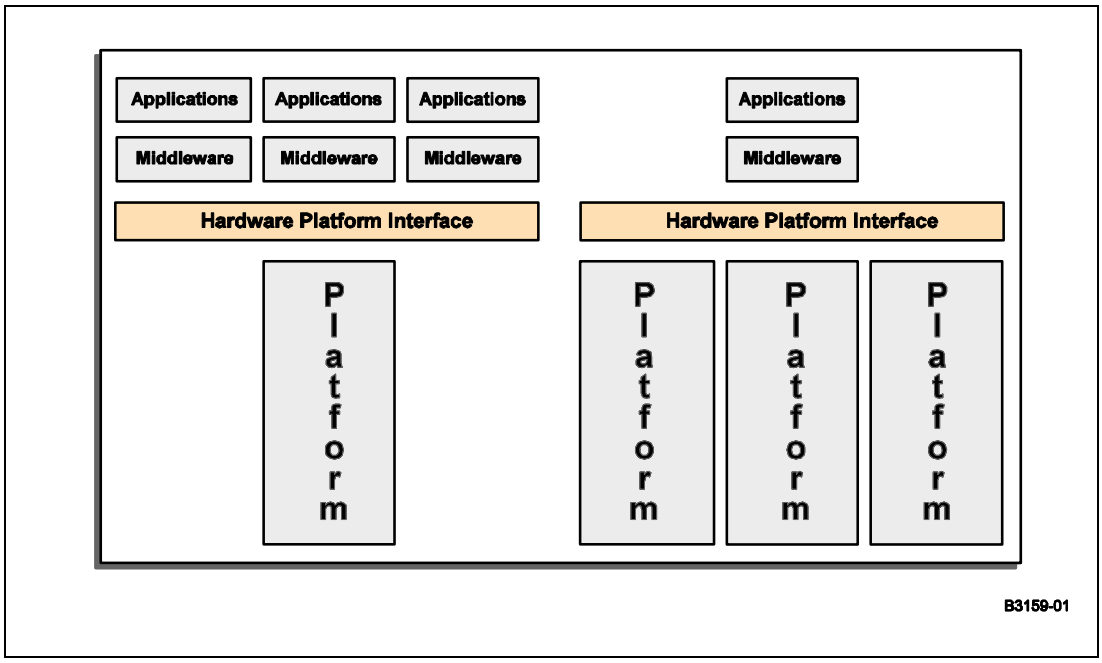
- Provide fault management capabilities, including facilities for status monitoring, fault detection and diagnosis, fault isolation, fault recovery, and component replacement.
- Communicate configuration, availability status, fault conditions, diagnostic test results, fault recovery actions, and related "internal state data" to HPI Users.

In addition to “local” or atomic fault management and communication capabilities, integrated collections of building blocks or platforms must:

- Be able to manage and recover from certain classes of faults without impact to the service being performed by the platform.
- Provide the ability to monitor and control collective system hardware and communicate aggregate platform “state data” to HPI Users.
- Provide the ability to test and validate individual integrated building blocks integrated into the platform to ensure that fault management actions and communications function properly and within the time constraints prescribed by the service or application.

The HPI specification addresses the interfaces between building blocks, subsystems, and aggregate platforms in order to simplify the integration of hardware and software building blocks into functional, highly available platforms for critical services. The portability of the HPI is shown in Figure 1.

Figure 1. Many to Many Portability



2.3 HPI’s Legacy in IPMI

The SAF HPI draws heavily on the concepts set forth by the Intelligent Platform Management Interface (IPMI) specification to define platform-independent capabilities and data formats. Thus, an implementation of the HPI interface on a platform that uses IPMI as a platform management infrastructure may be very straightforward. However, since HPI is a generic interface specification, it can be implemented on any platform with sufficient underlying platform management technology.

3 The HPI Model

In essence, the HPI model is comprised of four basic concepts -- ***Sessions, Domains, Resources and Entities*** – each of which is described briefly below, and in more detail in the following sections.

Starting at the basic foundation of the HPI model, ***entities*** represent the physical components of the system. Each entity has a unique identifier, called an entity path, which is defined by the component's location in the physical containment hierarchy of the system. An entity's manageability is modeled in HPI by management instruments, which are defined in resource data records associated with the entity. These management instruments are the mechanisms by which HPI Users can control and receive information about the state of the system. Entity management via the HPI may include any combination of the following functions:

- Reading values related to the operation or health of a component. This ability to read operational or health data is modeled via "Sensors" associated with the entity.
- Controlling aspects of the operation of a component. This ability to control a component is modeled via "Controls" associated with the entity, plus special functions to control the powering and resetting of a component.
- Reporting inventory and static configuration data. This data is reported via the "Inventory Data Repository" associated with the entity.
- Operating watchdog timers on components. Watchdog timers may cause implementation-defined actions to occur when the timers expire. The ability to operate watchdog timers is modeled via "Watchdog Timers" associated with the entity.
- Announcing status and fault condition information on a component. This is accomplished by using "Annunciators" associated with the entity.

Resources then provide management access to the entities within the system. Each resource is responsible for managing and presenting to the HPI User the entities that it has management control over. Additionally, resources may provide the following functions:

- Monitoring and controlling the insertion and removal of components in the system as it operates. This is reported through the interface as "Hot Swap" events and controlled via a set of "Hot Swap" functions.
- Storing a historical log of events from that resource for later retrieval. This storage and retrieval mechanism is modeled as a Resource Event Log contained in the resource.
- Updating management parameters, storing new parameters in non-volatile storage.

The HPI view of a system is divided into one or more ***domains***, where a domain provides access to some set of the resources within the system. A domain represents some part of the system that is capable of being managed by an HPI User; many systems may have a single domain, whereas systems that have areas dedicated to separate tasks, for example, may manage these through separate domains. Additionally, domains provide the following functions:

- Forwarding events generated by resources in the domain to HPI Users who have subscribed to receive domain events.
- Storing a historical log of events from the resources in the domain for later retrieval. This storage and retrieval mechanism is modeled as a Domain Event Log contained in the domain.
- Monitoring and controlling the insertion and removal of components in the system as it operates. This is reported through the interface as "Hot Swap" events, and reflected in a Resource Presence Table (RPT) accessible via the domain.
- Maintaining a table of current fault conditions in the domain.
- Maintaining a table of peer and/or tiered domains associated with the domain.

Sessions provide all access to an HPI implementation. An HPI session is opened on a single domain; one HPI User may have multiple sessions open at once, and there may be multiple sessions open on any given domain at once. It is intended that, in future releases, access control to the HPI will be performed at the session level; thus different sessions may have different access control. Sessions also provide access to events which occur in the domain accessed by the session.

The following sections describe all of these concepts in more detail.

3.1 Sessions

An HPI User accesses the system through ***sessions***, where each session is opened on a domain. A session provides access only to resources that are visible in the domain upon which the session is opened.

When an HPI User initiates an HPI session, a domain identifier must be provided, and a session identifier is returned. All subsequent API calls are passed the session identifier, and will access only resources that are visible in that domain. Each session open on the same domain accesses the same underlying data, such as entity states, Domain Event Logs, etc.; thus, a change made to this data in one session will affect all other sessions which are viewing the same data through HPI.

The set of resources that "belong" to a specific domain is implementation-specific and undefined by HPI.

Sessions also define the scope of events that are presented to an HPI User. Within a given session, an HPI User will be supplied with all events generated by resources in the associated domain.

One domain can have multiple sessions open on it (allowing for redundant access, for example); and any HPI User may have multiple sessions open at once. It should be noted, also, that resources may be visible in multiple domains. Therefore, there is no guarantee that a session provides exclusive access to any given resource. Synchronization issues are discussed in Section 3.6 on page 25.

Future versions of the HPI will incorporate security for the establishment and maintenance of sessions.

3.2 Domains

An HPI "System" is organized into one or more ***domains***. A domain is a grouping of zero or more resources, plus a set of associated services and capabilities; the latter are logically grouped into an abstraction called a ***domain controller***. While an HPI domain is a logical grouping of resources, it has no relationship with service groups defined in the Service Availability™ Forum's Application Interface Specification. HPI domains provide a mechanism for system implementers to group resources to define a hardware management view of a "System". While an HPI implementer can represent a physical enclosure, like a bladed server chassis, as an HPI domain, an HPI domain does not imply a physical grouping of resources or any physical meaning.

HPI domains can also reference other HPI domains allowing "Systems" composed of resources spread among multiple domains. Because of the flexible domain structure of HPI, many possible domain architectures may be constructed.

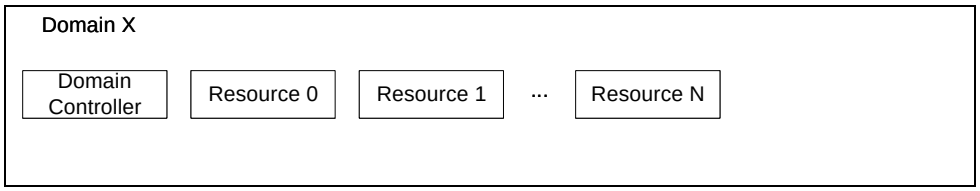
An HPI domain also defines a namespace. For example, a *ResourceId* must be unique within a domain, but the same *ResourceId* may identify a different resource in a different domain.

Domains provide a powerful feature, allowing an HPI User to discover the makeup of a "System" without any prior knowledge. By reading tables contained in the domains, HPI Users can discover all the managed entities in a "system".

Regardless of how domains are architected by a specific HPI implementation, to HPI Users, all domains have a common structure. Figure 2 shows an example domain structure. A domain consists of a domain controller, and it may contain zero or more resources, modeling management access points and/or field-replaceable units.

There are no requirements for how resources are mapped to domains; the same resource may be accessed through more than one domain, as described in the peer domain architecture in Section 3.2.2 on page 19. However, all resources accessible through the HPI must be included in at least one domain.

Figure 2. Example Domain Structure



3.2.1 Domain Controller

An HPI domain includes an abstraction called a domain controller, which provides a centralized set of services for the domain including:

- Domain Reference Table
- Resource Presence Table
- Event Management Service
- Domain Event Log
- Alarm Management Service

These services are provided at the domain level and a single instance of each service is present in every domain.

Domain Reference Table

The ***domain controller*** includes a Domain Reference Table (DRT) which provides information about other domains associated with the domain.

The DRT contains an entry for each associated domain, and HPI Users may read these entries to discover the presence of additional domains within a "System". The discovered domains can, in turn, be used to discover additional resources and domains. The DRT is automatically built and maintained by the HPI implementation. Domain entries in the DRT may change over time if the "System" configuration changes. The domain controller generates an event when a domain is added to the DRT (`SAHPI_DOMAIN_REF_ADDED`) and when a domain is removed from the DRT (`SAHPI_DOMAIN_REF_REMOVED`).

Resource Presence Table

The ***domain controller*** includes a Resource Presence Table (RPT) which provides information about the resources contained in the domain.

The RPT contains an entry for each resource currently present in the domain, and HPI Users may read these entries. The resources can, in turn, be used to discover which manageable entities are present. The RPT is automatically built and maintained by the HPI implementation. Resource entries will be dynamically added to or removed from the RPT as Field Replaceable Units (FRUs) are physically added to or removed from a platform.

Each resource currently accessible in a domain must be represented by a record in the RPT of that domain. If a resource is contained in multiple domains, it will be recorded in the RPT of each of the respective domains.

Event Management Service

The ***domain controller*** includes an event management service. The domain event management service is based on a limited "publish/subscribe" model consisting of a single event channel with a single publisher, the domain, and multiple subscribers, the sessions. The domain event management service collects events and distributes those events to the Domain Event Log and to sessions that have subscribed for events. If the HPI implementation presents multiple domains, the domain controller in each domain maintains a separate event management service. Similarly, each domain controller will have a separate Domain Event Log that will only contain events collected by that domain controller. Upon event subscription, a session will receive events from the domain controller for the domain on which that session is open. Figure 3 shows HPI event management flow.

HPI events are used to announce state changes within the domain. Events can originate from the domain controller, a contained resource, or a user application. Each event includes a severity, which represents its significance. Along with the severity, HPI events include the following information:

- Event Source
- Event Type
- Event Timestamp

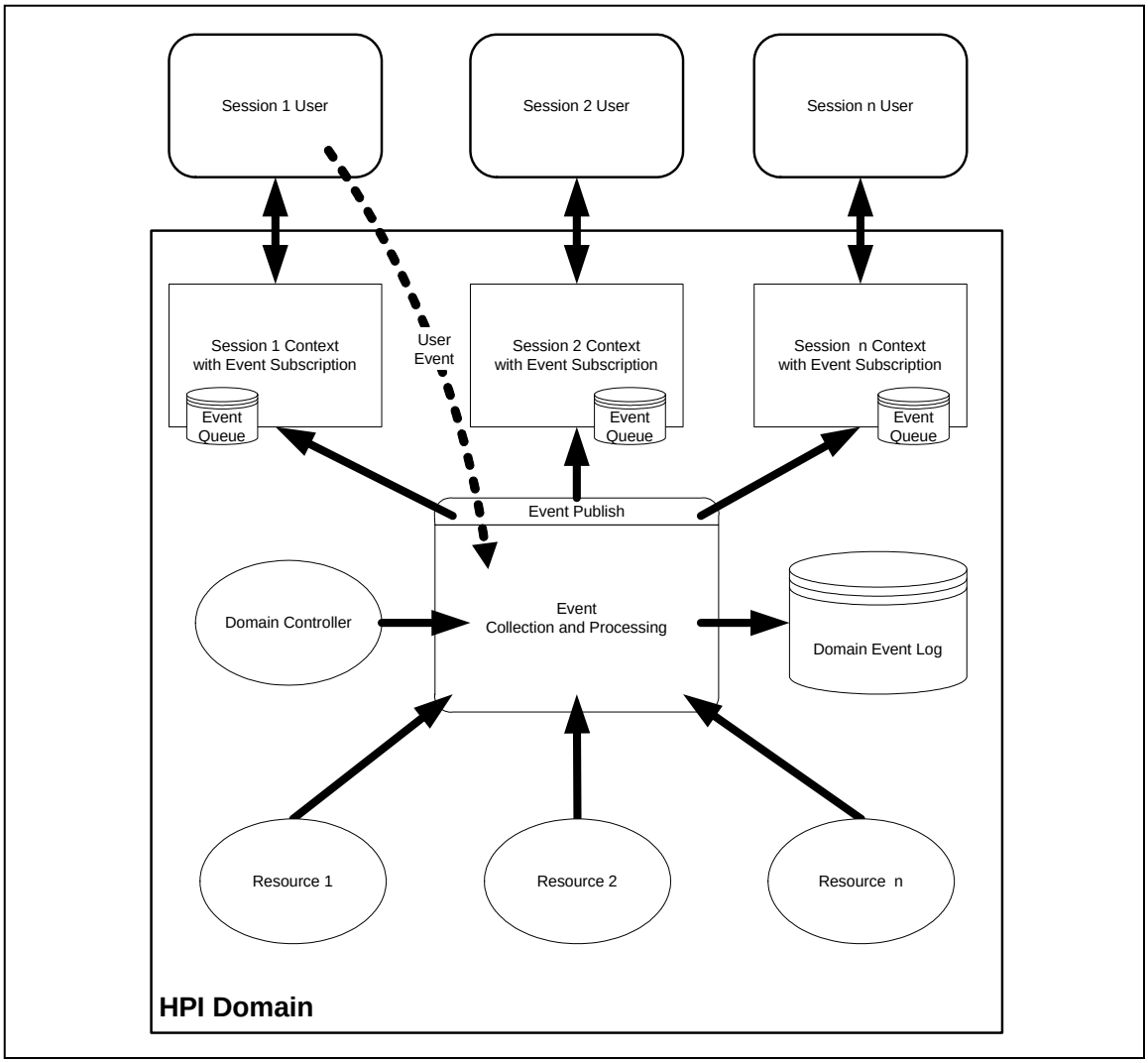
The Event Source identifies the resource from which the event originated. A *ResourceId* of `SAHPI_UNSPECIFIED_RESOURCE_ID` indicates that the event originated from the domain controller or an HPI User.

The Event Type describes what kind of event occurred. Valid event types include:

- Sensor (Event state assertion or de-assertion)
- Sensor event enable change
- Resource (Change in operational state of a resource)
- Domain (Addition or removal of a reference)
- Watchdog (Watchdog timer expiration)
- Hot swap (Resource hot swap state change)
- HPI Software Event (Audit discrepancies)
- OEM (Custom event data created by an HPI implementation)
- User (Custom event data created by an HPI User program)

The Event Timestamp indicates the time at which the event occurred. The originating source is not required to provide a timestamp for the event, and when the timestamp is not provided, the receiving domain controller will provide the event timestamp. If the timestamp is provided by the originating source, it may not be synchronized with the domain controller's timestamp.

Figure 3. HPI Event Management Service



Domain Event Log

The ***domain controller*** maintains a Domain Event Log for events collected in that domain. Events are stored in the Domain Event Log in the order in which they were received by the domain controller. A timestamp is added to each event before it is logged in the Domain Event Log. Exactly what events are placed in the Domain Event Log is implementation-specific.

The Domain Event Log may be managed by an HPI User through the set of HPI functions described in Section 6.4 on page 46. Management of the Domain Event Log includes activities such as reading records from it, writing records to it, clearing it, setting the timestamp clock, etc.

Alarm Management Service

The ***domain controller*** provides an alarm management service.

An ***alarm*** indicates the presence of a fault condition within a domain. Alarms are reported in a table maintained by the domain controller called the Domain Alarm Table (DAT). As a fault condition is detected, an HPI implementation creates an entry in the DAT. When the fault condition clears, the implementation deletes the corresponding entry from the DAT. Alarm entries in the DAT have an associated severity that represents the severity of the fault condition.

HPI Users can track and manage alarms using the set of API functions described in Section 6.6 on page 65. The DAT provides a central location for determining the presence of fault conditions within the domain.

In HPI, a fault condition reflected by an entry in the DAT is defined as one of the following:

- A “Significant” Asserted Sensor State
- A “Significant” Resource Failure
- A Platform-specific OEM Defined Condition
- An HPI User Defined Condition

More details on these conditions are included in Section 6.6 on page 65.

HPI implementations should use the contents of a Domain Alarm Table to control the annunciation of alarms on a platform. Alarm annunciation by a particular HPI implementation is implementation-specific. The implementation may directly control annunciation hardware as a result of changes to the DAT, or it may utilize HPI control or annunciator management instruments in various resources to announce the alarms. It is implementation-specific what annunciation actions are taken when alarms are added, removed, or acknowledged in the Domain Alarm Table.

A second capability is also included in HPI to provide a finer level of control over annunciation: the Annunciator management instrument. This capability is described in Section 7.6 on page 120.

3.2.2 Domain Architectures

An HPI domain provides a grouping of resources and/or references to other domains. The ***domain controller*** includes a table describing resources contained within the domain (the RPT) and a table describing other domains referenced by the domain (the DRT). These tables define the domain architecture.

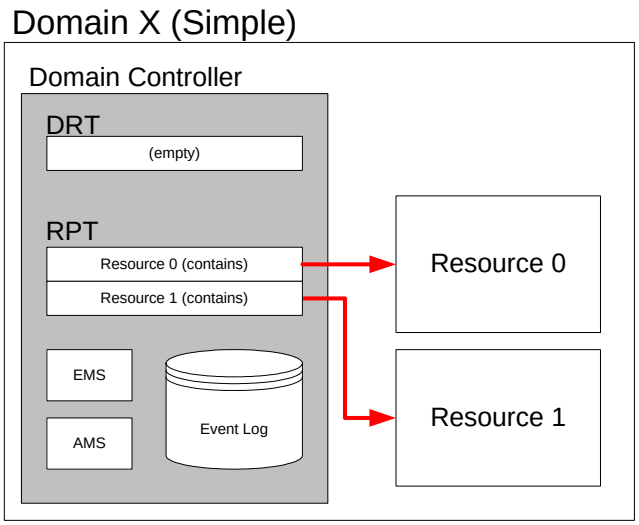
The “contains” versus “references” concept is important to understand. Since a domain ***contains*** a resource, access to that resource is conducted through the domain. A session must be opened on the domain before the resource can be accessed. Since a domain ***references*** another domain, access to the referenced domain is not conducted through the referencing domain. An HPI User must open up another session on the referenced domain, before the domain controller or resources within the referenced domain can be accessed.

The ability to contain resources and reference other domains allows for flexible domain architectures. Using the RPT and DRT, three types of domain architectures can be defined for an HPI implementation. Creating domain architectures for large complex “systems” can be accomplished by combining the different domain architectures described below.

Simple

A simple domain architecture consists of a single domain that contains only resources. Only the RPT is populated in this model. The DRT remains empty since the simple domain does not reference any additional domains. A conceptual view of the Simple Domain Architecture is shown in Figure 4.

Figure 4. Simple Domain Architecture

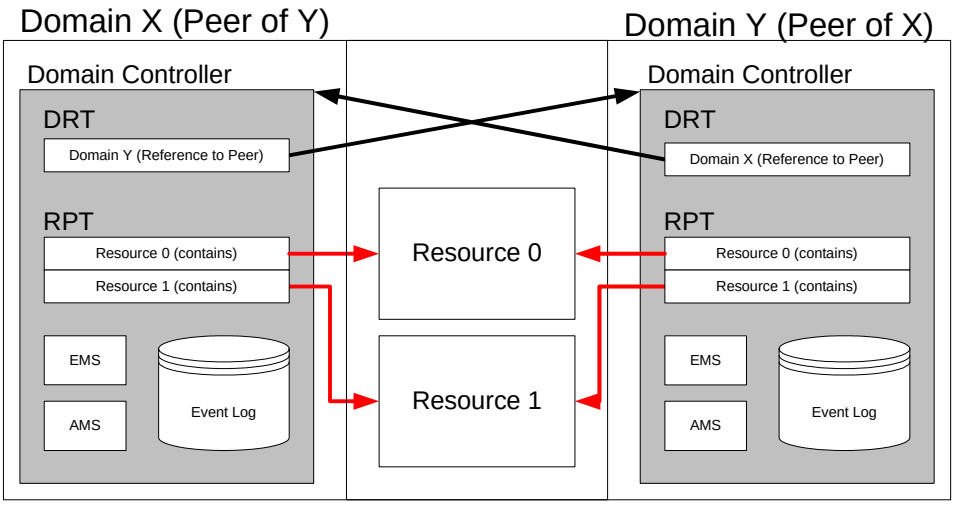


Peer

A peer domain architecture consists of two or more domains that are expected to contain the same resources and domain references. Each domain in a peer relationship contains an RPT listing all resources present in each domain. A flag in the DRT entry indicates that a domain reference is a ***peer*** domain. A ***peer*** domain reference indicates that the domain referenced by the DRT entry is expected to contain the same resources in its RPT and the same domain references in its DRT with one exception. While a peer domain will include DRT entries for its peers, it will not contain a DRT entry for itself; a peer domain would contain a reference to the current domain as its peer.

A conceptual view of the Peer Domain Architecture is shown in Figure 5.

Figure 5. Peer Domain Architecture



In this architecture, the HPI implementation represents independent domains that are expected to contain the same resources. An HPI User should treat peer domains that do not contain the same resources as a fault condition. Ordering of resources in the peer RPTs may be different and should not be treated as a fault condition. Likewise, an HPI User should treat peer domains that do not contain the same domain references, excluding a reference to itself, as a fault condition. Because peer domains represent independent and active domains, the domains will publish the same events. An HPI User should treat peer domains that do not generate the same events as a fault condition. The order in which events are received may also differ and should not be considered a fault condition.

The peer domain architecture is one way that redundancy may be modeled in an HPI implementation. This architecture is used to describe a "system" containing two access points for hardware management. This architecture does not imply that redundancy aspects, such as synchronization, active/standby status, and fail-over, are handled by the HPI implementation. An HPI User is responsible for keeping the domains in sync, thus setting a resource tag, or changing the severity of a resource in a domain does not automatically make the same changes in peer domains.

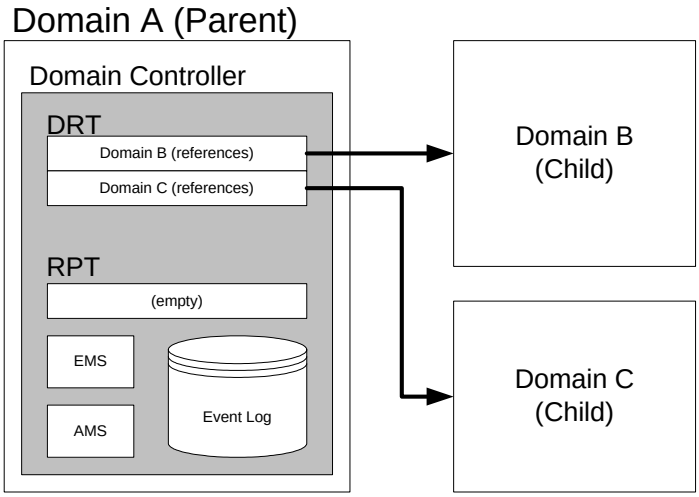
Alternatively, an HPI implementation may provide redundant, fault-tolerant access to resources within the implementation itself. In this case the implementation would not have peer domains. Rather, a single domain would be presented that contained the resources for which redundant access is provided. The redundancy aspects, such as synchronization, active/standby status, and fail-over, would then be handled transparently by the HPI implementation.

Tiered

A tiered domain architecture is used when all the resources in a "system" cannot be accessed in a single domain. A "parent" domain references one or more "child" domains in its DRT. The "child" domains may be simple domains containing only resources, or they may themselves be a "parent" to other "child" domains. Domains in a tiered relationship are disjointed and do not contain the same resources.

A conceptual view of Tiered Domain Architectures is shown in Figure 6.

Figure 6. Tiered Domain Architecture



In order to work with tiered domains, multiple sessions will need to be opened. Initially, a session to the parent domain is opened. Then sessions may be opened on all domains that are found within the "parent" domain. This continues until the corresponding DRTs do not contain any further "child" domain references. It is not legal for a "child" domain to contain a domain reference back to its "parent" or any higher level "parent".

Resources may exist in both "parent" and "child" domains in this architecture. In other words, resources are the leaf-nodes of a domain hierarchy tree.

The tiered domain architecture gives the implementer great flexibility in architecting a wide variety of solutions. They may be used to manage a variety of actions with complex subsystems of hardware resources.

3.2.3 Domain Identifier

An HPI implementation assigns domain identifiers to HPI domains. Each domain identifier is unique within the "system" and may be used by an HPI User to open a session with a particular domain. If an HPI User does not know a particular domain identifier, `SAHPI_UNSPECIFIED_DOMAIN_ID` can be specified, and the implementation will select a default domain. Which domain is accessed when `SAHPI_UNSPECIFIED_DOMAIN_ID` is used is implementation-specific, and may be different for different HPI Users.

When a domain is accessed with `SAHPI_UNSPECIFIED_DOMAIN_ID`, the actual domain identifier associated with the domain may be obtained from the *DomainInfo* structure, in the `saHpiDomainInfoGet()` function.

3.3 Resources

Resources represent the management access to the components of the system. Each resource provides access to information about some of the components of the system, called ***entities***.

The set of entities that are manageable through a resource is implementation-dependent. However, because including or excluding resources in domains controls accessibility to entity management, it is expected that entities that are somehow bound together for the purposes of platform management will be collected into resources by the implementation. An important example of this is a set of entities that are on a single FRU. As a FRU is inserted into or removed from the system, it will become manageable or unmanageable, respectively. Because of this, the HPI models FRU insertion or removal via the inclusion or exclusion of resources in domains. Thus, each FRU must be modeled as a separate resource, and all entities contained on that FRU would generally be associated with that resource.

Another example of shared management accessibility would be entities that are managed via a shared management controller, such as an alarm card. If a single alarm card is responsible for managing several different system entities (fans, power supplies, etc.), then those entities may be best associated with a single resource. If the alarm card fails, this will be reported as the resource becoming unavailable, which will then indicate to an HPI User which entities are no longer accessible via the HPI interface.

Generally, all sensors, controls, etc., associated with a single entity are associated with a single resource. There is one case, however, where this may not be appropriate. If a management controller in the system (e.g., an alarm card) is responsible for managing entities on other FRUs (e.g., fans on a non-intelligent removable fantray), there are two different occurrences which can impact the manageability of those entities. If a FRU containing managed entities (i.e., fantray) is removed from the system, those entities become unmanageable. But, also, if the managing FRU (i.e., alarm card) fails, then all entities it manages become unmanageable. To correctly model this, each of these "remotely managed" FRU entities needs to be associated with two different resources: one associated with its own local FRU (i.e., fantray), and one associated with the managing FRU (i.e., alarm card). An acceptable method to reflect this situation in the HPI is to create a resource for the managed entity's FRU (i.e., fantray) which contains no sensors, controls, etc., but provides only the "hot swap" capability for the FRU. Each of the sensors, controls, etc., for the FRU would be included in the RDR table for the resource associated with the managing FRU (i.e., alarm card). When a management instrument (sensor, control, etc.) located in one resource is associated with a different FRU in this way, the *IsFru* flag is set in the Resource Data Record for that management instrument to indicate that fact.

Appendix A provides a detailed example describing this case.

The RPT entry for a resource includes a flag that indicates what capabilities the resource supports. Resource capabilities include inventory data reporting, sensors, controls, etc. for entities managed via that resource. Resources report information on their detailed capabilities and configuration using Resource Data Records (RDRs). Each resource which contains any management instruments (sensors, controls, watchdog timers, inventory data repositories, and annunciators) will also contain an ***RDR repository***, which contains all of the RDRs relating to the management instruments hosted by that resource.

A resource that represents a FRU may additionally support managed hot swap; see Section 7.7 on page 132. This capability provides a flexible and powerful way to manage hot insertion and extraction of FRUs. The HPI also provides functions for power management, reset control, etc.

3.4 Entities

Each manageable component of the system is identified as a unique ***entity*** in the system. Every entity is uniquely named by an ***entity path*** (described below) that identifies the component in terms of its physical containment within the system.

There is no single element of the HPI that represents the entity. Rather, each entity is modeled as a collection of Resource Data Records (RDRs), which contain information about the management instruments associated with the entity. Every RDR contains the entity path of the entity to which it relates; hence, an HPI User can determine the type of a given entity (from the leaf element of the entity path), and the set of RDRs that contain all information about the entity, by reading all of the RDRs in the system, and correlating RDRs with the same entity path.

Every entity exposed by the HPI implementation must have at least one RDR associated with it. To expose an entity that has no management instruments an HPI implementation may supply an inventory data repository RDR for that entity (which will contain the entity path of the entity), even if the set of inventory data returned for it is empty.

3.4.1 Entity Paths

An ***entity path*** consists of an ordered set of {Entity Type, Entity Location} pairs. The path defines the physical location of the entity in the system, in terms of which entity it is contained within, and the entity that its container is contained in, etc. The path is ordered from the entity itself, to the "root" of the system hierarchy; thus, the first {Entity Type, Entity Location} pair in the set identifies the specific physical entity identified by the entity path. For example, if the managed entity is a power supply, the Entity Type will be set to the enumerated value for "Power Supply" and the Entity Location will contain a number that identifies *which* power supply, and its location within the containing entity. In order to determine the actual physical location of the entity, the system provider must provide a mapping of Entity Location Numbers to physical positions in the system. Subsequent {Entity Type, Entity Location} pairs identify a series of containing entities that further specify which physical entity in the overall system is being addressed.

For example, consider a system that contains multiple racks, each of which contain multiple subracks, with each subrack containing power supplies that serve that subrack. A full Entity Path for an individual power supply may be:

Entity Type	Enumerated Value	Entity Location
Power Supply	SAHPI_ENT_POWER_SUPPLY	2
Subrack	SAHPI_ENT_SUBRACK	4
Rack	SAHPI_ENT_RACK	3
<Root>	SAHPI_ENT_ROOT	-

This Entity Path would identify "Power supply at location 2, contained in Subrack at location 4, contained in Rack at location 3" and the entity path would look like:

{{SAHPI_ENT_POWER_SUPPLY, 2}, {SAHPI_ENT_SUBRACK, 4}, {SAHPI_ENT_RACK, 3}, {SAHPI_ENT_ROOT, 0}}.

As described above, its entity path uniquely identifies every manageable entity in the system. Every RDR representing a particular management instrument will contain the entity path of the specific physical device with which that management instrument is associated. Each entity in a system must have a unique entity path.

Every resource in the system also has an entity path associated with it, stored in its RPT entry. This entity path defines the physical management access device which the resource models. In the case of a resource that models a FRU, the entity path stored in the resource's RPT entry identifies the physical FRU modeled by that resource.

In the case where entities, and the resources that manage them, are visible in multiple domains, the HPI implementation must always use the same entity path for each entity and resource across all domains. HPI Users may therefore use these entity paths to correlate the views presented in different domains, and determine which entities are shared between the domains, and which are not.

In many platforms there's a notion of a blade that resides in a slot. Such situations can be represented with an entity path that has the blade entity contained in the slot entity, as shown:

Entity Type	Enumerated Value	Entity Location
Blade	SAHPI_ENT_SBC_BLADE	1
Physical Slot	SAHPI_ENT_PHYSICAL_SLOT	4
AdvancedTCA™ Chassis	SAHPI_ENT_ADVANCEDTCA_CHASSIS	3
<Root>	SAHPI_ENT_ROOT	-

This would indicate a Single Board Computer blade in slot 4 of an AdvancedTCA™ chassis at position 3 and the entity path would look like:

{{SAHPI_ENT_SBC_BLADE, 1}, {SAHPI_ENT_PHYSICAL_SLOT, 4}, {SAHPI_ENT_ADVANCEDTCA_CHASSIS, 3}, {SAHPI_ENT_ROOT 0}}.

3.5 Discovery

A key characteristic of the HPI is that it permits management software to dynamically discover what manageable components are present in a system.

Since domains are the largest aggregation of resources that can be addressed, the first task in discovering the content of a system is to determine the domains that are present in the system. There is no defined way to discover what domains represent -- whole chassis, multiple chassis, parts of chassis, etc.; however, every HPI implementation will provide access to a default domain when `SAHPI_UNSPECIFIED_DOMAIN_ID` is used as the domain identifier. This allows the HPI User to open an initial session to the HPI, and serves as the starting point for a full discovery procedure. This domain may contain references to additional domains, allowing them to be discovered.

The discovery process will typically proceed in a number of steps, as follows:

1) Open a session to a domain; initially, the HPI User can use the domain identifier, `SAHPI_UNSPECIFIED_DOMAIN_ID`, to begin the discovery process.
2) Read the RPT for the default domain.
3) For each resource in the RPT, extract the capability flags for that resource.
4) Read the RDR repository for the resource to find the RDRs for all entities managed by the resource; use their entity paths to determine which RDRs refer to the same entities.
5) Read the DRT for the domain.
6) For each domain in the DRT, repeat the same process for this domain starting at item 1 using the domain identifier for the domain found in the DRT entry. This will allow all domains to be discovered. If the DRT entry references a peer domain, the resources in the peer domain must match the resources in the domain referencing the peer. If the DRT entry references a child domain, the resources in the child domain should be treated as new resources.

Throughout the discovery process, an HPI User should make sure that no resources or domain references were added or removed. This can be accomplished by using RPT and DRT update counters and events.

While reading RPT and RDR data, an HPI User may build a physical model of the system using the Entity Paths contained in each RPT entry or RDR. This would allow an HPI User to determine the true set of entities that exist in the system, by correlating multiple RDRs for each entity, and by correlating the same entities visible through multiple domains.

3.6 Synchronization

An HPI implementation is defined to be a single -- whether centralized or distributed -- model of the domains and resources in the system. An HPI implementation must allow multiple sessions to be opened to it simultaneously; this shall include multiple sessions open to a single domain, and (if the HPI implementation presents multiple domains) sessions open onto multiple domains. It is therefore possible that multiple HPI Users will be able to view and operate on any given resource simultaneously; either due to:

- a resource being mapped into more than one domain; or
- multiple open sessions on the same domain.

There may be multiple HPI implementations present in a system, such as those offered by different vendors. HPI Users should not assume any synchronization between different HPI implementations.

3.6.1 Synchronization Responsibilities

It is the responsibility of an HPI implementation to ensure that a single, consistent view of the system and its domains and resources is presented to all HPI Users. In the face of multiple concurrent changes, the HPI implementation should attempt to make updates visible system-wide in a timely manner; however, no specific timing is specified.

An HPI implementation shall guarantee that each HPI operation on any resource is atomic; that is, if two writes are attempted to a resource (e.g. from different sessions), one write shall succeed entirely, and then the other write shall succeed entirely. The order in which the writes occur may be undefined, depending on timing and the locations of the sources of the writes.

Any one HPI implementation is required to report all events for all resources to all sessions which have subscribed to receive events and which have visibility of those resources.

3.6.2 Multiple HPI implementations

If multiple HPI implementations exist in a system, then they should only be used to manage non-overlapping entities. If a single entity is managed through two or more HPI implementations, then there is no guarantee of any consistent view of the entity state.

Any software layer using concurrent access via multiple HPI implementations should take appropriate care; for example, by updating both RDR tables, reading most current sensor values, etc., if it is possible that anything may have had an effect on the other HPI implementation.

3.7 Remote Access to the Platform Interface

In some scenarios, the HPI implementation(s) in a system may physically reside on the same entity (entities) which have physical access to the platform's controls and sensors. In many scenarios, however, this will not be the case. For example, there may be fixed-function "chassis management" modules, with access to the underlying controls and sensors, which are then accessed across a bus or network. If these modules are not capable of hosting the HPI implementation(s), then HPI will need to reside on some other entity and access the management modules across a bus or network connection.

It is the responsibility of an HPI implementation to handle this issue internally, and to present the interfaces described herein to its HPI Users. The HPI implementation should handle faults that occur while accessing remote entities; for example, it may be appropriate for an implementation to try multiple paths in the event of a network or bus failure. Faults discovered by the HPI implementation should be reported against the appropriate HPI resources.

3.8 Resource Failures

The HPI specification supports resources as independent modules. If a resource is not functional, this should not impact the availability of other resources in the HPI implementation. When a resource is associated with a FRU, if the FRU is not present in the system then the resource is not present, and therefore cannot be functional. Resources may also be non-functional, though, due to faults in the management infrastructure used by the HPI implementation. An infrastructure fault may result in the failure of any resource, whether it is associated with a FRU, or with a non-removable part of the system.

The way the HPI interface reports the failure of a resource depends on whether the resource is a FRU or a non-removable part of the system. And if it is a FRU, it further depends on whether the HPI implementation can detect the presence of the FRU when the resource is non-functional.

3.8.1 Failure of a Non-FRU Resource

For resources that are not associated with FRUs, a failure of the resource, or the inability to communicate with the resource is indicated by setting a *ResourceFailed* flag in the RPT entry for that resource indicating that it is not currently functional. Further attempts by an HPI User to access resources that are so flagged will fail with the error return `SA_ERR_HPI_NO_RESPONSE`. If a resource becomes functional, the *ResourceFailed* flag is cleared from its RPT entry, and access to the resource can proceed normally. However, when a failed resource becomes functional, it may be in a different state than when the resource failed. For example, sensor event states may have changed, watchdog timers may have expired, etc. An HPI User may thus need to query the resource to learn its current state after it has exited from a failed state.

There are three events associated with resource operational state:

- Resource Failed – A resource currently represented in the RPT has failed. This event is issued using the *ResourceSeverity* level defined in the RPT entry for the resource.
- Resource Restored – A resource marked as failed in the RPT is restored to functionality. This event is issued using the *ResourceSeverity* level defined in the RPT entry for the resource.
- Resource Added – A resource not present or not functional at system startup is added to the RPT. This event is issued at the severity level of `SAHPI_INFORMATIONAL`.

If a resource is not functional at the time an HPI implementation is started, the resource may not be detected, and thus not populated in the RPTs of domains that should contain the resource. It is not the function of the HPI implementation to insure that all resources that ought to be present actually are present, so this failure to detect a non-functional resource is acceptable. However, if the resource later becomes functional, it should then be added to the RPTs of domains that should contain the resource. If this occurs, a “Resource Added” event is generated indicating the *ResourceId* of the resource added to the RPT.

3.8.2 Failure of a FRU Resource

In many systems, the presence of a FRU cannot be detected if the resource associated with that FRU is not functional. In these systems, the failure of a resource associated with a FRU is indistinguishable by the management infrastructure from the unexpected removal of the FRU. Thus, the failure of the resource will be reported to HPI Users as a "surprise extraction" of the FRU (see Section 7.7.1 on page 135) and the resource will be removed from the RPT of any domain that contained the resource. Further attempts by an HPI User to access this resource will fail with the error return `SA_ERR_HPI_INVALID_RESOURCE` – just as they would fail if the FRU was actually removed from the system.

Similarly, in these systems, if a failed FRU resource becomes functional, it will be added to the appropriate RPTs and reported as a hot swap insertion event. Again, because the infrastructure cannot determine whether this was an actual hot swap event or a resource recovering from a failure condition, this will appear in most respects like any other hot swap insertion event, as though a new resource is being added to the system. The *ResourceId* assigned to the resource when it is added may be the same value that was previously used when the resource failed, but implementations may also assign a new *ResourceId*.

While hot swap events are used to communicate resource failures there is one potential difference from the normal hot swap sequence. When a resource recovers from a failure, it may not be in the INSERTION PENDING hot swap state. Thus, the hot swap event will indicate a transition from the NOT PRESENT state to whatever hot swap state is current for the resource. See Section 7.7 on page 132 for more information about hot swap events.

On the other hand, some systems can detect the presence or absence of a FRU even if the resource associated with the FRU is not functional. For example, presence detection for blades in a shelf may be done by a shelf-manager module that senses hardware signals independently from the failed blade-management module. If it is possible to detect that a FRU is present even when the resource associated with the FRU is not functional, then the HPI implementation should report the failure or recovery of the resource as described in Section 3.8.1 on page 26 for non-FRU resources. That is, it should set or reset the *ResourceFailed* flag in the RPT entry and issue events indicating a change of functional state for the resource.

However, the "Resource Added" event described in Section 3.8.1 on page 26 should never be used for a FRU resource. If a FRU resource is added to an RPT after HPI initialization, this should always be reported as a hot swap event, as described in Section 7.7 on page 132.

3.9 Implementation Requirements

All HPI implementations must be reentrant. A reentrant implementation allows multiple applications and multi-threaded applications to make simultaneous API calls without corruption of data or context.

4 API Conventions

4.1 Return Codes

Table 1 describes each of the HPI return codes.

The order of detection of return codes is undefined. Thus, if more than one error occurs, any one of the possible return codes may be returned.

Table 1. HPI Return Codes

Return Code	Definition
SA_OK	This code indicates that a command completed successfully.
SA_ERR_HPI_ERROR	An unspecified error occurred. This code should be returned only as a last resort; e.g. if the cause of an error cannot be determined.
SA_ERR_HPI_UNSUPPORTED_API	The HPI implementation does not support this API. Because there are no optional APIs, a compliant HPI implementation must not use this error return value. It is provided so that implementations, which are not fully compliant may indicate that a particular API has not been implemented. Note that while there are no optional APIs, there are a number of APIs, which are optionally supported on a resource-by-resource basis. Such support is indicated via the Resource Capabilities of resource's RPT entry. When an API is called for one of the unsupported Resource Capabilities, the proper error return code is SA_ERR_HPI_CAPABILITY. That remains the proper return even if the HPI implementation contains no resources that support a particular Resource Capability.
SA_ERR_HPI_BUSY	The command cannot be performed because the targeted device is busy.
SA_ERR_HPI_INTERNAL_ERROR	The HPI implementation has encountered an error.
SA_ERR_HPI_INVALID_CMD	The specific object to which a command was directed does not support that command (which was otherwise valid).
SA_ERR_HPI_TIMEOUT	The requested operation, which had a timeout value specified, timed out. For example, when reading input with a timeout value, if no input arrives within the timeout interval, this code should be returned. This should only be returned in cases where a timeout is anticipated as a valid consequence of the operation; if the addressed entity is not responding due to a fault, use SA_ERR_HPI_NO_RESPONSE.
SA_ERR_HPI_OUT_OF_SPACE	The requested command failed due to resource limits.
SA_ERR_HPI_OUT_OF_MEMORY	This code is returned if the HPI implementation does not have sufficient memory to complete the requested action.
SA_ERR_HPI_INVALID_PARAMS	One or more parameters to the command were invalid. This return code is used to indicate that an input parameter was invalid with respect to the specification (such as setting a text language in a SaHpiTextBufferT to a value not in the enumerated list.)
SA_ERR_HPI_INVALID_DATA	This return code is used when passing in data that is invalid for the configuration of the implementation (such as setting an analog control to a value that is out of range for that control.)
SA_ERR_HPI_NOT_PRESENT	The requested object was not present. For example, this code would be returned when attempting to access an entry in a RPT or RDR repository, which is not present. As another example, this code would also be returned when accessing an invalid management instrument on a valid resource.
SA_ERR_HPI_NO_RESPONSE	There was no response from the domain or object targeted by the command, due to some fault. This code indicates an un-anticipated failure to respond; compare with SA_ERR_HPI_TIMEOUT.
SA_ERR_HPI_DUPLICATE	Duplicate request -- such as attempting to subscribe to a session, which has already has an active subscription.

Return Code	Definition
SA_ERR_HPI_INVALID_SESSION	An invalid session identifier was specified in the command.
SA_ERR_HPI_INVALID_DOMAIN	Invalid domain identifier specified – i.e. a domain identifier, which does not correspond to any real domain was specified in the command.
SA_ERR_HPI_INVALID_RESOURCE	Invalid resource identifier specified – i.e. a resource identifier which does not correspond to a resource in the addressed domain was specified in the command.
SA_ERR_HPI_INVALID_REQUEST	The request is invalid in the current context. An example would be attempting to unsubscribe for events, when the session has not subscribed to receive events.
SA_ERR_HPI_ENTITY_NOT_PRESENT	The addressed management instrument is not active because the entity with which it is associated is not present. This condition could occur, for instance, when an alarm module is managing a fan tray FRU. The alarm module would contain management instruments (sensors, etc) for the fan tray. The fan tray may be removed, even though the management instruments are still represented in the alarm module. In this case, SA_ERR_HPI_ENTITY_NOT_PRESENT would be returned if a management instrument associated with a removed entity is accessed.
SA_ERR_HPI_READ_ONLY	The request cannot be completed, as the data to be operated upon is read-only.
SA_ERR_HPI_CAPABILITY	This request cannot be completed, because the specified resource does not support the required capability. This code is returned when the appropriate RPT capability flag is not set for the resource. (For example, this code would be returned if saHpiEventLogInfoGet() is called, when the SAHPI_CAPABILITY_EVENT_LOG is not set.)
SA_ERR_HPI_UNKNOWN	The HPI implementation cannot determine an appropriate response. Only used with saHpiResourceIdGet().

Note: Situations may occur where both `SA_ERR_HPI_INVALID_PARAMS` and `SA_ERR_HPI_INVALID_DATA` could apply to the same input parameters. In such cases, where both of the return codes are applicable, the specification adopts the convention that the more stringent `SA_ERR_HPI_INVALID_DATA` code be returned.

4.2 Generic Return Codes

The return codes listed in Table 2 are global to the majority of functions in the HPI specification. These conditions are to be applied in addition to those spelled out individually in each function. Anticipated uses for compliance testing are indicated in Table 2 as well.

Table 2. Generic Return Codes

Return Code	Condition
SA_ERR_HPI_BUSY	For situations where the underlying hardware is busy. It is unlikely that this condition will be tested in black-box compliance testing.
SA_ERR_HPI_ERROR	For conditions not covered by the return codes listed here or the return codes listed with the individual APIs. It is unlikely that this condition will be tested in black-box compliance testing.
SA_ERR_HPI_INTERNAL_ERROR	For all other abnormal internally generated situations not specifically covered here. It is unlikely that this condition will be tested in black-box compliance testing.
SA_ERR_HPI_INVALID_RESOURCE	For all functions for which ResourceId is an input parameter and the ResourceId passed in is invalid. Note that this condition is likely to be tested in black-box compliance testing.
SA_ERR_HPI_INVALID_SESSION	For all functions for which SessionId is an input parameter and the SessionId passed in is invalid. Note that this condition is likely to be tested in black-box compliance testing.
SA_ERR_HPI_NO_RESPONSE	For situations where the implementation fails to get a response from an HPI implementation (which may be remote from the library.)

Return Code	Condition
	It is unlikely that this condition will be tested in black-box compliance testing.
SA_ERR_HPI_OUT_OF_MEMORY	For situations where the implementation runs out of memory. It is unlikely that this condition will be tested in black-box compliance testing.
SA_ERR_HPI_UNSUPPORTED_API	For APIs that are not implemented in the current version. Compliance tests will fail, upon receipt of this return code.
SA_ERR_HPI_ENTITY_NOT_PRESENT	For situations where the resource is present, but the management instrument cannot be accessed. This situation occurs when the entity associated with the management instrument has been extracted, but the resource allowing access to the management instrument is still present. It is unlikely that this condition will be tested in black-box compliance testing.

4.3 Interface Behavior when a Function Returns an Error

Unless an exception is explicitly listed in the specification for a particular function, when any HPI function returns an error code other than `SA_OK`, no memory pointed to by any INOUT or OUT parameters will be changed. When the error code `SA_OK` is returned, the memory pointed to by INOUT and OUT parameters will be updated as defined for the function.

Also, unless an exception is explicitly listed in the specification for a particular function, when any HPI function returns an error code other than `SA_OK`, `SA_ERR_HPI_NO_RESPONSE`, `SA_ERR_HPI_INTERNAL_ERROR`, or `SA_ERR_HPI_ERROR`, the function call will have no effects on the managed system. When the error code `SA_OK` is returned, the function call will have the effects on the managed system that are defined for that function with the parameters that were passed. When an error code of `SA_ERR_HPI_NO_RESPONSE`, `SA_ERR_HPI_INTERNAL_ERROR`, or `SA_ERR_HPI_ERROR` is returned, the function call may have partially or fully completed its work prior to retuning the error code. Therefore, it is undefined what effects the function call may have on the managed system.

4.4 Pointer Conventions

When discussing pointers in the descriptive text, no de-referencing is done. For example, when the text describes "…*NextEntryId* will be set to the identifier for the next valid entry", and *NextEntryId* is defined as a pointer in the corresponding prototype, the reader should interpret this to be that the contents of the memory location pointed to by *NextEntryId* will be set to the identifier for the next valid entry.

5 General Functions

5.1 Implementation Version Checking

The HPI interface header file defines a symbol, `SAHPI_INTERFACE_VERSION` of type `SaHpiVersionT` that is equal to the specification version represented by that header file. A library function, `saHpiVersionGet()` will return a variable of type `SaHpiVersionT` that is equal to the specification version supported by the library. By comparing these two values, an HPI User may determine whether a compatible version of the library is being accessed.

5.1.1 saHpiVersionGet()

This function returns the version identifier of the SaHpi specification version supported by the HPI implementation.

Prototype

```
SaHpiVersionT SAHPI_API saHpiVersionGet ( void );
```

Parameters

None.

Return Value

The interface version identifier, of type `SaHpiVersionT` is returned.

Remarks

This function differs from all other interface functions in that it returns the version identifier rather than a standard return code. This is because the version itself is necessary in order for an HPI User to properly interpret subsequent API return codes. Thus, the `saHpiVersionGet()` function returns the interface version identifier unconditionally.

This function returns the value of the header file symbol `SAHPI_INTERFACE_VERSION` in the SaHpi.h header file used when the library was compiled. An HPI User may compare the returned value to the `SAHPI_INTERFACE_VERSION` symbol in the SaHpi.h header file used by the calling program to determine if the accessed library is compatible with the calling program.

6 Domain Functions

6.1 Session Management

This set of functions deals with opening and closing sessions for the HPI implementation. Sessions are managed on a domain-by-domain basis, and provide the appropriate scope for accessing resources and retrieving events.

When an HPI User initiates an HPI session, a domain identifier is provided and all subsequent function calls on that session will access resources in that domain exclusively. Limiting access on a session to resources contained in a single domain provides a capability to have multiple HPI Users with each having restricted management access. For example, a tenant in a multi-tenant system could be given management capabilities over its leased resources, but not to other tenant's resources since its domain and subsequent sessions do not provide access to resources outside of the specified domain. Which set of resources “belong” to a specific domain is implementation-specific and undefined by HPI.

Once a session is opened, it is associated with a single domain. Thus, subsequent function calls that reference that session (via a *SessionId* that is returned when the session is opened), implicitly address a specific domain. For example, a function call on a session to read an RPT entry would read an entry from the RPT for the domain associated with the session. In addition to identifying a domain, a session also may contain an event queue. Therefore, most function calls require the use of a *SessionId* to identify the session context for an HPI User.

The expected scope of a session identifier is a *process*. A session identifier should not be shared between *processes*, and such use is strongly discouraged. Session identifiers may be shared between *threads* contained within a single *process*. A single *process* may open multiple sessions on a given domain, thus obtaining multiple session identifiers.

6.1.1 saHpiSessionOpen()

This function opens an HPI session for a given domain and set of security characteristics (future).

Prototype

```
SaErrorT SAHPI_API saHpiSessionOpen (
    SAHPI_IN  SaHpiDomainIdT      DomainId,
    SAHPI_OUT SaHpiSessionIdT     *SessionId,
    SAHPI_IN  void                *SecurityParams
);
```

Parameters

DomainId – [in] Domain identifier of the domain to be accessed by the HPI User. A domain identifier of `SAHPI_UNSPECIFIED_DOMAIN_ID` requests that a session be opened to a default domain.

SessionId – [out] Pointer to a location to store an identifier for the newly opened session. This identifier is used for subsequent access to domain resources and events.

SecurityParams – [in] Pointer to security and permissions data structure. This parameter is reserved for future use, and must be set to NULL.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_INVALID_DOMAIN` is returned if no domain matching the specified domain identifier exists.

`SA_ERR_HPI_INVALID_PARAMS` is returned if:

- A non-null *SecurityParams* pointer is passed.
- The *SessionId* pointer is passed in as NULL.

`SA_ERR_HPI_OUT_OF_SPACE` is returned if no more sessions can be opened.

Remarks

None.

6.1.2 saHpiSessionClose()

This function closes an HPI session. After closing a session, the *SessionId* will no longer be valid.

Prototype

```
SaErrorT SAHPI_API saHpiSessionClose (
    SAHPI_IN SaHpiSessionIdT SessionId
);
```

Parameters

SessionId – [in] Session identifier previously obtained using `saHpiSessionOpen()`.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

Remarks

None.

6.1.3 saHpiDiscover()

This function requests the underlying management service to discover information about resources and associated domains.

This function may be called during operation to update the DRT table and the RPT table. An HPI implementation may exhibit latency between when hardware changes occur and when the domain DRT and RPT are updated. To overcome this latency, the `saHpiDiscover()` function may be called. When this function returns, the DRT and RPT should be updated to reflect the current system configuration and status.

Prototype

```
SaErrorT SAHPI_API saHpiDiscover (
    SAHPI_IN SaHpiSessionIdT SessionId
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

Remarks

None.

6.2 Domain Discovery

The HPI implementation maintains a list of associated domains in the "system" using the Domain Reference Table (DRT). An HPI User may discover associated domains by accessing the DRT for each domain in the "system". This table should represent up-to-date information on the other domains referenced by a given domain. The DRT may be accessed using the DRT functions described below.

The DRT contains an entry for each domain referenced by a given domain. Each entry describing another domain includes the domain identifier for that domain. If the domain reference is a peer reference, the DRT entry will be flagged as a peer domain, else the domain reference is a tiered domain reference.

6.2.1 saHpiDomainInfoGet()

This function is used for requesting information about the domain, the Domain Reference Table (DRT), the Resource Presence Table (RPT), and the Domain Alarm Table (DAT), such as table update counters and timestamps, and the unique domain identifier associated with the domain.

Prototype

```
SaErrorT SAHPI_API saHpiDomainInfoGet (
    SAHPI_IN  SaHpiSessionIdT      SessionId,
    SAHPI_OUT SaHpiDomainInfoT     *DomainInfo
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

DomainInfo – [out] Pointer to the information describing the domain and DRT.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_INVALID_PARAMS` is returned if the *DomainInfo* pointer is passed in as NULL.

Remarks

None.

6.2.2 saHpiDrtEntryGet()

This function retrieves domain information for the specified entry of the DRT. This function allows an HPI User to read the DRT entry-by-entry.

Prototype

```
SaErrorT SAHPI_API saHpiDrtEntryGet (
    SAHPI_IN  SaHpiSessionIdT     SessionId,
    SAHPI_IN  SaHpiEntryIdT       EntryId,
    SAHPI_OUT SaHpiEntryIdT       *NextEntryId,
    SAHPI_OUT SaHpiDrtEntryT      *DrtEntry
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

EntryId – [in] Identifier of the DRT entry to retrieve. Reserved *EntryId* values:

- `SAHPI_FIRST_ENTRY` Get first entry
- `SAHPI_LAST_ENTRY` Reserved as delimiter for end of list. Not a valid entry identifier.

NextEntryId – [out] Pointer to location to store the *EntryId* of next entry in DRT.

DrtEntry – [out] Pointer to the structure to hold the returned DRT entry.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_NOT_PRESENT` is returned if the:

- Entry identified by *EntryId* is not present.
- *EntryId* is `SAHPI_FIRST_ENTRY` and the DRT is empty.

`SA_ERR_HPI_INVALID_PARAMS` is returned if the:

- *DrtEntry* pointer is passed in as NULL.
- *NextEntryId* pointer is passed in as NULL.
- *EntryId* is an invalid reserved value such as `SAHPI_LAST_ENTRY`.

Remarks

If the *EntryId* parameter is set to `SAHPI_FIRST_ENTRY`, the first entry in the DRT will be returned. When an entry is successfully retrieved, *NextEntryId* will be set to the identifier of the next valid entry; however, when the last entry has been retrieved, *NextEntryId* will be set to `SAHPI_LAST_ENTRY`. To retrieve an entire list of entries, call this function first with an *EntryId* of `SAHPI_FIRST_ENTRY` and then use the returned *NextEntryId* in the next call. Proceed until the *NextEntryId* returned is `SAHPI_LAST_ENTRY`.

If an HPI User has not subscribed to receive events and a DRT entry is added while the DRT is being read, that new entry may be missed. The update counter provides a means for insuring that no domains are missed when stepping through the DRT. In order to use this feature, an HPI User should call `saHpiDomainInfoGet()` to get the update counter value before retrieving the first DRT entry. After reading the last entry, the HPI User should again call `saHpiDomainInfoGet()` to get the update counter value. If the update counter has not been incremented, no new entries have been added.

6.2.3 saHpiDomainTagSet()

This function allows an HPI User to set a descriptive tag for a particular domain. The domain tag is an informational value that supplies an HPI User with naming information for the domain.

Prototype

```
SaErrorT SAHPI_API saHpiDomainTagSet (
    SAHPI_IN  SaHpiSessionIdT      SessionId,
    SAHPI_IN  SaHpiTextBufferT     *DomainTag
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

DomainTag – [in] Pointer to `SaHpiTextBufferT` containing the domain tag.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_INVALID_PARAMS` is returned if the `SaHpiTextBufferT` structure passed as *DomainTag* is not valid. This would occur when:

- The *DataType* is not one of the enumerated values for that type, or
- The data field contains characters that are not legal according to the value of *DataType*, or
- The *Language* is not one of the enumerated values for that type when the *DataType* is `SAHPI_TL_TYPE_UNICODE` or `SAHPI_TL_TYPE_TEXT`.

`SA_ERR_HPI_INVALID_PARAMS` is returned if the *DomainTag* pointer is passed in as NULL.

Remarks

Typically, the HPI implementation will provide an appropriate default value for the domain tag; this function is provided so that an HPI User can override the default, if desired. The value of the domain tag may be retrieved from the domain's information structure.

The domain tag is not necessarily persistent, and may return to its default value if the domain controller function for the domain restarts.

6.3 Resource Presence Table

The HPI implementation is required to discover the resources and entities under management control. An HPI User may then access this information via the Resource Presence Table (RPT) for each domain in the system. This table should represent up-to-date information on the resources currently present in the domain with which it is associated. The RPT may be accessed using the RPT functions described below.

The RPT contains an entry for each resource in the domain. Each entry contains a *ResourceId* for that resource. Flags in the capabilities field of each RPT entry identify the functionality supported by that resource.

6.3.1 saHpiRptEntryGet()

This function retrieves resource information for the specified entry of the resource presence table. This function allows an HPI User to read the RPT entry-by-entry.

Prototype

```
SaErrorT SAHPI_API saHpiRptEntryGet (
    SAHPI_IN  SaHpiSessionIdT     SessionId,
    SAHPI_IN  SaHpiEntryIdT       EntryId,
    SAHPI_OUT SaHpiEntryIdT       *NextEntryId,
    SAHPI_OUT SaHpiRptEntryT      *RptEntry
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

EntryId – [in] Identifier of the RPT entry to retrieve. Reserved *EntryId* values:

- `SAHPI_FIRST_ENTRY` Get first entry.
- `SAHPI_LAST_ENTRY` Reserved as delimiter for end of list. Not a valid entry identifier.

NextEntryId – [out] Pointer to location to store the *EntryId* of next entry in RPT.

RptEntry – [out] Pointer to the structure to hold the returned RPT entry.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_NOT_PRESENT` is returned when the:

- Entry identified by *EntryId* is not present.
- *EntryId* is `SAHPI_FIRST_ENTRY` and the RPT is empty.

`SA_ERR_HPI_INVALID_PARAMS` is returned if the:

- *RptEntry* pointer is passed in as NULL.
- *NextEntryId* pointer is passed in as NULL.
- *EntryId* is an invalid reserved value such as `SAHPI_LAST_ENTRY`.

Remarks

If the *EntryId* parameter is set to `SAHPI_FIRST_ENTRY`, the first entry in the RPT will be returned. When an entry is successfully retrieved, *NextEntryId* will be set to the identifier of the next valid entry; however, when the last entry has been retrieved, *NextEntryId* will be set to `SAHPI_LAST_ENTRY`. To retrieve an entire list of entries, call this function first with an *EntryId* of `SAHPI_FIRST_ENTRY` and then use the returned *NextEntryId* in the next call. Proceed until the *NextEntryId* returned is `SAHPI_LAST_ENTRY`.

At initialization, an HPI User may not wish to turn on eventing, since the context of the events, as provided by the RPT, is not known. In this instance, if a FRU is inserted into the system while the RPT is being read entry by entry, the resource associated with that FRU may be missed. (Keep in mind that there is no specified ordering for the RPT entries.) The update counter provides a means for insuring that no resources are missed when stepping through the RPT. In order to use this feature, an HPI User should invoke `saHpiDomainInfoGet()`, and get the update counter value before retrieving the first RPT entry. After reading the last entry, an HPI User should again invoke the `saHpiDomainInfoGet()` to get the RPT update counter value. If the update counter has not been incremented, no new records have been added.

6.3.2 saHpiRptEntryGetByResourceId()

This function retrieves resource information from the resource presence table for the specified resource using its *ResourceId*.

Prototype

```
SaErrorT SAHPI_API saHpiRptEntryGetByResourceId (
    SAHPI_IN  SaHpiSessionIdT      SessionId,
    SAHPI_IN  SaHpiResourceIdT     ResourceId,
    SAHPI_OUT SaHpiRptEntryT       *RptEntry
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

ResourceId – [in] Resource identified for this operation.

RptEntry – [out] Pointer to structure to hold the returned RPT entry.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_INVALID_PARAMS` is returned if the *RptEntry* pointer is passed in as NULL.

Remarks

Typically at start-up, the RPT is read entry-by-entry, using `saHpiRptEntryGet()`. From this, an HPI User can establish the set of *ResourceIds* to use for future calls to the HPI functions.

However, there may be other ways of learning *ResourceIds* without first reading the RPT. For example, resources may be added to the domain while the system is running in response to a hot swap action. When a resource is added, the application will receive a hot swap event containing the *ResourceId* of the new resource. The application may then want to search the RPT for more detailed information on the newly added resource. In this case, the *ResourceId* can be used to locate the applicable RPT entry information.

Note that `saHpiRptEntryGetByResourceId()` is valid in any hot swap state, except for `SAHPI_HS_STATE_NOT_PRESENT`.

6.3.3 saHpiResourceSeveritySet()

This function allows an HPI User to set the severity level applied to an event issued if a resource unexpectedly becomes unavailable to the HPI. A resource may become unavailable for several reasons including:

- The FRU associated with the resource is no longer present in the system (a surprise extraction has occurred.)
- A catastrophic failure has occurred.

Prototype

```
SaErrorT SAHPI_API saHpiResourceSeveritySet (
    SAHPI_IN  SaHpiSessionIdT     SessionId,
    SAHPI_IN  SaHpiResourceIdT    ResourceId,
    SAHPI_IN  SaHpiSeverityT      Severity
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

ResourceId – [in] Resource identified for this operation.

Severity – [in] Severity level of event issued when the resource unexpectedly becomes unavailable to the HPI.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_INVALID_PARAMS` is returned when the value for *Severity* is not one of the valid enumerated values for this type.

Remarks

Typically, the HPI implementation will provide an appropriate default value for the resource severity, which may vary by resource; an HPI User can override this default value by use of this function.

If a resource is removed from, then re-added to the RPT (e.g., because of a hot swap action), the HPI implementation may reset the value of the resource severity.

6.3.4 saHpiResourceTagSet()

This function allows an HPI User to set the resource tag of an RPT entry for a particular resource.

Prototype

```
SaErrorT SAHPI_API saHpiResourceTagSet (
    SAHPI_IN  SaHpiSessionIdT     SessionId,
    SAHPI_IN  SaHpiResourceIdT    ResourceId,
    SAHPI_IN  SaHpiTextBufferT    *ResourceTag
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

ResourceId – [in] Resource identified for this operation.

ResourceTag – [in] Pointer to `SaHpiTextBufferT` containing the resource tag.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_INVALID_PARAMS` is returned if the `SaHpiTextBufferT` structure passed as *ResourceTag* is not valid. This would occur when:

- The *DataType* is not one of the enumerated values for that type, or
- The data field contains characters that are not legal according to the value of `DataType`, or
- The *Language* is not one of the enumerated values for that type when the `DataType` is `SAHPI_TL_TYPE_UNICODE` or `SAHPI_TL_TYPE_TEXT`.

`SA_ERR_HPI_INVALID_PARAMS` is returned if the *ResourceTag* pointer is passed in as NULL.

Remarks

The resource tag is a data field within an RPT entry available to an HPI User for associating application specific data with a resource. HPI User supplied data is purely informational and is not used by the HPI implementation, domain, or associated resource. For example, an HPI User can set the resource tag to a "descriptive" value, which can be used to identify the resource in messages to a human operator.

Since the resource tag is contained within an RPT entry, its scope is limited to a single domain. A resource that exists in more than one domain will have independent resource tags within each domain.

Typically, the HPI implementation will provide an appropriate default value for the resource tag; this function is provided so that an HPI User can override the default, if desired. The value of the resource tag may be retrieved from the resource's RPT entry.

If a resource is removed from, then re-added to the RPT (e.g., because of a hot swap action), the HPI implementation may reset the value of the resource tag.

6.3.5 saHpiResourceIdGet()

This function returns the *ResourceId* of the resource associated with the entity upon which the HPI User is running.

Prototype

```
SaErrorT SAHPI_API saHpiResourceIdGet (
    SAHPI_IN  SaHpiSessionIdT      SessionId,
    SAHPI_OUT SaHpiResourceIdT     *ResourceId
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

ResourceId – [out] Pointer to location to hold the returned *ResourceId*.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_INVALID_PARAMS` is returned if the *ResourceId* pointer is passed in as NULL.

`SA_ERR_HPI_NOT_PRESENT` is returned if the entity the HPI User is running on is not manageable in the addressed domain.

`SA_ERR_HPI_UNKNOWN` is returned if the domain controller cannot determine an appropriate response. That is, there may be an appropriate *ResourceId* in the domain to return, but it cannot be determined.

Remarks

This function must be issued within a session to a domain that includes a resource associated with the entity upon which the HPI User is running, or the `SA_ERR_HPI_NOT_PRESENT` return will be issued.

Since entities are contained within other entities, there may be multiple possible resources that could be returned to this call. For example, if there is a *ResourceId* associated with a particular compute blade upon which the HPI User is running, and another associated with the chassis which contains the compute blade, either could logically be returned as an indication of a resource associated with the entity upon which the HPI User was running. The function should return the *ResourceId* of the “smallest” resource that is associated with the HPI User. So, in the example above, the function should return the *ResourceId* of the compute blade.

Once the function has returned the *ResourceId*, the HPI User may issue further HPI calls using that *ResourceId* to learn the type of resource that been identified.

6.4 Event Log Management

A domain controller maintains a Domain Event Log for events collected in that domain. Additionally, individual resources may maintain Resource Event Logs. If a resource supports a Resource Event Log, it will indicate this by having the Resource Event Log capability (`SAHPI_CAPABILITY_EVENT_LOG`) set in its RPT table entry.

The Event Log API allows each Event Log to have its own time clock, which is used for setting timestamps on Event Log entries (the "timestamp clock"). This allows for Event Logs to be implemented on separate hardware units which may have their own time-of-day clocks. An HPI User should therefore be aware that Event Log entries from a given Event Log are time-stamped using that Event Log's time clock, which may be different to other time clocks. The functions `saHpiEventLogTimeGet()` and `saHpiEventLogTimeSet()` are provided to allow an individual Event Log's clock to be read and set, respectively.

Event Log entries contain two timestamps: the Event Log timestamp and the embedded event timestamp. The Event Log timestamp indicates the time that the event was placed in the Event Log. The embedded event timestamp indicates the best approximation to when the event actually occurred (it is permissible to return `SAHPI_TIME_UNSPECIFIED` for this timestamp). Because of this, it is possible that entries are chronologically ordered by the Event Log timestamp, but are out of chronological order if looking at their event timestamps.

Event Logs are not re-ordered, nor are existing entries re-timestamped, as a result of a `saHpiEventLogTimeSet()`. References to the events being chronologically ordered mean that they retain the order in which they were actually added to the Event Log, regardless of what the timestamp fields in the entries may indicate as a result of clocks being reset.

Exactly what events are placed in the Resource and Domain Event Logs and how long they remain in the Event Log are implementation-specific. All Event Logs, either the Domain Event Log, or a Resource Event Log may be managed by an HPI User through these HPI functions. Management of an Event Log includes activities such as reading records from it, writing records to it, clearing it, setting the timestamp clock, etc.

Using the value `SAHPI_UNSPECIFIED_RESOURCE_ID` as the *ResourceId* specifies a Domain Event Log. Resource Event Logs are specified with the *ResourceId* of the resource that is hosting the Resource Event Log; thus, the *ResourceId* must be a value other than `SAHPI_UNSPECIFIED_RESOURCE_ID`.

A Domain Event Log may also be visible as a specific Resource Event Log. If so, then it is possible that the Domain Event Log may be accessed either using `SAHPI_UNSPECIFIED_RESOURCE_ID` as the *ResourceId*, or with the *ResourceId* of the resource that is hosting the Domain Event Log.

Each entry in an Event Log has an *EntryId* which is unique within that Event Log. Once an entry is made in an Event Log, its *EntryId* will not change. *EntryIds* may be reused in an Event Log only after a particular Event Log entry is permanently removed from the Event Log (e.g., by clearing the Event Log, or by an implementation-specific policy of removing obsolete entries). There is no requirement that the implementation assign *EntryIds* sequentially.

The Event Log is read chronologically by using the *NextEntryId* and *PrevEntryId* values returned by `saHpiEventLogEntryGet()`, which will identify the *EntryId* of the next or previous entry in chronological order, respectively.

When an overflow occurs, an Event Log will take one of two implementation-dependent actions. Either it will drop new events for which there is no space, or it will delete existing entries to make room for new ones. Which action a particular implementation takes when an overflow occurs is reported in the Event Log info record.

Event Log implementations may include policies to automatically remove obsolete entries periodically or to make room for new entries when needed. This automatic removal of obsolete entries is not an Event Log overflow. Rather, an overflow condition exists when an Event Log is not able to hold all entries that its implemented policy indicates should be held.

When an overflow occurs, and either new events are dropped, or existing, non-obsolete events are overwritten, the overflow flag will be set in the Event Log info record. The overflow flag will remain set until the flag is reset by calling `saHpiEventLogOverflowReset()` or `saHpiEventLogClear()`.

6.4.1 saHpiEventLogInfoGet()

This function retrieves the current number of entries in the Event Log, total size of the Event Log, the time of the most recent update to the Event Log, the current value of the Event Log's clock (i.e., timestamp that would be placed on an entry at this moment), the enabled/disabled status of the Event Log (see Section 6.4.8 on page 57), the overflow flag, and the action taken by the Event Log if an overflow occurs.

Prototype

```
SaErrorT SAHPI_API saHpiEventLogInfoGet (
    SAHPI_IN  SaHpiSessionIdT     SessionId,
    SAHPI_IN  SaHpiResourceIdT    ResourceId,
    SAHPI_OUT SaHpiEventLogInfoT  *Info
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

ResourceId – [in] Identifier for the Resource containing the Event Log. Set to `SAHPI_UNSPECIFIED_RESOURCE_ID` to address the Domain Event Log.

Info – [out] Pointer to the returned Event Log information.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_CAPABILITY` is returned if the resource does not have an Event Log capability (`SAHPI_CAPABILITY_EVENT_LOG`) set. Note this condition only applies to Resource Event Logs. Domain Event Logs are mandatory, and should not return this code.

`SA_ERR_HPI_INVALID_PARAMS` is returned if the *Info* pointer is passed in as NULL.

Remarks

The *size* field in the returned Event Log information indicates the maximum number of entries that can be held in the Event Log. This number should be constant for a particular Event Log.

6.4.2 saHpiEventLogEntryGet()

This function retrieves an Event Log entry.

Prototype

```
SaErrorT SAHPI_API saHpiEventLogEntryGet (
    SAHPI_IN    SaHpiSessionIdT          SessionId,
    SAHPI_IN    SaHpiResourceIdT         ResourceId,
    SAHPI_IN    SaHpiEventLogEntryIdT    EntryId,
    SAHPI_OUT   SaHpiEventLogEntryIdT    *PrevEntryId,
    SAHPI_OUT   SaHpiEventLogEntryIdT    *NextEntryId,
    SAHPI_OUT   SaHpiEventLogEntryT      *EventLogEntry,
    SAHPI_INOUT SaHpiRdrT                *Rdr,
    SAHPI_INOUT SaHpiRptEntryT           *RptEntry
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

ResourceId – [in] Identifier for the Resource containing the Event Log. Set to `SAHPI_UNSPECIFIED_RESOURCE_ID` to address the Domain Event Log.

EntryId – [in] Identifier of event log entry to retrieve. Reserved values:

- `SAHPI_OLDEST_ENTRY` Oldest entry in the Event Log.
- `SAHPI_NEWEST_ENTRY` Newest entry in the Event Log.
- `SAHPI_NO_MORE_ENTRIES` Not valid for this parameter. Used only when retrieving the next and previous *EntryIds*.

PrevEntryId – [out] Event Log entry identifier for the previous (older adjacent) entry. Reserved values:

- `SAHPI_OLDEST_ENTRY` Not valid for this parameter. Used only for the *EntryId* parameter.
- `SAHPI_NEWEST_ENTRY` Not valid for this parameter. Used only for the *EntryId* parameter.
- `SAHPI_NO_MORE_ENTRIES` No more entries in the Event Log before the one referenced by the *EntryId* parameter.

NextEntryId – [out] Event Log entry identifier for the next (newer adjacent) entry. Reserved values:

- `SAHPI_OLDEST_ENTRY` Not valid for this parameter. Used only for the *EntryId* parameter.
- `SAHPI_NEWEST_ENTRY` Not valid for this parameter. Used only for the *EntryId* parameter.
- `SAHPI_NO_MORE_ENTRIES` No more entries in the Event Log after the one referenced by the *EntryId* parameter.

EventLogEntry – [out] Pointer to retrieved Event Log entry.

Rdr – [in/out] Pointer to structure to receive resource data record associated with the Event Log entry, if available. If NULL, no RDR data will be returned.

RptEntry – [in/out] Pointer to structure to receive RPT entry associated with the Event Log entry, if available. If NULL, no RPT entry data will be returned.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

SA_ERR_HPI_CAPABILITY is returned if the resource does not have an Event Log capability (SAHPI_CAPABILITY_EVENT_LOG) set. Note this condition only applies to Resource Event Logs. Domain Event Logs are mandatory, and should not return this code.

SA_ERR_HPI_NOT_PRESENT is returned when:

- The Event Log has no entries.
- The entry identified by *EntryId* is not present.

SA_ERR_HPI_INVALID_PARAMS is returned when:

- Any of *PrevEntryId*, *NextEntryId* and *EventLogEntry* pointers are passed in as NULL.
- SAHPI_NO_MORE_ENTRIES is passed in to *EntryId*.

Remarks

The special *EntryIds* SAHPI_OLDEST_ENTRY and SAHPI_NEWEST_ENTRY are used to select the oldest and newest entries, respectively, in the Event Log being read. A returned *NextEntryId* of SAHPI_NO_MORE_ENTRIES indicates that the newest entry has been returned; there are no more entries going forward (time-wise) in the Event Log. A returned *PrevEntryId* of SAHPI_NO_MORE_ENTRIES indicates that the oldest entry has been returned.

To retrieve an entire list of entries going forward (oldest entry to newest entry) in the Event Log, call this function first with an *EntryId* of SAHPI_OLDEST_ENTRY and then use the returned *NextEntryId* as the *EntryId* in the next call. Proceed until the *NextEntryId* returned is SAHPI_NO_MORE_ENTRIES.

To retrieve an entire list of entries going backward (newest entry to oldest entry) in the Event Log, call this function first with an *EntryId* of SAHPI_NEWEST_ENTRY and then use the returned *PrevEntryId* as the *EntryId* in the next call. Proceed until the *PrevEntryId* returned is SAHPI_NO_MORE_ENTRIES.

Event Logs may include RPT entries and resource data records associated with the resource and sensor issuing an event along with the basic event data in the Event Log. Because the system may be reconfigured after the event was entered in the Event Log, this stored information may be important to interpret the event. If the Event Log includes logged RPT entries and/or RDRs, and if an HPI User provides a pointer to a structure to receive this information, it will be returned along with the Event Log entry.

If an HPI User provides a pointer for an RPT entry, but the Event Log does not include a logged RPT entry for the Event Log entry being returned, *RptEntry->ResourceCapabilities* will be set to zero. No valid *RptEntry* will have a zero *Capabilities* field value.

If an HPI User provides a pointer for an RDR, but the Event Log does not include a logged RDR for the Event Log entry being returned, *Rdr->RdrType* will be set to SAHPI_NO_RECORD.

The *EntryIds* returned via the *PrevEntryId* and *NextEntryId* parameters may not be in sequential order, but will reflect the previous and next entries in a chronological ordering of the Event Log, respectively.

6.4.3 saHpiEventLogEntryAdd()

This function enables an HPI user to add entries to the Event Log.

Prototype

```
SaErrorT SAHPI_API saHpiEventLogEntryAdd (
    SAHPI_IN SaHpiSessionIdT     SessionId,
    SAHPI_IN SaHpiResourceIdT    ResourceId,
    SAHPI_IN SaHpiEventT         *EvtEntry
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

ResourceId – [in] Identifier for the Resource containing the Event Log. Set to `SAHPI_UNSPECIFIED_RESOURCE_ID` to address the Domain Event Log.

EvtEntry – [in] Pointer to event data to write to the Event Log. The *Event* field must be of type `SAHPI_ET_USER`, and the *Source* field must be `SAHPI_UNSPECIFIED_RESOURCE_ID`.

Return Value

`SA_OK` is returned if the event is successfully written in the Event Log; otherwise, an error code is returned.

`SA_ERR_HPI_CAPABILITY` is returned if the resource does not have an Event Log capability (`SAHPI_CAPABILITY_EVENT_LOG`) set. Note this condition only applies to Resource Event Logs. Domain Event Logs are mandatory, and should not return this code.

`SA_ERR_HPI_INVALID_DATA` is returned if the event *DataLength* is larger than that supported by the implementation and reported in the Event Log info record.

`SA_ERR_HPI_INVALID_PARAMS` is returned if the:

- *EvtEntry* pointer is passed in as NULL.
- Event structure passed via the *EvtEntry* parameter is not an event of type `SAHPI_ET_USER` with the *Source* field set to `SAHPI_UNSPECIFIED_RESOURCE_ID`.
- The *Severity* is not one of the valid enumerated values for this type.
- `SaHpiTextBufferT` structure passed as part of the User Event structure is not valid. This would occur when:
  - The *DataType* is not one of the enumerated values for that type, or
  - The data field contains characters that are not legal according to the value of *DataType*, or
  - The *Language* is not one of the enumerated values for that type when the *DataType* is `SAHPI_TL_TYPE_UNICODE` or `SAHPI_TL_TYPE_TEXT`.

`SA_ERR_HPI_OUT_OF_SPACE` is returned if the event cannot be written to the Event Log because the Event Log is full, and the Event Log *OverflowAction* is `SAHPI_EL_OVERFLOW_DROP`.

Remarks

This function writes an event in the addressed Event Log. Nothing else is done with the event.

If the Event Log is full, overflow processing occurs as defined by the Event Log's *OverflowAction* setting, reported in the Event Log info record. If, due to an overflow condition, the event is not written, or if existing events are overwritten, then the *OverflowFlag* in the Event Log info record will be set, just as it would be if an internally generated event caused an overflow condition. If the Event Log's *OverflowAction* is `SAHPI_EL_OVERFLOW_DROP`, then an error will be returned (`SA_ERR_HPI_OUT_OF_SPACE`) indicating that the `saHpiEventLogEntryAdd()` function did not add the event to the Event Log. If the Event Log's *OverflowAction* is `SAHPI_EL_OVERFLOW_OVERWRITE`, then the `saHpiEventLogEntryAdd()` function will return `SA_OK`, indicating that the event was added to the Event Log, even though an overflow occurred as a side-effect of this operation. The overflow may be detected by checking the *OverflowFlag* in the Event Log info record.

Specific implementations of HPI may have restrictions on how much data may be passed to the `saHpiEventLogEntryAdd()` function. The Event Log info record reports the maximum *DataLength* that is supported by the Event Log for User Events. If `saHpiEventLogEntryAdd()` is called with a User Event that has a larger *DataLength* than is supported, the event will not be added to the Event Log, and an error will be returned.

6.4.4 saHpiEventLogClear()

This function erases the contents of the specified Event Log.

Prototype

```
SaErrorT SAHPI_API saHpiEventLogClear (
    SAHPI_IN  SaHpiSessionIdT     SessionId,
    SAHPI_IN  SaHpiResourceIdT    ResourceId
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

ResourceId – [in] Identifier for the Resource containing the Event Log. Set to `SAHPI_UNSPECIFIED_RESOURCE_ID` to address the Domain Event Log.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_CAPABILITY` is returned if the resource does not have an Event Log capability (`SAHPI_CAPABILITY_EVENT_LOG`) set. Note this condition only applies to Resource Event Logs. Domain Event Logs are mandatory, and should not return this code

Remarks

The *OverflowFlag* field in the Event Log info record will be reset when this function is called.

6.4.5 saHpiEventLogTimeGet()

This function retrieves the current time from the Event Log's clock. This clock is used to timestamp entries written into the Event Log.

Prototype

```
SaErrorT SAHPI_API saHpiEventLogTimeGet (
    SAHPI_IN  SaHpiSessionIdT      SessionId,
    SAHPI_IN  SaHpiResourceIdT     ResourceId,
    SAHPI_OUT SaHpiTimeT           *Time
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

ResourceId – [in] Identifier for the Resource containing the Event Log. Set to `SAHPI_UNSPECIFIED_RESOURCE_ID` to address the Domain Event Log.

Time – [out] Pointer to the returned current Event Log time.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_CAPABILITY` is returned if the resource does not have an Event Log capability (`SAHPI_CAPABILITY_EVENT_LOG`) set. Note this condition only applies to Resource Event Logs. Domain Event Logs are mandatory, and should not return this code.

`SA_ERR_HPI_INVALID_PARAMS` is returned if the *Time* pointer is passed in as NULL.

Remarks

If the implementation cannot supply an absolute time value, then it may supply a time relative to some system-defined epoch, such as system boot. If the time value is less than or equal to `SAHPI_TIME_MAX_RELATIVE`, then it is relative; if it is greater than `SAHPI_TIME_MAX_RELATIVE`, then it is absolute. The HPI implementation must provide valid timestamps for Event Log entries, using a default time base if no time has been set. Thus, the value `SAHPI_TIME_UNSPECIFIED` is never returned.

6.4.6 saHpiEventLogTimeSet()

This function sets the Event Log's clock, which is used to timestamp events written into the Event Log.

Prototype

```
SaErrorT SAHPI_API saHpiEventLogTimeSet (
    SAHPI_IN SaHpiSessionIdT      SessionId,
    SAHPI_IN SaHpiResourceIdT     ResourceId,
    SAHPI_IN SaHpiTimeT           Time
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

ResourceId – [in] Identifier for the Resource containing the Event Log. Set to `SAHPI_UNSPECIFIED_RESOURCE_ID` to address the Domain Event Log.

Time – [in] Time to which the Event Log clock should be set.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_CAPABILITY` is returned if the resource does not have an Event Log capability (`SAHPI_CAPABILITY_EVENT_LOG`) set. Note this condition only applies to Resource Event Logs. Domain Event Logs are mandatory, and should not return this code.

`SA_ERR_HPI_INVALID_PARAMS` is returned when the *Time* parameter is set to `SAHPI_TIME_UNSPECIFIED`.

For situations when the underlying implementation cannot represent a time value that is specified in *Time*, `SA_ERR_HPI_INVALID_DATA` is returned.

Remarks

If the *Time* parameter value is less than or equal to `SAHPI_TIME_MAX_RELATIVE`, but not `SAHPI_TIME_UNSPECIFIED`, then it is relative; if it is greater than `SAHPI_TIME_MAX_RELATIVE`, then it is absolute. Setting this parameter to the value `SAHPI_TIME_UNSPECIFIED` is invalid and will result in an error return code of `SA_ERR_HPI_INVALID_PARAMS`.

Entries placed in the Event Log after this function is called will have Event Log timestamps (i.e., the *Timestamp* field in the `SaHpiEventLogEntryT` structure) based on the new time. Setting the clock does not affect existing Event Log entries. If the time is set to a relative time, subsequent entries placed in the Event Log will have an Event Log timestamp expressed as a relative time; if the time is set to an absolute time, subsequent entries will have an Event Log timestamp expressed as an absolute time.

This function only sets the Event Log time clock and does not have any direct bearing on the timestamps placed on events (i.e., the *Timestamp* field in the `SaHpiEventT` structure), or the timestamps placed in the domain RPT info record. Setting the clocks used to generate timestamps other than Event Log timestamps is implementation-dependent, and should be documented by the HPI implementation provider.

Some underlying implementations may not be able to handle the same relative and absolute time ranges, as those defined in HPI. Such limitations will be documented. When a time value is set in a region that is not supported by the implementation, an error code of `SA_ERR_HPI_INVALID_DATA` will be returned.

6.4.7 saHpiEventLogStateGet()

This function enables an HPI User to get the Event Log state.

Prototype

```
SaErrorT SAHPI_API saHpiEventLogStateGet (
    SAHPI_IN  SaHpiSessionIdT     SessionId,
    SAHPI_IN  SaHpiResourceIdT    ResourceId,
    SAHPI_OUT SaHpiBoolT          *EnableState
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

ResourceId – [in] Identifier for the Resource containing the Event Log. Set to `SAHPI_UNSPECIFIED_RESOURCE_ID` to address the Domain Event Log.

EnableState – [out] Pointer to the current Event Log enable state. True indicates that the Event Log is enabled; False indicates that it is disabled.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_CAPABILITY` is returned if the resource does not have an Event Log capability (`SAHPI_CAPABILITY_EVENT_LOG`) set. Note this condition only applies to Resource Event Logs. Domain Event Logs are mandatory, and should not return this code.

`SA_ERR_HPI_INVALID_PARAMS` is returned if the *EnableState* pointer is passed in as NULL.

Remarks

If the Event Log is disabled, no events generated within the HPI implementation will be added to the Event Log. Events may still be added to the Event Log with the `saHpiEventLogEntryAdd()` function. When the Event Log is enabled, events may be automatically added to the Event Log as they are generated in a resource or a domain, however, it is implementation-specific which events are automatically added to any Event Log.

6.4.8 saHpiEventLogStateSet()

This function enables an HPI User to set the Event Log enabled state.

Prototype

```
SaErrorT SAHPI_API saHpiEventLogStateSet (
    SAHPI_IN SaHpiSessionIdT      SessionId,
    SAHPI_IN SaHpiResourceIdT     ResourceId,
    SAHPI_IN SaHpiBoolT           EnableState
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

ResourceId – [in] Identifier for the Resource containing the Event Log. Set to `SAHPI_UNSPECIFIED_RESOURCE_ID` to address the Domain Event Log.

EnableState – [in] Event Log state to be set. True indicates that the Event Log is to be enabled; False indicates that it is to be disabled.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_CAPABILITY` is returned if the resource does not have an Event Log capability (`SAHPI_CAPABILITY_EVENT_LOG`) set. Note this condition only applies to Resource Event Logs. Domain Event Logs are mandatory, and should not return this code.

Remarks

If the Event Log is disabled no events generated within the HPI implementation will be added to the Event Log. Events may still be added to the Event Log using the `saHpiEventLogEntryAdd()` function. When the Event Log is enabled events may be automatically added to the Event Log as they are generated in a resource or a domain. The actual set of events that are automatically added to any Event Log is implementation-specific.

Typically, the HPI implementation will provide an appropriate default value for this parameter, which may vary by resource. This function is provided so that an HPI User can override the default, if desired.

If a resource hosting an Event Log is re-initialized (e.g., because of a hot swap action), the HPI implementation may reset the value of this parameter.

6.4.9 saHpiEventLogOverflowReset()

This function resets the *OverflowFlag* in the Event Log info record of the specified Event Log.

Prototype

```
SaErrorT SAHPI_API saHpiEventLogOverflowReset (
    SAHPI_IN  SaHpiSessionIdT     SessionId,
    SAHPI_IN  SaHpiResourceIdT    ResourceId
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

ResourceId – [in] Identifier for the Resource containing the Event Log. Set to `SAHPI_UNSPECIFIED_RESOURCE_ID` to address the Domain Event Log.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_INVALID_CMD` is returned if the implementation does not support independent clearing of the *OverflowFlag* on this Event Log.

`SA_ERR_HPI_CAPABILITY` is returned if the resource does not have an Event Log capability (`SAHPI_CAPABILITY_EVENT_LOG`) set. Note this condition only applies to Resource Event Logs. Domain Event Logs are mandatory, and should not return this code.

Remarks

The only effect of this function is to clear the *OverflowFlag* field in the Event Log info record for the specified Event Log. If the Event Log is still full, the *OverflowFlag* will be set again as soon as another entry needs to be added to the Event Log.

Some Event Log implementations may not allow resetting of the *OverflowFlag* except as a by-product of clearing the entire Event Log with the `saHpiEventLogClear()` function. Such an implementation will return the error code, `SA_ERR_HPI_INVALID_CMD` to this function. The *OverflowResetable* flag in the Event Log info record indicates whether or not the implementation supports resetting the *OverflowFlag* without clearing the Event Log.

6.5 Events

Events are collected and processed by the domain controller. The domain controller is responsible for processing each event in the order it is received. Event processing by the domain controller includes optionally logging the event to the Domain Event Log, and publishing the event to all session subscriptions.

The HPI event management service allows a session user to receive events as they are generated, or when requested. Before events can be received, the session user must subscribe to the domain's events. A session that subscribes to events will receive all events, which are collected in the domain controller for the domain associated with the session; that is, all events relevant to that domain.

Invoking the `saHpiEventGet()` function retrieves events. This function may be invoked in blocking or non-blocking modes; to retrieve events as they occur, invoke `saHpiEventGet()` with blocking set (i.e., set the timeout parameter to `SAHPI_TIMEOUT_BLOCK`), and it will return when an event is available. In either case, `saHpiEventGet()` will return the next available event on the session's event queue, if there is one already there.

After the subscription, all subsequent events are added to the event queue as the domain controller collects them. Capacity and management of event queues is implementation-specific. Event queues may be of a fixed size, or may be dynamically sized depending on available memory resources at any moment. A mechanism is provided, however, to report an event queue overflow to an HPI User. When an overflow is reported, this means that one or more events were unable to be queued to a particular event queue because of space limitations, regardless of how the implementation manages event queue space.

Implementations should drop the new, incoming events when reporting queue overflows rather than deleting existing queued events to make room for the new ones. If additional events are to be queued before an HPI User has read any events from the queue (and thus had the opportunity to be informed of the overflow), and the implementation is able to queue the additional events, it may do so, but should leave the overflow flag set until an HPI User next calls `saHpiEventGet()`.

Implementations should be designed to ensure that at least one event may always be placed on every event queue. Thus, it should not be possible to have the overflow flag set for a queue that has no events queued. This is important because the overflow flag is reported to an HPI User in conjunction with an event being returned to an HPI User.

6.5.1 saHpiSubscribe()

This function allows an HPI User to subscribe for events. This single call provides subscription to all session events, regardless of event type or event severity.

Prototype

```
SaErrorT SAHPI_API saHpiSubscribe (
    SAHPI_IN SaHpiSessionIdT      SessionId
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_DUPLICATE` is returned when a subscription is already in place for this session.

Remarks

Only one subscription is allowed per session, and additional subscribers will receive an appropriate error code. No event filtering will be done by the HPI implementation.

6.5.2 saHpiUnsubscribe()

This function removes the event subscription for the session.

Prototype

```
SaErrorT SAHPI_API saHpiUnsubscribe (
    SAHPI_IN SaHpiSessionIdT  SessionId
);
```

Parameters

SessionId – [in] Session for which event subscription will be closed.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_INVALID_REQUEST` is returned if the session is not currently subscribed for events.

Remarks

After removal of a subscription, additional `saHpiEventGet()` calls will not be allowed on the session unless an HPI User re-subscribes for events on the session first. Any events that are still in the event queue when this function is called will be cleared from it.

6.5.3 saHpiEventGet()

This function allows an HPI User to get an event. This call is only valid within a session that has subscribed for events.

Prototype

```
SaErrorT SAHPI_API saHpiEventGet (
    SAHPI_IN SaHpiSessionIdT            SessionId,
    SAHPI_IN SaHpiTimeoutT              Timeout,
    SAHPI_OUT SaHpiEventT               *Event,
    SAHPI_INOUT SaHpiRdrT               *Rdr,
    SAHPI_INOUT SaHpiRptEntryT          *RptEntry,
    SAHPI_INOUT SaHpiEvtQueueStatusT    *EventQueueStatus
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

Timeout – [in] The number of nanoseconds to wait for an event to arrive. Reserved time out values:

- `SAHPI_TIMEOUT_IMMEDIATE` Time out immediately if there are no events available (non-blocking call).
- `SAHPI_TIMEOUT_BLOCK` Call should not return until an event is retrieved.

Event – [out] Pointer to the next available event.

Rdr – [in/out] Pointer to structure to receive the resource data associated with the event. If NULL, no RDR will be returned.

RptEntry – [in/out] Pointer to structure to receive the RPT entry associated with the resource that generated the event. If NULL, no RPT entry will be returned.

EventQueueStatus – [in/out] Pointer to location to store event queue status. If NULL, event queue status will not be returned.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_INVALID_REQUEST` is returned if an HPI User is not currently subscribed for events in this session.

`SA_ERR_HPI_INVALID_PARAMS` is returned if the:

- *Event* pointer is passed in as NULL.
- *Timeout* parameter is not set to `SAHPI_TIMEOUT_BLOCK`, `SAHPI_TIMEOUT_IMMEDIATE` or a positive value.

`SA_ERR_HPI_TIMEOUT` is returned if no event is available to return within the timeout period. If `SAHPI_TIMEOUT_IMMEDIATE` is passed in the *Timeout* parameter, this error return will be used if there is no event queued when the function is called.

Remarks

`SaHpiEventGet()` will also return an *EventQueueStatus* flag to an HPI User. This flag indicates whether or not a queue overflow has occurred. The overflow flag is set if any events were unable to be queued because of space limitations in the interface implementation. The overflow flag is reset whenever `saHpiEventGet()` is called.

If there are one or more events on the event queue when this function is called, it will immediately return the next event on the queue. Otherwise, if the *Timeout* parameter is SAHPI_TIMEOUT_IMMEDIATE, it will return SA_ERR_HPI_TIMEOUT immediately. Otherwise, it will block for time specified by the timeout parameter; if an event is added to the queue within that time it will be returned immediately; if not, saHpiEventGet() will return SA_ERR_HPI_TIMEOUT. If the *Timeout* parameter is SAHPI_TIMEOUT_BLOCK, the saHpiEventGet() will block indefinitely, until an event becomes available, and then return that event. This provides for notification of events as they occur.

If an HPI User provides a pointer for an RPT entry, but the event does not include a valid *ResourceId* for a resource in the domain (e.g., OEM or USER type event), then the *RptEntry->ResourceCapabilities* field will be set to zero. No valid RPT entry will have a zero *ResourceCapabilities*.

If an HPI User provides a pointer for an RDR, but there is no valid RDR associated with the event being returned (e.g., returned event is not a sensor event), then the *Rdr->RdrType* field will be set to SAHPI_NO_RECORD.

The timestamp reported in the returned event structure is the best approximation an implementation has to when the event actually occurred. The implementation may need to make an approximation (such as the time the event was placed on the event queue) because it may not have access to the actual time the event occurred. The value SAHPI_TIME_UNSPECIFIED indicates that the time of the event cannot be determined.

If the implementation cannot supply an absolute timestamp, then it may supply a timestamp relative to some system-defined epoch, such as system boot. If the timestamp value is less than or equal to SAHPI_TIME_MAX_RELATIVE, but not SAHPI_TIME_UNSPECIFIED, then it is relative; if it is greater than SAHPI_TIME_MAX_RELATIVE, then it is absolute.

If an HPI User passes a NULL pointer for the returned *EventQueueStatus* pointer, the event status will not be returned, but the overflow flag, if set, will still be reset. Thus, if an HPI User needs to know about event queue overflows, the *EventQueueStatus* parameter should never be NULL, and the overflow flag should be checked after every call to saHpiEventGet().

If saHpiEventGet() is called with a timeout value other than SAHPI_TIMEOUT_IMMEDIATE, and the session is subsequently closed from another thread, this function will return with SA_ERR_HPI_INVALID_SESSION. If saHpiEventGet() is called with a timeout value other than SAHPI_TIMEOUT_IMMEDIATE, and an HPI User subsequently calls saHpiUnsubscribe() from another thread, this function will return with SA_ERR_HPI_INVALID_REQUEST.

6.5.4 saHpiEventAdd()

This function enables an HPI User to add events to the HPI domain identified by the *SessionId*. The domain controller processes an event added with this function as if the event originated from within the domain. The domain controller will attempt to publish events to all active event subscribers and will attempt to log events in the Domain Event Log, if room is available.

Prototype

```
SaErrorT SAHPI_API saHpiEventAdd (
    SAHPI_IN SaHpiSessionIdT      SessionId,
    SAHPI_IN SaHpiEventT          *EvtEntry
);
```

Parameters

SessionId - [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

EvtEntry - [in] Pointer to event to add to the domain. *Event* must be of type `SAHPI_ET_USER`, and the *Source* field must be `SAHPI_UNSPECIFIED_RESOURCE_ID`.

Return Value

`SA_OK` is returned if the event is successfully added to the domain; otherwise, an error code is returned.

`SA_ERR_HPI_INVALID_PARAMS` is returned if the:

- *EvtEntry* parameter is NULL.
- Event structure passed via the *EvtEntry* parameter is not an event of type `SAHPI_ET_USER` with the *Source* field being `SAHPI_UNSPECIFIED_RESOURCE_ID`.
- Event structure passed via the *EvtEntry* parameter has an invalid *Severity*.
- `SaHpiTextBufferT` structure passed as part of the User Event structure is not valid. This would occur when:
  - The *DataType* is not one of the enumerated values for that type, or
  - The data field contains characters that are not legal according to the value of *DataType*, or
  - The *Language* is not one of the enumerated values for that type when the *DataType* is `SAHPI_TL_TYPE_UNICODE` or `SAHPI_TL_TYPE_TEXT`.

`SA_ERR_HPI_INVALID_DATA` is returned if the event data does not meet implementation-specific restrictions on how much event data may be provided in a `SAHPI_ET_USER` event.

Remarks

Specific implementations of HPI may have restrictions on how much data may be included in a `SAHPI_ET_USER` event. If more event data is provided than can be processed, an error will be returned. The event data size restriction for the `SAHPI_ET_USER` event type is provided in the *UserEventMaxSize* field in the domain Event Log info structure. An HPI User should call the function `saHpiEventLogInfoGet()` to retrieve the Event Log info structure.

The domain controller will attempt to publish the event to all sessions within the domain with active event subscriptions; however, a session's event queue may overflow due to the addition of the new event.

The domain controller will attempt to log the event in the Domain Event Log; however, the Domain Event Log may overflow due to the addition of the new event.

6.6 Domain Alarm Table

The domain controller maintains a Domain Alarm Table (DAT) which contains entries for each active alarm in the domain. Alarms are added to and deleted from the DAT by the HPI implementation as the presence or absence of the corresponding conditions are detected by the domain controller. HPI functions are provided so that HPI Users may also add or delete entries in the DAT to reflect HPI User-detected alarm conditions.

There will be an entry in the DAT whenever one of the following conditions is present in the domain, or in a resource contained in the domain:

- Significant Asserted Sensor Event State,
- Significant Resource Failure,
- OEM Alarm Condition,
- User Defined Condition.

Each of these conditions is described in more detail below.

Significant Asserted Sensor Event State

A significant asserted sensor event state is any event state on a sensor that is asserted, provided that:

a) the sensor is enabled,
b) event generation for the sensor is enabled,
c) the sensor assertion event mask is configured so that the assertion of the event state causes an event to be generated, and
d) the event that is generated when the event state is asserted has a severity of `SAHPI_MINOR`, `SAHPI_MAJOR`, or `SAHPI_CRITICAL`.

A separate entry in the DAT will be present for each event state on each sensor in the domain for which the preceding conditions apply. HPI Users cannot remove these entries from the DAT; entries reflecting significant asserted sensor states will be automatically removed from the table when:

a) the event state is no longer asserted on the sensor,
b) a change in the configuration or status of the sensor is made such that the asserted event state would no longer constitute an alarm. For example, the sensor is disabled, sensor event generation is disabled, etc.
c) the resource is no longer present in the domain, or
d) the resource is marked as "Failed" in the domain.

Significant Resource Failure

A significant resource failure is any detected resource failure provided that the severity associated with the resource entry in the RPT is `SAHPI_MINOR`, `SAHPI_MAJOR`, or `SAHPI_CRITICAL`.

HPI Users cannot remove these entries from the DAT; entries reflecting significant resource failures will be automatically removed from the table when:

a) the resource is no longer failed,
b) the severity associated with the resource entry in the RPT is set to something other than `SAHPI_MINOR`, `SAHPI_MAJOR`, or `SAHPI_CRITICAL`, or
c) the resource associated with the alarm is removed from the domain.

Note: Note that resource failures that are indistinguishable from hot swap extractions do not result in alarms being added to the DAT. See Section 3.8 on page 26 for more information about resource failures.

OEM Alarm Condition

An HPI implementation may add OEM alarms to the DAT to reflect platform-specific alarm conditions.

a) HPI Users cannot remove these entries from the DAT; entries reflecting OEM alarm conditions will be automatically removed from the table by the HPI implementation when the alarm condition is cleared.

User Defined Condition

HPI Users may also add alarm entries to the DAT using the `saHpiAlarmAdd()` function described in Section 6.6.4 on page 72. By adding entries to the DAT, an HPI User can represent fault conditions of significant severity that are not detected by an HPI implementation. A User Alarm remains present in the DAT until a subsequent HPI User operation deletes it. HPI Users may only add alarm entries to the DAT with a severity of `SAHPI_MINOR`, `SAHPI_MAJOR` or `SAHPI_CRITICAL`.

An HPI User may manage the DAT using the HPI functions defined below. HPI User management of the DAT includes retrieving alarms for processing, acknowledging alarms, and adding or removing User Alarms. Alarms can be retrieved, acknowledged, and removed by *AlarmId* (single alarm), or by severity (group of alarms). For example, an HPI User can retrieve a list of all active alarms, or a list of all active critical alarms.

When alarms are automatically added to the DAT by the HPI implementation, the alarm entry in the DAT is initially flagged as "unacknowledged." An HPI User can change the state to "acknowledged" by calling `saHpiAlarmAcknowledge()`. When a User Alarm is added to the DAT, it may be initially added as either "acknowledged" or "unacknowledged."

In order to ensure that there is room in the DAT to store all non-User Alarms that may be required at any one time, an HPI implementation will generally need to impose a limit on the number of User Alarms that can be added to the DAT. This limit is defined in the Domain Info structure, and errors will be returned if an HPI User attempts to add more alarms than are permitted.

There is also a flag in the Domain Info structure indicating that one or more non-User Alarms that should be in the DAT are not present due to an overflow condition. However, the DAT should be sized, and User Alarms limited so that overflows of non-User Alarms will not occur except under very unusual circumstances.

When peer domains are defined, each peer will contain a DAT. Because the peer domains each contain the same set of resources, generally each peer domain's DAT will contain the same set of non-User Alarms reflecting alarm conditions in those resources. However, these alarms may not be identical in the peer domains. The *AlarmId* or acknowledged status may differ between the DATs. Additionally, entries in the DATs reflecting domain alarm conditions (e.g., a resource failure alarm indicating the inability to communicate with a particular resource) may differ among peer domains. Also, HPI User actions – adding or deleting User Alarms, acknowledging alarms – are specific to a particular DAT. If an HPI User adds, acknowledges, or deletes an alarm in the DAT of a domain, this does not automatically cause similar changes to the DAT in a peer domain.

6.6.1 saHpiAlarmGetNext()

This function allows retrieval of an alarm from the current set of alarms held in the Domain Alarm Table (DAT).

Prototype

```
SaErrorT SAHPI_API saHpiAlarmGetNext(
    SAHPI_IN SaHpiSessionIdT            SessionId,
    SAHPI_IN SaHpiSeverityT             Severity,
    SAHPI_IN SaHpiBoolT                 UnacknowledgedOnly,
    SAHPI_INOUT SaHpiAlarmT             *Alarm
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

Severity – [in] Severity level of alarms to retrieve. Set to `SAHPI_ALL_SEVERITIES` to retrieve alarms of any severity; otherwise, set to requested severity level.

UnacknowledgedOnly – [in] Set to True to indicate only unacknowledged alarms should be returned. Set to False to indicate either an acknowledged or unacknowledged alarm may be returned.

Alarm – [in/out] Pointer to the structure to hold the returned alarm entry. Also, on input, *Alarm->AlarmId* and *Alarm->Timestamp* are used to identify the previous alarm.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_INVALID_PARAMS` is returned when:

- *Severity* is not one of the valid enumerated values for this type.
- The *Alarm* parameter is passed in as NULL.

`SA_ERR_HPI_NOT_PRESENT` is returned:

- If there are no additional alarms in the DAT that meet the criteria specified by the *Severity* and *UnacknowledgedOnly* parameters:
- If the passed *Alarm->AlarmId* field was set to `SAHPI_FIRST_ENTRY` and there are no alarms in the DAT that meet the criteria specified by the *Severity* and *UnacknowledgedOnly* parameters.

`SA_ERR_HPI_INVALID_DATA` is returned if the passed *Alarm->AlarmId* matches an alarm in the DAT, but the passed *Alarm->Timestamp* does not match the timestamp of that alarm.

Remarks

All alarms contained in the DAT are maintained in the order in which they were added. This function will return the next alarm meeting the specifications given by an HPI User that was added to the DAT after the alarm whose *AlarmId* and *Timestamp* is passed by an HPI User, even if the alarm associated with the *AlarmId* and *Timestamp* has been deleted. If `SAHPI_FIRST_ENTRY` is passed as the *AlarmId*, the first alarm in the DAT meeting the specifications given by an HPI User is returned.

Alarm selection can be restricted to only alarms of a specified severity, and/or only unacknowledged alarms.

To retrieve all alarms contained within the DAT meeting specific requirements, call `saHpiAlarmGetNext()` with the *Alarm->AlarmId* field set to `SAHPI_FIRST_ENTRY` and the *Severity* and *UnacknowledgedOnly* parameters set to select what alarms should be returned. Then, repeatedly call `saHpiAlarmGetNext()` passing the previously returned alarm as the *Alarm* parameter, and the same values for *Severity* and *UnacknowledgedOnly* until the function returns with the error code `SA_ERR_HPI_NOT_PRESENT`.

`SAHPI_FIRST_ENTRY` and `SAHPI_LAST_ENTRY` are reserved *AlarmId* values, and will never be assigned to an alarm in the DAT.

The elements *AlarmId* and *Timestamp* are used in the *Alarm* parameter to identify the previous alarm; the next alarm added to the table after this alarm that meets the *Severity* and *UnacknowledgedOnly* requirements will be returned. *Alarm->AlarmId* may be set to `SAHPI_FIRST_ENTRY` to select the first alarm in the DAT meeting the *Severity* and *UnacknowledgedOnly* requirements. If *Alarm->AlarmId* is `SAHPI_FIRST_ENTRY`, then *Alarm->Timestamp* is ignored.

6.6.2 saHpiAlarmGet()

This function allows retrieval of a specific alarm in the Domain Alarm Table (DAT) by referencing its *AlarmId*.

Prototype

```
SaErrorT SAHPI_API saHpiAlarmGet(
    SAHPI_IN SaHpiSessionIdT            SessionId,
    SAHPI_IN SaHpiAlarmIdT              AlarmId,
    SAHPI_OUT SaHpiAlarmT               *Alarm
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

AlarmId – [in] *AlarmId* of the alarm to be retrieved from the DAT.

Alarm – [out] Pointer to the structure to hold the returned alarm entry.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_NOT_PRESENT` is returned if the requested *AlarmId* does not correspond to an alarm contained in the DAT.

`SA_ERR_HPI_INVALID_PARAMS` is returned when the *Alarm* parameter is passed in as NULL.

Remarks

`SAHPI_FIRST_ENTRY` and `SAHPI_LAST_ENTRY` are reserved *AlarmId* values, and will never be assigned to an alarm in the DAT.

6.6.3 saHpiAlarmAcknowledge()

This function allows an HPI User to acknowledge a single alarm entry or a group of alarm entries by severity.

Prototype

```
SaErrorT SAHPI_API saHpiAlarmAcknowledge(
    SAHPI_IN SaHpiSessionIdT            SessionId,
    SAHPI_IN SaHpiAlarmIdT              AlarmId,
    SAHPI_IN SaHpiSeverityT             Severity
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

AlarmId – [in] Identifier of the alarm to be acknowledged. Reserved *AlarmId* values:

- `SAHPI_ENTRY_UNSPECIFIED` Ignore this parameter.

Severity – [in] Severity level of alarms to acknowledge. Ignored unless *AlarmId* is `SAHPI_ENTRY_UNSPECIFIED`.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_NOT_PRESENT` is returned if an alarm entry identified by the *AlarmId* parameter does not exist in the DAT.

`SA_ERR_HPI_INVALID_PARAMS` is returned if *AlarmId* is `SAHPI_ENTRY_UNSPECIFIED` and *Severity* is not one of the valid enumerated values for this type.

Remarks

An HPI User acknowledges an alarm to indicate that it is aware of the alarm and to influence platform-specific alarm annunciation that may be provided by the implementation. Typically, an implementation ignores acknowledged alarms when announcing an alarm on annunciation devices such as audible sirens and dry contact closures. However, alarm annunciation is implementation-specific.

An acknowledged alarm will have the *Acknowledged* field in the alarm entry set to True.

Alarms are acknowledged by one of two ways: a single alarm entry by *AlarmId* regardless of severity or as a group of alarm entries by *Severity* regardless of *AlarmId*.

To acknowledge all alarms contained within the DAT, set the *Severity* parameter to `SAHPI_ALL_SEVERITIES`, and set the *AlarmId* parameter to `SAHPI_ENTRY_UNSPECIFIED`.

To acknowledge all alarms of a specific severity contained within the DAT, set the *Severity* parameter to the appropriate value, and set the *AlarmId* parameter to `SAHPI_ENTRY_UNSPECIFIED`.

To acknowledge a single alarm entry, set the *AlarmId* parameter to a value other than `SAHPI_ENTRY_UNSPECIFIED`. The *AlarmId* must be a valid identifier for an alarm entry present in the DAT at the time of the function call.

If an alarm has been previously acknowledged, acknowledging it again has no effect. However, this is not an error.

If the *AlarmId* parameter has a value other than `SAHPI_ENTRY_UNSPECIFIED`, the *Severity* parameter is ignored.

If the *AlarmId* parameter is passed as `SAHPI_ENTRY_UNSPECIFIED`, and no alarms are present that meet the requested *Severity*, this function will have no effect. However, this is not an error.

`SAHPI_ENTRY_UNSPECIFIED` is defined as the same value as `SAHPI_FIRST_ENTRY`, so using either symbol will have the same effect. However, `SAHPI_ENTRY_UNSPECIFIED` should be used with this function for clarity.

6.6.4 saHpiAlarmAdd()

This function is used to add a User Alarm to the DAT.

Prototype

```
SaErrorT SAHPI_API saHpiAlarmAdd(
    SAHPI_IN SaHpiSessionIdT            SessionId,
    SAHPI_INOUT SaHpiAlarmT             *Alarm
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

Alarm– [in/out] Pointer to the alarm entry structure that contains the new User Alarm to add to the DAT.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_INVALID_PARAMS` is returned when the *Alarm* pointer is passed in as NULL.

`SA_ERR_HPI_INVALID_PARAMS` is returned when *Alarm->Severity* is not one of the following enumerated values: `SAHPI_MINOR`, `SAHPI_MAJOR`, or `SAHPI_CRITICAL`.

`SA_ERR_HPI_INVALID_PARAMS` is returned when *Alarm->AlarmCond.Type* is not `SAHPI_STATUS_COND_TYPE_USER`.

`SA_ERR_HPI_OUT_OF_SPACE` is returned if the DAT is not able to add an additional User Alarm due to space limits or limits imposed on the number of User Alarms permitted in the DAT.

Remarks

The *AlarmId*, and *Timestamp* fields within the *Alarm* parameter are not used by this function. Instead, on successful completion, these fields are set to new values associated with the added alarm.

6.6.5 saHpiAlarmDelete()

This function allows an HPI User to delete a single User Alarm or a group of User Alarms from the DAT. Alarms may be deleted individually by specifying a specific *AlarmId*, or they may be deleted as a group by specifying a *Severity*.

Prototype

```
SaErrorT SAHPI_API saHpiAlarmDelete(
    SAHPI_IN SaHpiSessionIdT            SessionId,
    SAHPI_IN SaHpiAlarmIdT              AlarmId,
    SAHPI_IN SaHpiSeverityT             Severity
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

AlarmId – [in] Alarm identifier of the alarm entry to delete. Reserved values:

- `SAHPI_ENTRY_UNSPECIFIED` Ignore this parameter.

Severity – [in] Severity level of alarms to delete. Ignored unless *AlarmId* is `SAHPI_ENTRY_UNSPECIFIED`.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_INVALID_PARAMS` is returned if *AlarmId* is `SAHPI_ENTRY_UNSPECIFIED` and *Severity* is not one of the valid enumerated values for this type.

`SA_ERR_HPI_NOT_PRESENT` is returned if an alarm entry identified by the *AlarmId* parameter does not exist in the DAT.

`SA_ERR_HPI_READ_ONLY` is returned if the *AlarmId* parameter indicates a non-User Alarm.

Remarks

Only User Alarms added to the DAT can be deleted. When deleting alarms by severity, only User Alarms of the requested severity will be deleted.

To delete a single, specific alarm, set the *AlarmId* parameter to a value representing an actual User Alarm in the DAT. The *Severity* parameter is ignored when the *AlarmId* parameter is set to a value other than `SAHPI_ENTRY_UNSPECIFIED`.

To delete a group of User Alarms, set the *AlarmId* parameter to `SAHPI_ENTRY_UNSPECIFIED`, and set the *Severity* parameter to identify which severity of alarms should be deleted. To clear all User Alarms contained within the DAT, set the *Severity* parameter to `SAHPI_ALL_SEVERITIES`.

If the *AlarmId* parameter is passed as `SAHPI_ENTRY_UNSPECIFIED`, and no User Alarms are present that meet the specified *Severity*, this function will have no effect. However, this is not an error.

`SAHPI_ENTRY_UNSPECIFIED` is defined as the same value as `SAHPI_FIRST_ENTRY`, so using either symbol will have the same effect. However, `SAHPI_ENTRY_UNSPECIFIED` should be used with this function for clarity.

7 Resource Functions

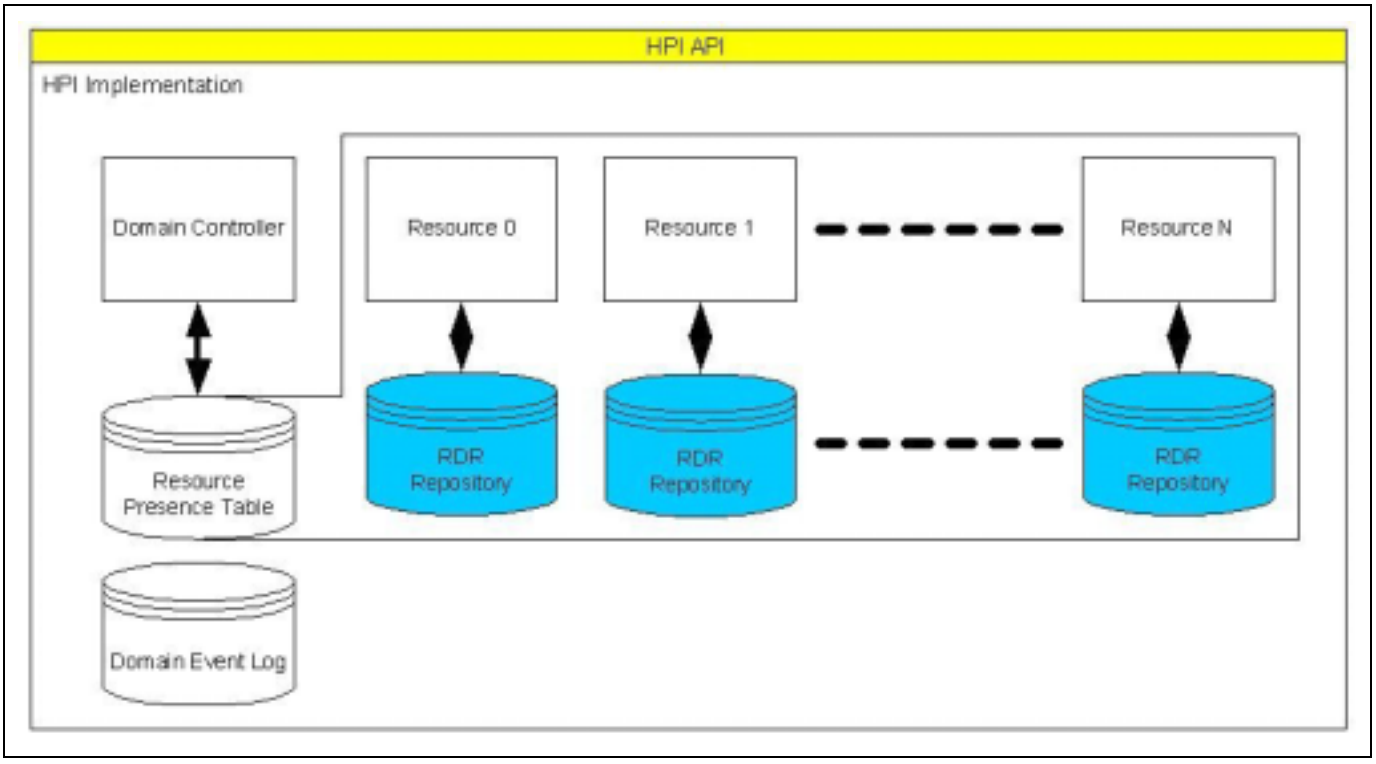
7.1 Resource Data Record (RDR) Repository Management

This set of HPI functions is used to access the Resource Data Record (RDR) repository for a specific resource. Every resource that contains any management instruments (sensors, controls, watchdog timers, inventory data repositories, and annunciators) must have an associated RDR (i.e., the "RDR Repository" capability, `SAHPI_CAPABILITY_RDR`, must be set in their corresponding RPT entries). Most resources will contain management instruments, so most will contain an RDR. The RDR repository holds information indicating the set of sensors, controls, watchdogs, inventory data repositories and annunciators for all of the entities that are managed by a resource. All sensors, controls, watchdogs, inventory data repositories and annunciators present in a resource must be specified in the resource's RDR repository.

The concept of RDRs provides for much of HPI's portability and extensibility across a multitude of hardware platform implementations. Because each platform will have a different population of domain controllers and resources, the RDR concept provides a means of discovering and managing these varied populations of hardware platforms and their management instruments. The RDR repository is used during discovery to learn the management capabilities of the resource.

The HPI model uses a distributed repository where each resource maintains a local repository of records.

Figure 7. Distributed Resource Data Record Repositories



At the resource level, the RDR repository is a logical database containing a collection of records that describe management instruments. Each RDR contains common fields that define the record type and naming information.

7.1.1 saHpiRdrGet()

This function returns a resource data record from the addressed resource.

Prototype

```
SaErrorT SAHPI_API saHpiRdrGet (
    SAHPI_IN  SaHpiSessionIdT       SessionId,
    SAHPI_IN  SaHpiResourceIdT      ResourceId,
    SAHPI_IN  SaHpiEntryIdT         EntryId,
    SAHPI_OUT SaHpiEntryIdT         *NextEntryId,
    SAHPI_OUT SaHpiRdrT             *Rdr
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

ResourceId – [in] Resource identified for this operation.

EntryId – [in] Identifier of the RDR entry to retrieve. Reserved *EntryId* values:

- `SAHPI_FIRST_ENTRY` Get first entry.
- `SAHPI_LAST_ENTRY` Reserved as delimiter for end of list. Not a valid entry identifier.

NextEntryId – [out] Pointer to location to store *EntryId* of next entry in RDR repository.

Rdr – [out] Pointer to the structure to receive the requested resource data record.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_CAPABILITY` is returned if the resource contains no RDR records (and thus does not have the `SAHPI_CAPABILITY_RDR` flag set in its RPT entry).

`SA_ERR_HPI_NOT_PRESENT` is returned if an *EntryId* (other than `SAHPI_FIRST_ENTRY`) is passed that does not correspond to an actual *EntryId* in the resource's RDR repository.

`SA_ERR_HPI_INVALID_PARAMS` is returned if:

- `SAHPI_LAST_ENTRY` is passed in to *EntryId*.
- *NextEntryId* pointer is passed in as NULL.
- *Rdr* pointer is passed in as NULL.

Remarks

Submitting an *EntryId* of `SAHPI_FIRST_ENTRY` results in the first RDR being read. A returned *NextEntryId* of `SAHPI_LAST_ENTRY` indicates the last RDR has been returned. A successful retrieval will include the next valid *EntryId*. To retrieve the entire list of RDRs, call this function first with an *EntryId* of `SAHPI_FIRST_ENTRY` and then use the returned *NextEntryId* in the next call. Proceed until the *NextEntryId* returned is `SAHPI_LAST_ENTRY`.

A resource's RDR repository is static over the lifetime of the resource; therefore no precautions are required against changes to the content of the RDR repository while it is being accessed.

7.1.2 saHpiRdrGetByInstrumentId()

This function returns the Resource Data Record (RDR) for a specific management instrument hosted by the addressed resource.

Prototype

```
SaErrorT SAHPI_API saHpiRdrGetByInstrumentId (
    SAHPI_IN  SaHpiSessionIdT        SessionId,
    SAHPI_IN  SaHpiResourceIdT       ResourceId,
    SAHPI_IN  SaHpiRdrTypeT          RdrType,
    SAHPI_IN  SaHpiInstrumentIdT     InstrumentId,
    SAHPI_OUT SaHpiRdrT              *Rdr
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

ResourceId – [in] Resource identified for this operation.

RdrType – [in] Type of RDR being requested.

InstrumentId – [in] Instrument number identifying the specific RDR to be returned. This is a sensor number, control number, watchdog timer number, IDR number, or annunciator number, depending on the value of the *RdrType* parameter.

Rdr – [out] Pointer to the structure to receive the requested RDR.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_CAPABILITY` is returned if the:

- Resource contains no RDR records (and thus does not have the `SAHPI_CAPABILITY_RDR` flag set in its RPT entry).
- Type of management instrument specified in the *RdrType* parameter is not supported by the resource, as indicated by the *Capability* field in its RPT entry.

`SA_ERR_HPI_NOT_PRESENT` is returned if the specific management instrument identified in the *InstrumentId* parameter is not present in the addressed resource.

`SA_ERR_HPI_INVALID_PARAMS` is returned when the:

- *RdrType* parameter is not a valid enumerated value for the type.
- *RdrType* is `SAHPI_NO_RECORD`.
- *Rdr* pointer is passed in as NULL.

Remarks

The RDR to be returned is identified by *RdrType* (sensor, control, watchdog timer, inventory data repository, or annunciator) and *InstrumentId* (sensor number, control number, watchdog number, IDR number, or annunciator number) for the specific management instrument for the RDR being requested.

7.2 Sensors

These functions are valid for resources that have the Sensor capability (`SAHPI_CAPABILITY_SENSOR`) set in their corresponding RPT entries.

Sensors may report a "Reading" (a value related to whatever it is that the sensor is measuring or monitoring) and up to 15 different "Event States." Sensors may monitor an individual condition, or they may report an aggregate of conditions as described in Section 7.2.3 on page 79.

7.2.1 Sensor Events and Sensor Event States

Each Event State is a single bit value that may be asserted or deasserted. The set of Event States a sensor may support is defined by the sensor's "Event Category." A particular sensor, however, does not have to support all the Event States defined for its event category. The specific Event States that a particular sensor supports are indicated in the "Events" field in the sensor's RDR.

A sensor is not required to support any Event States. If no Event States are supported by a sensor, the "Events" field will be 0x0000. A sensor may not support any Event States that are not defined for its event category.

Each Event State is independent, although in some event categories the meaning assigned to the Event States will imply that certain Event States will be mutually exclusive. For example, a sensor that uses the `SAHPI_EC_LIMIT` event category should only have one of the two Event States, `SAHPI_ES_LIMIT_NOT_EXCEEDED` or `SAHPI_ES_LIMIT_EXCEEDED` asserted at any one time.

Except where mutual exclusion is implied, however, sensors may have multiple Event States asserted simultaneously. For example, the event category `SAHPI_EC_THRESHOLD` uses six different event states to report the relationship between the value currently measured by the sensor and up to six different "threshold" values. Each threshold is independently examined, and each Event State that represents a threshold that has been "crossed" will be asserted. Threshold values must be configured so that "Minor", "Major" and "Critical" thresholds are increasingly extreme readings, such that when a "Major" threshold has been crossed, the corresponding "Minor" threshold will also have been crossed, and thus both Event States will be asserted.

When a sensor Event State is asserted or deasserted, an event may be generated by the resource. The generated event identifies the sensor and the event state being asserted or deasserted. The HPI implementation may assign event severity levels for each event state assertion or deassertion on an individual sensor basis. However, sensors that have event categories of `SAHPI_EC_THRESHOLD` or `SAHPI_EC_SEVERITY` must use these specific severities for these events:

Table 3. Event Severities for the Event Category SAHPI_EC_THRESHOLD

SAHPI_EC_THRESHOLD Event State	Severity for Assertion Event	Severity for Deassertion Event
SAHPI_ES_LOWER_MINOR	SAHPI_ MINOR	SAHPI_ MINOR
SAHPI_ES_LOWER_MAJOR	SAHPI_ MAJOR	SAHPI_ MAJOR
SAHPI_ES_LOWER_CRITICAL	SAHPI_ CRITICAL	SAHPI_ CRITICAL
SAHPI_ES_UPPER_MINOR	SAHPI_ MINOR	SAHPI_ MINOR
SAHPI_ES_UPPER_MAJOR	SAHPI_ MAJOR	SAHPI_ MAJOR
SAHPI_ES_UPPER_CRITICAL	SAHPI_ CRITICAL	SAHPI_ CRITICAL

Table 4. Event Severities for the Event Category SAHPI_EC_SEVERITY

SAHPI_EC_SEVERITY Event State	Severity for Assertion Event	Severity for Deassertion Event
SAHPI_ES_OK	SAHPI_OK	SAHPI_OK
SAHPI_ES_MINOR_FROM_OK	SAHPI_MINOR	SAHPI_MINOR

SAHPI_ES_MAJOR_FROM_LESS	SAHPI_MAJOR	SAHPI_MAJOR
SAHPI_ES_CRITICAL_FROM_LESS	SAHPI_CRITICAL	SAHPI_CRITICAL
SAHPI_ES_MINOR_FROM_MORE	SAHPI_MINOR	SAHPI_MINOR
SAHPI_ES_MAJOR_FROM_CRITICAL	SAHPI_MAJOR	SAHPI_MAJOR
SAHPI_ES_CRITICAL	SAHPI_CRITICAL	SAHPI_CRITICAL
SAHPI_ES_MONITOR	<defined by implementation>	<defined by implementation>
SAHPI_ES_INFORMATIONAL	SAHPI_INFORMATIONAL	SAHPI_INFORMATIONAL

Whether or not an event is generated when an Event State is asserted or deasserted is dependent on three status settings within the sensor.

1) Each sensor contains “event masks” for assertion and deassertion events. These masks indicate which Event State assertions or deassertions will result in events being generated. When an Event State is asserted, if the bit corresponding to that Event State is set in the sensor’s assert event mask, then an event will be generated. If the corresponding bit is not set in the sensor’s assert event mask, no event will be generated. Similarly, when an Event State is deasserted, an event will be generated if the corresponding bit is set in the sensor’s deassert event mask.

   Each sensor maintains separate assert and deassert event masks, but in resources that have the `SAHPI_CAPABILITY_EVT_DEASSERTS` capability set, the values for these two masks in a sensor will always be the same. Thus, any events generated when an Event State is asserted will be matched by events generated when the Event State is deasserted. For sensors in these resources, changing the assertion event mask automatically makes corresponding changes to the deassertion event mask.

2) Each sensor contains a “sensor event enable” status. No events will be generated when Event States change unless the sensor event enable status is “enabled.” Changing the “sensor event enable” status for a sensor does not change the settings of the assert and deassert event masks. Thus, an HPI User can temporarily disable all event generation for a sensor, then later re-enable event generation as per the settings in the assert and deassert event masks.

3) Each sensor contains a “sensor enable” status. If the sensor enable status is set to “disabled,” the sensor does not report a reading or Event States in response to the `saHpiSensorReadingGet()` function call, nor does it generate any events when Event States change.

When an Event State is asserted (or deasserted), if the corresponding bit in the sensor’s assert (or deassert) event mask is set, and the “sensor event enable” status is “enabled”, and the “sensor enable” status is “enabled,” then an event will be generated reporting the event state assertion or deassertion.

When a sensor is disabled (i.e., the “sensor enable” status for the sensor is set to “disabled”), the assertion/deassertion status of all event states supported by the sensor are undefined. When the sensor becomes enabled, an implementation may generate events, as configured by the sensor event enable and event mask settings, as each event state becomes defined. Alternatively, an implementation may establish the initial assertion/deassertion status of all event states upon becoming enabled. In this case, event state transitions will not occur, and events will not be generated for the already-existing event states.

HPI function calls can be used to change the assert and deassert event masks, as well as the settings of the “sensor event enable” and “sensor enable” status for the sensor. However, sensors may restrict the level of control available to an HPI User. These restrictions are advertised in the RDR for that sensor.

The changes made to a sensor configuration by an HPI User function are applicable to all HPI Users that have access to the sensor. Disabling event generation means that the sensor will not generate the disabled event at all – not just that the requesting HPI User will not see the event. Similarly, changing the threshold values for a particular sensor (with the `saHpiSensorThresholdsSet()` function) will change the thresholds used by the sensor as seen by all HPI Users.

Events are generated whenever the sensor enable status, the sensor event enable status, or the assert or deassert event mask for a sensor changes. The "Sensor Enable Change" event reports the new event enable configuration (sensor enable status, sensor event enable status, and assert/deassert event masks) and optionally reports the current Event State for the sensor. This event is only generated when an actual change is made to the sensor's enable status. For example, if an HPI User calls `saHpiSensorEventEnableSet()` to disable event generation for a sensor that already has events disabled, no event will be generated.

7.2.2 Sensor Configuration

Sensors contain several configuration parameters that may be set with the appropriate set sensor functions described below. Typically, these parameters are set to appropriate defaults by the HPI implementation. These "set" functions are available for management software to override these defaults.

Note: When the FRU managed by a resource is removed and reinserted, or the resource is otherwise re-initialized, the HPI implementation may reset these parameters. If the resource that hosts the sensor supports parameter control (see Section 7.8 on page 151), it may be possible to store the newly loaded parameter values in non-volatile storage so that the new settings will remain with the resource through removal, reinsertion, etc.

7.2.3 Aggregate Sensors

If a resource entry in the RPT has the capability bit of `SAHPI_CAPABILITY_AGGREGATE_STATUS` set it indicates that the resource includes three pre-defined sensors that report aggregate resource status. These sensors, if present, are reflected in the RDR table of the resource and are accessible with the APIs described in Sections 7.2.5 through 7.2.14, just like any other sensors hosted by the resource. The purpose of these sensors is to provide an HPI User with a simple way to detect overall operational, power, or temperature status of the entity managed by the resource. If the `SAHPI_CAPABILITY_AGGREGATE_STATUS` flag is set, all three of the predefined sensors described in Table 5 must be present in the resource.

Each resource, which supports aggregate resource status, must provide the following default sensors:

Table 5. Aggregate Resource Sensors

Sensor Number	Sensor Type	Sensor Category	Comment
SAHPI_DEFAGSENS_OPER (0x00000100)	SAHPI_OPERATIONAL	SAHPI_EC_ENABLE	Aggregate operational status of Resource
SAHPI_DEFAGSENS_PWR (0x00000101)	SAHPI_POWER_UNIT	SAHPI_EC_THRESHOLD	Aggregate power status of resource
SAHPI_DEFAGSENS_TEMP (0x00000102)	SAHPI_TEMPERATURE	SAHPI_EC_THRESHOLD	Aggregate thermal status of resource

Note: When one of the aggregate power/temperature sensors has a critical threshold status, the operational aggregate sensor should be set to "disabled".

7.2.4 Sensor Ranges

A range of named sensor numbers has been defined to be 0x00000100 through 0x000001FF. Implementations shall not assign sensor numbers within the named sensor number range, unless the associated sensors are specifically named by the specification.

Within this range, the sensor numbers from 0x00000100 to 0x0000010F are used exclusively for standard aggregate sensors. Currently three standard aggregate sensors are named above. Implementations shall not assign sensor numbers within the named sensor number range, unless the associated sensors are specifically named by the specification.

7.2.5 saHpiSensorReadingGet()

This function is used to retrieve a sensor reading.

Prototype

```
SaErrorT SAHPI_API saHpiSensorReadingGet (
    SAHPI_IN    SaHpiSessionIdT       SessionId,
    SAHPI_IN    SaHpiResourceIdT      ResourceId,
    SAHPI_IN    SaHpiSensorNumT       SensorNum,
    SAHPI_INOUT SaHpiSensorReadingT   *Reading,
    SAHPI_INOUT SaHpiEventStateT      *EventState
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

ResourceId – [in] Resource identified for this operation.

SensorNum – [in] Sensor number for which the sensor reading is being retrieved.

Reading – [in/out] Pointer to a structure to receive sensor reading values. If NULL, the sensor reading value will not be returned.

EventState – [in/out] Pointer to location to receive sensor event states. If NULL, the sensor event states will not be returned.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_CAPABILITY` is returned if the resource does not support sensors, as indicated by `SAHPI_CAPABILITY_SENSOR` in the resource's RPT entry.

`SA_ERR_HPI_INVALID_REQUEST` is returned if the sensor is currently disabled.

`SA_ERR_HPI_NOT_PRESENT` is returned if the sensor is not present.

Remarks

For sensors that return a type of `SAHPI_SENSOR_READING_TYPE_BUFFER`, the format of the returned data buffer is implementation-specific.

If the sensor does not provide a reading, the *Reading* structure returned by the `saHpiSensorReadingGet()` function will indicate the reading is not supported by setting the *IsSupported* flag to False.

If the sensor does not support any event states, a value of 0x0000 will be returned for the *EventState* value. This is indistinguishable from the return for a sensor that does support event states, but currently has no event states asserted. The Sensor RDR *Events* field can be examined to determine if the sensor supports any event states.

It is legal for both the *Reading* parameter and the *EventState* parameter to be NULL. In this case, no data is returned other than the return code. This can be used to determine if a sensor is present and enabled without actually returning current sensor data. If the sensor is present and enabled, `SA_OK` is returned; otherwise, an error code is returned.

7.2.6 saHpiSensorThresholdsGet()

This function retrieves the thresholds for the given sensor.

Prototype

```
SaErrorT SAHPI_API saHpiSensorThresholdsGet (
    SAHPI_IN  SaHpiSessionIdT            SessionId,
    SAHPI_IN  SaHpiResourceIdT           ResourceId,
    SAHPI_IN  SaHpiSensorNumT            SensorNum,
    SAHPI_OUT SaHpiSensorThresholdsT     *SensorThresholds
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

ResourceId – [in] Resource identified for this operation.

SensorNum – [in] Sensor number for which threshold values are being retrieved.

SensorThresholds – [out] Pointer to returned sensor thresholds.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_CAPABILITY` is returned if the resource does not support sensors, as indicated by `SAHPI_CAPABILITY_SENSOR` in the resource's RPT entry.

`SA_ERR_HPI_INVALID_PARAMS` is returned if the *SensorThresholds* pointer is passed in as NULL.

`SA_ERR_HPI_INVALID_CMD` is returned if:

- Getting a threshold on a sensor that is not a threshold type.
- The sensor does not have any readable threshold values.

`SA_ERR_HPI_NOT_PRESENT` is returned if the sensor is not present.

Remarks

This function only applies to sensors that support readable thresholds, as indicated by the *IsAccessible* field in the `SaHpiSensorThdDefnT` structure of the sensor's RDR being set to True and the *ReadThold* field in the same structure having a non-zero value.

For thresholds that do not apply to the identified sensor, the *IsSupported* flag of the threshold value field will be set to False.

7.2.7 saHpiSensorThresholdsSet()

This function sets the specified thresholds for the given sensor.

Prototype

```
SaErrorT SAHPI_API saHpiSensorThresholdsSet (
    SAHPI_IN  SaHpiSessionIdT          SessionId,
    SAHPI_IN  SaHpiResourceIdT         ResourceId,
    SAHPI_IN  SaHpiSensorNumT          SensorNum,
    SAHPI_IN  SaHpiSensorThresholdsT   *SensorThresholds
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

ResourceId – [in] Resource identified for this operation.

SensorNum – [in] Sensor number for which threshold values are being set.

SensorThresholds – [in] Pointer to the sensor thresholds values being set.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_INVALID_DATA` is returned if any of the threshold values are provided in a format not supported by the sensor.

`SA_ERR_HPI_CAPABILITY` is returned if the resource does not support sensors, as indicated by `SAHPI_CAPABILITY_SENSOR` in the resource's RPT entry.

`SA_ERR_HPI_NOT_PRESENT` is returned if the sensor is not present.

`SA_ERR_HPI_INVALID_CMD` is returned when:

- Writing to a threshold that is not writable.
- Setting a threshold on a sensor that is not a threshold type as indicated by the *IsAccessible* field of the `SaHpiSensorThdDefnT` structure.
- Setting a threshold outside of the Min-Max range as defined by the *Range* field of the *SensorDataFormat* of the RDR.

`SA_ERR_HPI_INVALID_DATA` is returned when:

- Thresholds are set out-of-order (see Remarks).
- A negative hysteresis value is provided.

Remarks

This function only applies to sensors that support writable thresholds, as indicated by the *IsAccessible* field in the `SaHpiSensorThdDefnT` structure of the sensor's RDR being set to True and the *WriteThold* field in the same structure having a non-zero value.

The type of value provided for each threshold setting must correspond to the reading format supported by the sensor, as defined by the reading type in the *DataFormat* field of the sensor's RDR (`saHpiSensorRecT`).

Sensor thresholds cannot be set outside of the range defined by the *Range* field of the *SensorDataFormat* of the Sensor RDR. If `SAHPI_SRF_MAX` indicates that a maximum reading exists, no sensor threshold may be set greater than the Max value. If `SAHPI_SRF_MIN` indicates that a minimum reading exists, no sensor threshold may be set less than the Min value.

Thresholds are required to be set progressively in-order, so that Upper Critical >= Upper Major >= Upper Minor >= Lower Minor >= Lower Major >= Lower Critical.

7.2.8 saHpiSensorTypeGet()

This function retrieves the sensor type and event category for the specified sensor.

Prototype

```
SaErrorT SAHPI_API saHpiSensorTypeGet (
    SAHPI_IN  SaHpiSessionIdT       SessionId,
    SAHPI_IN  SaHpiResourceIdT      ResourceId,
    SAHPI_IN  SaHpiSensorNumT       SensorNum,
    SAHPI_OUT SaHpiSensorTypeT      *Type,
    SAHPI_OUT SaHpiEventCategoryT   *Category
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

ResourceId – [in] Resource identified for this operation.

SensorNum – [in] Sensor number for which the type is being retrieved.

Type – [out] Pointer to returned enumerated sensor type for the specified sensor.

Category – [out] Pointer to location to receive the returned sensor event category.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_CAPABILITY` is returned if the resource does not support sensors, as indicated by `SAHPI_CAPABILITY_SENSOR` in the resource's RPT entry.

`SA_ERR_HPI_NOT_PRESENT` is returned if the sensor is not present.

`SA_ERR_HPI_INVALID_PARAMS` is returned if the:

- *Type* pointer is passed in as NULL.
- *Category* pointer is passed in as NULL.

Remarks

None.

7.2.9 saHpiSensorEnableGet()

This function returns the current sensor enable status for an addressed sensor.

Prototype

```
SaErrorT SAHPI_API saHpiSensorEnableGet (
    SAHPI_IN  SaHpiSessionIdT         SessionId,
    SAHPI_IN  SaHpiResourceIdT        ResourceId,
    SAHPI_IN  SaHpiSensorNumT         SensorNum,
    SAHPI_OUT SaHpiBoolT              *SensorEnabled
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

ResourceId – [in] Resource identified for this operation.

SensorNum – [in] Sensor number for which the sensor enable status is being requested.

SensorEnabled – [out] Pointer to the location to store the sensor enable status.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_CAPABILITY` is returned if the resource does not support sensors, as indicated by `SAHPI_CAPABILITY_SENSOR` in the resource's RPT entry.

`SA_ERR_HPI_INVALID_PARAMS` is returned when the *SensorEnabled* pointer is set to NULL.

`SA_ERR_HPI_NOT_PRESENT` is returned if the sensor is not present.

Remarks

The `SaHpiBoolT` value pointed to by the *SensorEnabled* parameter will be set to True if the sensor is enabled, or False if the sensor is disabled.

7.2.10 saHpiSensorEnableSet()

This function sets the sensor enable status for an addressed sensor.

Prototype

```
SaErrorT SAHPI_API saHpiSensorEnableSet (
    SAHPI_IN  SaHpiSessionIdT         SessionId,
    SAHPI_IN  SaHpiResourceIdT        ResourceId,
    SAHPI_IN  SaHpiSensorNumT         SensorNum,
    SAHPI_IN  SaHpiBoolT              SensorEnabled
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

ResourceId – [in] Resource identified for this operation.

SensorNum – [in] Sensor number for which the sensor enable status is being set.

SensorEnabled – [in] Sensor enable status to be set for the sensor.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_CAPABILITY` is returned if the resource does not support sensors, as indicated by `SAHPI_CAPABILITY_SENSOR` in the resource's RPT entry.

`SA_ERR_HPI_NOT_PRESENT` is returned if the sensor is not present.

`SA_ERR_HPI_READ_ONLY` is returned if the sensor does not support changing the enable status (i.e., the *EnableCtrl* field in the Sensor RDR is set to False).

Remarks

If a sensor is disabled, any calls to `saHpiSensorReadingGet()` for that sensor will return an error, and no events will be generated for the sensor.

Calling `saHpiSensorEnableSet()` with a *SensorEnabled* parameter of True will enable the sensor. A *SensorEnabled* parameter of False will disable the sensor.

If the sensor enable status changes as the result of this function call, an event will be generated.

7.2.11 saHpiSensorEventEnableGet()

This function returns the current sensor event enable status for an addressed sensor.

Prototype

```
SaErrorT SAHPI_API saHpiSensorEventEnableGet (
    SAHPI_IN  SaHpiSessionIdT         SessionId,
    SAHPI_IN  SaHpiResourceIdT        ResourceId,
    SAHPI_IN  SaHpiSensorNumT         SensorNum,
    SAHPI_OUT SaHpiBoolT              *SensorEventsEnabled
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

ResourceId – [in] Resource identified for this operation.

SensorNum – [in] Sensor number for which the sensor event enable status is being requested.

SensorEventsEnabled – [out] Pointer to the location to store the sensor event enable status.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_CAPABILITY` is returned if the resource does not support sensors, as indicated by `SAHPI_CAPABILITY_SENSOR` in the resource's RPT entry.

`SA_ERR_HPI_INVALID_PARAMS` is returned when the *SensorEventsEnabled* pointer is set to NULL.

`SA_ERR_HPI_NOT_PRESENT` is returned if the sensor is not present.

Remarks

The `SaHpiBoolT` value pointed to by the *SensorEventsEnabled* parameter will be set to True if event generation for the sensor is enabled, or False if event generation for the sensor is disabled.

7.2.12 saHpiSensorEventEnableSet()

This function sets the sensor event enable status for an addressed sensor.

Prototype

```
SaErrorT SAHPI_API saHpiSensorEventEnableSet (
    SAHPI_IN  SaHpiSessionIdT         SessionId,
    SAHPI_IN  SaHpiResourceIdT        ResourceId,
    SAHPI_IN  SaHpiSensorNumT         SensorNum,
    SAHPI_IN  SaHpiBoolT              SensorEventsEnabled
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

ResourceId – [in] Resource identified for this operation.

SensorNum – [in] Sensor number for which the sensor enable status is being set.

SensorEventsEnabled – [in] Sensor event enable status to be set for the sensor.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_CAPABILITY` is returned if the resource does not support sensors, as indicated by `SAHPI_CAPABILITY_SENSOR` in the resource's RPT entry.

`SA_ERR_HPI_NOT_PRESENT` is returned if the sensor is not present.

`SA_ERR_HPI_READ_ONLY` is returned if the sensor does not support changing the event enable status (i.e., the *EventCtrl* field in the Sensor RDR is set to `SAHPI_SEC_READ_ONLY`).

Remarks

If event generation for a sensor is disabled, no events will be generated as a result of the assertion or deassertion of any event state, regardless of the setting of the assert or deassert event masks for the sensor. If event generation for a sensor is enabled, events will be generated when event states are asserted or deasserted, according to the settings of the assert and deassert event masks for the sensor. Event states may still be read for a sensor even if event generation is disabled, by using the `saHpiSensorReadingGet()` function.

Calling `saHpiSensorEventEnableSet()` with a *SensorEventsEnabled* parameter of True will enable event generation for the sensor. A *SensorEventsEnabled* parameter of False will disable event generation for the sensor.

If the sensor event enabled status changes as a result of this function call, an event will be generated.

7.2.13 saHpiSensorEventMasksGet()

This function returns the assert and deassert event masks for a sensor.

Prototype

```
SaErrorT SAHPI_API saHpiSensorEventMasksGet (
    SAHPI_IN  SaHpiSessionIdT         SessionId,
    SAHPI_IN  SaHpiResourceIdT        ResourceId,
    SAHPI_IN  SaHpiSensorNumT         SensorNum,
    SAHPI_INOUT SaHpiEventStateT      *AssertEventMask,
    SAHPI_INOUT SaHpiEventStateT      *DeassertEventMask
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

ResourceId – [in] Resource identified for this operation.

SensorNum – [in] Sensor number for which the event enable configuration is being requested.

AssertEventMask – [in/out] Pointer to location to store sensor assert event mask. If NULL, assert event mask is not returned.

DeassertEventMask – [in/out] Pointer to location to store sensor deassert event mask. If NULL, deassert event mask is not returned.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_CAPABILITY` is returned if the resource does not support sensors, as indicated by `SAHPI_CAPABILITY_SENSOR` in the resource's RPT entry.

`SA_ERR_HPI_NOT_PRESENT` is returned if the sensor is not present.

Remarks

Two bit-mask values are returned by the `saHpiSensorEventMasksGet()` function; one for the sensor assert event mask, and one for the sensor deassert event mask. A bit set to '1' in the *AssertEventMask* value indicates that an event will be generated by the sensor when the corresponding event state for that sensor changes from deasserted to asserted. A bit set to '1' in the *DeassertEventMask* value indicates that an event will be generated by the sensor when the corresponding event state for that sensor changes from asserted to deasserted.

Events will only be generated by the sensor if the appropriate *AssertEventMask* or *DeassertEventMask* bit is set, sensor events are enabled, and the sensor is enabled.

For sensors hosted by resources that have the `SAHPI_CAPABILITY_EVT_DEASSERTS` flag set in its RPT entry, the *AssertEventMask* and the *DeassertEventMask* values will always be the same.

7.2.14 saHpiSensorEventMasksSet()

This function provides the ability to change the settings of the sensor assert and deassert event masks. Two parameters contain bit-mask values indicating which bits in the sensor assert and deassert event masks should be updated. In addition, there is an *Action* parameter.

Prototype

```
SaErrorT SAHPI_API saHpiSensorEventMasksSet (
    SAHPI_IN  SaHpiSessionIdT                 SessionId,
    SAHPI_IN  SaHpiResourceIdT                ResourceId,
    SAHPI_IN  SaHpiSensorNumT                 SensorNum,
    SAHPI_IN  SaHpiSensorEventMaskActionT     Action,
    SAHPI_IN  SaHpiEventStateT                AssertEventMask,
    SAHPI_IN  SaHpiEventStateT                DeassertEventMask
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

ResourceId – [in] Resource identified for this operation.

SensorNum – [in] Sensor number for which the event enable configuration is being set.

Action – [in] Enumerated value describing what change should be made to the sensor event masks:

- `SAHPI_SENS_ADD_EVENTS_TO_MASKS` – for each bit set in the *AssertEventMask* and *DeassertEventMask* parameters, set the corresponding bit in the sensor's assert and deassert event masks, respectively.
- `SAHPI_SENS_REMOVE_EVENTS_FROM_MASKS` – for each bit set in the *AssertEventMask* and *DeassertEventMask* parameters, clear the corresponding bit in the sensor's assert and deassert event masks, respectively.

AssertEventMask – [in] Bit mask or special value indicating which bits in the sensor's assert event mask should be set or cleared. (But see Remarks concerning resources with the `SAHPI_EVT_DEASSERTS_CAPABILITY` flag set.)

DeassertEventMask – [in] Bit mask or special value indicating which bits in the sensor's deassert event mask should be set or cleared. (But see Remarks concerning resources with the `SAHPI_EVT_DEASSERTS_CAPABILITY` flag set.)

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_CAPABILITY` is returned if the resource does not support sensors, as indicated by `SAHPI_CAPABILITY_SENSOR` in the resource's RPT entry.

`SA_ERR_HPI_INVALID_DATA` is returned if the *Action* parameter is `SAHPI_SENS_ADD_EVENTS_TO_MASKS`, and either of the *AssertEventMask* or *DeassertEventMask* parameters include a bit for an event state that is not supported by the sensor.

`SA_ERR_HPI_INVALID_PARAMS` is returned if the *Action* parameter is out of range.

`SA_ERR_HPI_NOT_PRESENT` is returned if the sensor is not present.

`SA_ERR_HPI_READ_ONLY` is returned if the sensor does not support updating the assert and deassert event masks (i.e., the *EventCtrl* field in the Sensor RDR is set to `SAHPI_SEC_READ_ONLY_MASKS` or `SAHPI_SEC_READ_ONLY`).

Remarks

The bits in the sensor assert and deassert event masks that correspond to '1' bits in the bit-mask parameters will be set or cleared, as indicated by the *Action* parameter. The bits in the sensor assert and deassert event masks corresponding to '0' bits in the bit-mask parameters will be unchanged.

Assuming that a sensor is enabled and event generation for the sensor is enabled, then for each bit set in the sensor's assert event mask, an event will be generated when the sensor's corresponding event state changes from deasserted to asserted. Similarly, for each bit set in the sensor's deassert event mask, an event will be generated when the sensor's corresponding event state changes from asserted to deasserted.

For sensors hosted by a resource that has the `SAHPI_CAPABILITY_EVT_DEASSERTS` flag set in its RPT entry, the assert and deassert event masks cannot be independently configured. When `saHpiSensorEventMasksSet()` is called for sensors in a resource with this capability, the *DeassertEventMask* parameter is ignored, and the *AssertEventMask* parameter is used to determine which bits to set or clear in both the assert event mask and deassert event mask for the sensor.

The *AssertEventMask* or *DeassertEventMask* parameter may be set to the special value, `SAHPI_ALL_EVENT_STATES`, indicating that all event states supported by the sensor should be added to or removed from, the corresponding sensor event mask.

If the sensor assert and/or deassert event masks change as a result of this function call, an event will be generated.

7.3 Controls

These functions are valid for resources, which have the Control capability (`SAHPI_CAPABILITY_CONTROL`) set in their corresponding RPT entries.

In HPI, controls have an associated state and mode. A control's state reflects how the control is currently set. For instance, an LED that was illuminated would reflect a state of on (`SAHPI_CTRL_STATE_ON`).

A control's mode describes how the control is managed. The control may be managed automatically by the implementation (auto mode). Or it may be managed by an HPI User (manual mode). Some controls allow their modes to be changed, allowing an HPI User to determine if they will manage the control, or relinquish the management to the implementation. But, other controls do not allow the mode to be changed. These static-mode controls are indicated with the *ReadOnly* flag set as part of the default control mode.

7.3.1 saHpiControlTypeGet()

This function retrieves the control type of a control object.

Prototype

```
SaErrorT SAHPI_API saHpiControlTypeGet (
    SAHPI_IN  SaHpiSessionIdT     SessionId,
    SAHPI_IN  SaHpiResourceIdT    ResourceId,
    SAHPI_IN  SaHpiCtrlNumT       CtrlNum,
    SAHPI_OUT SaHpiCtrlTypeT      *Type
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

ResourceId – [in] Resource identified for this operation.

CtrlNum – [in] Control number for which the type is being retrieved.

Type – [out] Pointer to `SaHpiCtrlTypeT` variable to receive the enumerated control type for the specified control.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_CAPABILITY` is returned if the resource does not support controls, as indicated by `SAHPI_CAPABILITY_CONTROL` in the resource's RPT entry.

`SA_ERR_HPI_NOT_PRESENT` is returned if the control is not present.

`SA_ERR_HPI_INVALID_PARAMS` is returned if the *Type* pointer is passed in as NULL.

Remarks

The *Type* parameter must point to a variable of type `SaHpiCtrlTypeT`. Upon successful completion, the enumerated control type is returned in the variable pointed to by *Type*.

7.3.2 saHpiControlGet()

This function retrieves the current state and mode of a control object.

Prototype

```
SaErrorT SAHPI_API saHpiControlGet (
    SAHPI_IN    SaHpiSessionIdT      SessionId,
    SAHPI_IN    SaHpiResourceIdT     ResourceId,
    SAHPI_IN    SaHpiCtrlNumT        CtrlNum,
    SAHPI_OUT   SaHpiCtrlModeT       *CtrlMode,
    SAHPI_INOUT SaHpiCtrlStateT      *CtrlState
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

ResourceId – [in] Resource identified for this operation.

CtrlNum – [in] Control number for which the state and mode are being retrieved.

CtrlMode – [out] Pointer to the mode of the control. If NULL, the control's mode will not be returned.

CtrlState – [in/out] Pointer to a control data structure into which the current control state will be placed. For text controls, the line number to read is passed in via *CtrlState->StateUnion.Text.Line*. If NULL, the control's state will not be returned.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_INVALID_CMD` is returned if the control is a write-only control, as indicated by the *WriteOnly* flag in the control's RDR (see remarks).

`SA_ERR_HPI_CAPABILITY` is returned if the resource does not support controls, as indicated by the `SAHPI_CAPABILITY_CONTROL` in the resource's RPT entry.

`SA_ERR_HPI_INVALID_DATA` is returned if the addressed control is a text control, and the line number passed in *CtrlState->StateUnion.Text.Line* does not exist in the control and is not `SAHPI_TLN_ALL_LINES`.

`SA_ERR_HPI_NOT_PRESENT` is returned if the control is not present.

Remarks

Note that the *CtrlState* parameter is both an input and an output parameter for this function. This is necessary to support line number inputs for text controls, as discussed below.

In some cases, the state of a control may be set, but the corresponding state cannot be read at a later time. Such controls are delineated with the *WriteOnly* flag in the Control's RDR.

Note that text controls are unique in that they have a state associated with each line of the control – the state being the text on that line. The line number to be read is passed in to `saHpiControlGet()` via *CtrlState->StateUnion.Text.Line*; the contents of that line of the control will be returned in *CtrlState->StateUnion.Text.Text*. The first line of the text control is line number "1".

If the line number passed in is `SAHPI_TLN_ALL_LINES`, then `saHpiControlGet()` will return the entire text of the control, or as much of it as will fit in a single `SaHpiTextBufferT`, in *CtrlState->StateUnion.Text.Text*. This value will consist of the text of all the lines concatenated, using the maximum number of characters for each line (no trimming of trailing blanks).

Note that depending on the data type and language, the text may be encoded in 2-byte Unicode, which requires two bytes of data per character.

Note that the number of lines and columns in a text control can be obtained from the control's Resource Data Record.

Write-only controls allow the control's state to be set, but the control state cannot be subsequently read. Such controls are indicated in the RDR, when the *WriteOnly* flag is set. `SA_ERR_HPI_INVALID_CMD` is returned when calling this function for a write-only control.

It is legal for both the *CtrlMode* parameter and the *CtrlState* parameter to be NULL. In this case, no data is returned other than the return code.

7.3.3 saHpiControlSet()

This function is used for setting the state and/or mode of the specified control object.

Prototype

```
SaErrorT SAHPI_API saHpiControlSet (
    SAHPI_IN SaHpiSessionIdT      SessionId,
    SAHPI_IN SaHpiResourceIdT     ResourceId,
    SAHPI_IN SaHpiCtrlNumT        CtrlNum,
    SAHPI_IN SaHpiCtrlModeT       CtrlMode,
    SAHPI_IN SaHpiCtrlStateT      *CtrlState
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

ResourceId – [in] Resource identified for this operation.

CtrlNum – [in] Control number for which the state and/or mode is being set.

CtrlMode – [in] The mode to set on the control.

CtrlState – [in] Pointer to a control state data structure holding the state to be set.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_CAPABILITY` is returned if the resource does not support controls, as indicated by the `SAHPI_CAPABILITY_CONTROL` in the resource's RPT entry.

`SA_ERR_HPI_NOT_PRESENT` is returned if the control is not present.

`SA_ERR_HPI_INVALID_DATA` is returned when the:

- *CtrlState->Type* field is not the correct type for the control identified by the *CtrlNum* parameter.
- *CtrlState->StateUnion.Analog* is out of range of the control record's analog Min and Max values.
- *CtrlState->StateUnion.Text.Text.DataLength*, combined with the *CtrlState->StateUnion.Text.Line*, overflows the remaining text control space.
- *CtrlState->StateUnion.Text.Text.DataType* is not set to the *DataType* specified in the RDR.
- *DataType* specified in the RDR is `SAHPI_TL_TYPE_UNICODE` or `SAHPI_TL_TYPE_TEXT` and *CtrlState->StateUnion.Text.Text.Language* is not set to the *Language* specified in the RDR.
- OEM control data is invalid (see remarks below).

`SA_ERR_HPI_INVALID_PARAMS` is returned if the:

- *CtrlMode* is not one of the valid enumerated values for this type.
- *CtrlMode* parameter is not `SAHPI_CTRL_MODE_AUTO` and the *CtrlState* pointer is passed in as NULL.
- *CtrlState->StateUnion.Digital* is not one of the valid enumerated values for this type.
- *CtrlState->StateUnion.Stream.StreamLength* is bigger than `SAHPI_CTRL_MAX_STREAM_LENGTH`.
- `SaHpiTextBufferT` structure passed as *CtrlState->StateUnion.Text.Text* contains text characters that are not allowed according to the value of *CtrlState->StateUnion.Text.Text.DataType*.

SA_ERR_HPI_INVALID_REQUEST is returned when SAHPI_CTRL_STATE_PULSE_ON is issued to a digital control, which is ON (in either manual or auto mode). It is also returned when SAHPI_CTRL_STATE_PULSE_OFF is issued to a digital control, which is OFF (in either manual or auto mode).

SA_ERR_HPI_READ_ONLY is returned when attempting to change the mode of a control with a read-only mode.

Remarks

When the mode is set to SAHPI_CTRL_MODE_AUTO, the state input is ignored. Ignored state inputs are not checked by the implementation.

The *CtrlState* parameter must be of the correct type for the specified control.

If the *CtrlMode* parameter is set to SAHPI_CTRL_MODE_AUTO, then the *CtrlState* parameter is not evaluated, and may be set to any value by an HPI User, including a NULL pointer. Text controls include a line number and a line of text in the *CtrlState* parameter, allowing update of just a single line of a text control. The first line of the text control is line number "1". If less than a full line of data is written, the control will clear all spaces beyond those written on the line. Thus writing a zero-length string will clear the addressed line. It is also possible to include more characters in the text passed in the *CtrlState* structure than will fit on one line; in this case, the control will wrap to the next line (still clearing the trailing characters on the last line written). Thus, there are two ways to write multiple lines to a text control: (a) call `saHpiControlSet()` repeatedly for each line, or (b) call `saHpiControlSet()` once and send more characters than will fit on one line. An HPI User should not assume any "cursor positioning" characters are available to use, but rather should always write full lines and allow "wrapping" to occur. When calling `saHpiControlSet()` for a text control, an HPI User may set the line number to SAHPI_TLN_ALL_LINES; in this case, the entire control will be cleared, and the data will be written starting on line 1. (This is different from simply writing at line 1, which only alters the lines written to.)

This feature may be used to clear the entire control, which can be accomplished by setting:

```
CtrlState->StateUnion.Text.Line = SAHPI_TLN_ALL_LINES;
CtrlState->StateUnion.Text.Text.DataLength = 0;
```

Note that the number of lines and columns in a text control can be obtained from the control's RDR.

The ManufacturerId (*MId*) field of an OEM control is ignored by the implementation when calling `saHpiControlSet()`.

On an OEM control, it is up to the implementation to determine what is invalid data, and to return the specified error code.

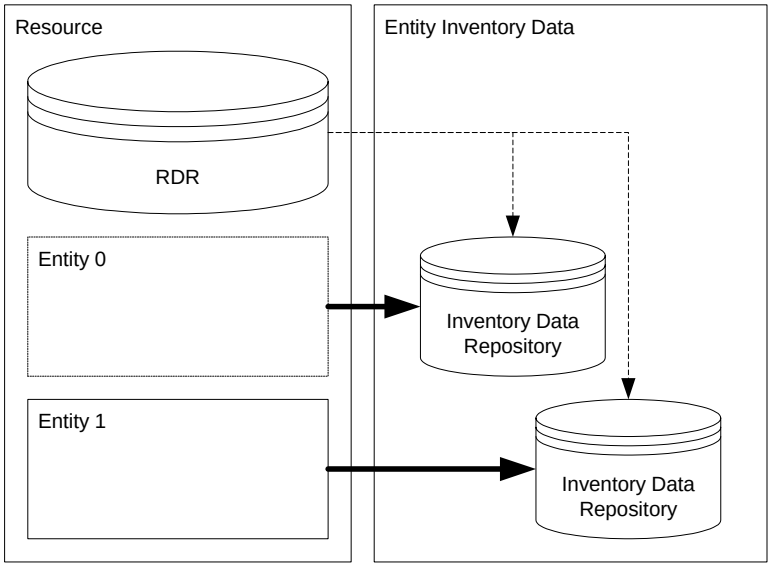
7.4 Inventory Data Repositories

Entity inventory data is descriptive information associated with a specific entity. This information may include serial numbers, part numbers, manufacturing dates, etc. The information is typically stored in the form of text strings and is grouped into categories such as Board Information, Product Information, etc. Inventory data is also extensible allowing storage of OEM and custom fields.

Entity inventory data is typically stored in some form of non-volatile storage, providing static data across system reboots and power cycles. However, there is no requirement for an HPI implementation to provide non-volatile storage.

A resource providing entity inventory data access will have the Inventory Data capability (`SAHPI_CAPABILITY_INVENTORY_DATA`) set in its corresponding RPT entry. Resources that support the Inventory Data capability contain one or more Inventory Data Repositories. An "Inventory Data Repository Record" in the RDR repository of the resource can be used to identify an available Inventory Data Repository.

Figure 8. IDR Association with Entity



Each resource that supports the Inventory Data capability must have an inventory data repository for the entity identified in the RPT entry for the resource (which will be the only, or the "primary" entity managed by this resource). The Inventory Data Repository for this entity must use the IDR identifier of `SAHPI_DEFAULT_INVENTORY_ID`. Additional entities managed by the resource may have their own Inventory Data Repositories with other identifiers.

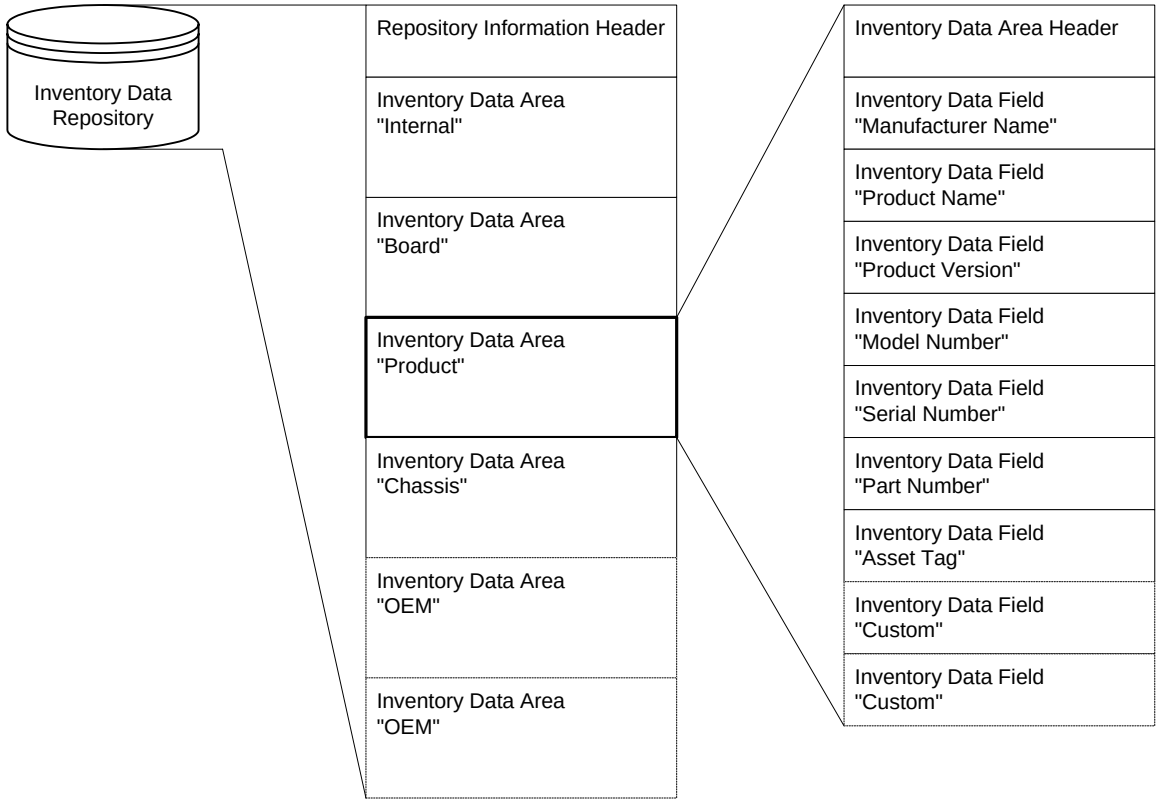
A category of inventory data is defined as an Inventory Data Area and a collection of Inventory Data Areas is defined as an Inventory Data Repository. An Inventory Data Repository is associated with a single entity as it contains information that can only be related to a single entity like a serial number.

An Inventory Data Repository (IDR) is composed of a Repository Information Header and one or more Inventory Data Areas. An Inventory Data Area is composed of an Area Header and one or more Inventory Data Fields.

An Inventory Data Repository may contain multiple areas of the same area type, like "OEM" areas. Inventory Data Areas may contain multiple fields of the same field type, like "custom" fields.

Inventory Data Area (IDA) identifiers are unique within a given IDR, meaning the *AreaId* will refer to only one area within a given IDR. Likewise, an Inventory Data Field identifier is unique within a given IDA, meaning the *FieldId* will refer to only one field within a given IDA. A *FieldId* is only unique within a given IDA, and may represent different fields within different Inventory Data Areas.

Figure 9. Depicted Layout of IDR



Functions are provided to retrieve and modify each element of an Inventory Data Repository. The "Get" functions allow management applications to read the Inventory Data Repository at the element level, and the "Add, Set, and Delete" functions allow updates of the individual repository elements.

The typical usage is to first locate the existence of the Inventory Data Repository by locating the Inventory Data Repository Record in the resource RDR Repository. Using the *IdrId* obtained from the Inventory Data Repository Record, retrieve the Inventory Data Repository Information, which will return the number of Inventory Data Areas contained within the Repository. Next retrieve the Inventory Data Area Headers for each Inventory Data Area in the Repository. The Area Header will contain the Area type, and the number of Inventory Data Fields contained with the area. Finally, retrieve the Inventory Data Fields contained within each Inventory Data Area. The Inventory Data Fields will identify the field type and include the field data in a standard `SaHpiTextBufferT` data structure. Areas can be added or removed from a given IDR. Fields can be added, modified, and removed from a given Area.

Modification of an IDR may be limited by the underlying storage implementation. An IDR implementation can be fixed in size and may have limited room or no room for extendibility. An IDR implementation can be read-only or have elements, which are read-only such as Areas or Fields. A fixed size IDR having limited extendibility may not allow additions of Areas and Fields. When an attempt is made to add an Area or Field, an appropriate error code is returned. An IDR with read-only elements does not allow additions, deletions or modifications of those elements. The read-only attribute may apply to a field, area, or the entire repository. When an attempt is made to modify a read-only element, an appropriate error code is returned.

The data format of Inventory Data Fields is implementation-specific. Specific implementations should provide documentation detailing the appropriate data formats for date and other specific fields. When an attempt is made to add or modify a field with incorrectly formatted data, an appropriate error code is returned.

7.4.1 saHpiIdrInfoGet()

This function returns the Inventory Data Repository information including the number of areas contained within the IDR and the update counter. The Inventory Data Repository is associated with a specific entity.

Prototype

```
SaErrorT SAHPI_API saHpiIdrInfoGet(
    SAHPI_IN SaHpiSessionIdT         SessionId,
    SAHPI_IN SaHpiResourceIdT        ResourceId,
    SAHPI_IN SaHpiIdrIdT             IdrId,
    SAHPI_OUT SaHpiIdrInfoT          *IdrInfo
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

ResourceId – [in] Resource identified for this operation.

IdrId – [in] Identifier for the Inventory Data Repository.

IdrInfo – [out] Pointer to the information describing the requested Inventory Data Repository.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_CAPABILITY` is returned if the resource does not support an inventory data repository, as indicated by `SAHPI_CAPABILITY_INVENTORY_DATA` in the resource's RPT entry.

`SA_ERR_HPI_NOT_PRESENT` is returned if the IDR is not present.

`SA_ERR_HPI_INVALID_PARAMS` is returned if the *IdrInfo* pointer is passed in as NULL.

Remarks

The update counter provides a means for insuring that no additions or changes are missed when retrieving the IDR data. In order to use this feature, an HPI User should invoke the `saHpiIdrInfoGet()`, and retrieve the update counter value before retrieving the first Area. After retrieving all Areas and Fields of the IDR, an HPI User should again invoke the `saHpiIdrInfoGet()` to get the update counter value. If the update counter value has not been incremented, no modification or additions were made to the IDR during data retrieval.

7.4.2 saHpiIdrAreaHeaderGet()

This function returns the Inventory Data Area header information for a specific area associated with a particular Inventory Data Repository.

Prototype

```
SaErrorT SAHPI_API saHpiIdrAreaHeaderGet(
    SAHPI_IN SaHpiSessionIdT          SessionId,
    SAHPI_IN SaHpiResourceIdT         ResourceId,
    SAHPI_IN SaHpiIdrIdT              IdrId,
    SAHPI_IN SaHpiIdrAreaTypeT        AreaType,
    SAHPI_IN SaHpiEntryIdT            AreaId,
    SAHPI_OUT SaHpiEntryIdT           *NextAreaId,
    SAHPI_OUT SaHpiIdrAreaHeaderT     *Header
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

ResourceId – [in] Resource identified for this operation.

IdrId – [in] Identifier for the Inventory Data Repository.

AreaType – [in] Type of Inventory Data Area.

AreaId – [in] Identifier of Area entry to retrieve from the IDR. Reserved *AreaId* values:

- `SAHPI_FIRST_ENTRY` Get first entry.
- `SAHPI_LAST_ENTRY` Reserved as a delimiter for end of list. Not a valid *AreaId*.

NextAreaId – [out] Pointer to location to store the *AreaId* of next area of the requested type within the IDR.

Header – [out] Pointer to Inventory Data Area Header into which the header information will be placed.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_CAPABILITY` is returned if the resource does not support an inventory data repository, as indicated by `SAHPI_CAPABILITY_INVENTORY_DATA` in the resource's RPT entry.

`SA_ERR_HPI_NOT_PRESENT` is returned if the:

- IDR is not present.
- *AreaType* parameter is set to `SAHPI_IDR_AREATYPE_UNSPECIFIED`, and the area specified by the *AreaId* parameter does not exist in the IDR.
- *AreaType* parameter is set to a specific area type, but an area matching both the *AreaId* parameter and *AreaType* parameter does not exist.

`SA_ERR_HPI_INVALID_PARAMS` is returned if:

- *AreaType* is not one of the valid enumerated values for this type.
- The *AreaId* is an invalid reserved value such as `SAHPI_LAST_ENTRY`.
- The *NextAreaId* pointer is passed in as NULL.
- The *Header* pointer is passed in as NULL.

Remarks

This function allows retrieval of an Inventory Data Area Header by one of two ways: by *AreaId* regardless of type or by *AreaType* and *AreaId*.

To retrieve all areas contained within an IDR, set the *AreaType* parameter to `SAHPI_IDR_AREATYPE_UNSPECIFIED`, and set the *AreaId* parameter to `SAHPI_FIRST_ENTRY` for the first call. For each subsequent call, set the *AreaId* parameter to the value returned in the *NextAreaId* parameter. Continue calling this function until the *NextAreaId* parameter contains the value `SAHPI_LAST_ENTRY`.

To retrieve areas of specific type within an IDR, set the *AreaType* parameter to a valid *AreaType* enumeration. Use the *AreaId* parameter in the same manner described above to retrieve all areas of the specified type. Set the *AreaId* parameter to `SAHPI_FIRST_ENTRY` to retrieve the first area of that type. Use the value returned in *NextAreaId* to retrieve the remaining areas of that type until `SAHPI_LAST_ENTRY` is returned.

7.4.3 saHpiIdrAreaAdd()

This function is used to add an Area to the specified Inventory Data Repository.

Prototype

```
SaErrorT SAHPI_API saHpiIdrAreaAdd(
    SAHPI_IN SaHpiSessionIdT          SessionId,
    SAHPI_IN SaHpiResourceIdT         ResourceId,
    SAHPI_IN SaHpiIdrIdT              IdrId,
    SAHPI_IN SaHpiIdrAreaTypeT        AreaType,
    SAHPI_OUT SaHpiEntryIdT           *AreaId
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

ResourceId – [in] Resource identified for this operation.

IdrId – [in] Identifier for the Inventory Data Repository.

AreaType – [in] Type of Area to add.

AreaId– [out] Pointer to location to store the *AreaId* of the newly created area.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_CAPABILITY` is returned if the resource does not support an inventory data repository, as indicated by `SAHPI_CAPABILITY_INVENTORY_DATA` in the resource's RPT entry.

`SA_ERR_HPI_NOT_PRESENT` is returned if the IDR is not present.

`SA_ERR_HPI_INVALID_DATA` is returned when attempting to add an area with an *AreaType* of `SAHPI_IDR_AREATYPE_UNSPECIFIED` or when adding an area that does not meet the implementation-specific restrictions.

`SA_ERR_HPI_OUT_OF_SPACE` is returned if the IDR does not have enough free space to allow the addition of the area.

`SA_ERR_HPI_INVALID_PARAMS` is returned if the:

- *AreaId* pointer is passed in as NULL.
- *AreaType* is not one of the valid enumerated values for this type.

`SA_ERR_HPI_READ_ONLY` is returned if the IDR is read-only and does not allow the addition of the area.

Remarks

On successful completion, the *AreaId* parameter will contain the *AreaId* of the newly created area.

On successful completion, the *ReadOnly* element in the new Area's header will always be False.

SAHPI_IDR_AREATYPE_UNSPECIFIED is not a valid area type, and should not be used with this function. If SAHPI_IDR_AREATYPE_UNSPECIFIED is specified as the area type, an error code of SA_ERR_HPI_INVALID_DATA is returned. This area type is only valid when used with the `saHpiIdrAreaHeaderGet()` function to retrieve areas of an unspecified type.

Some implementations may restrict the types of areas that may be added. These restrictions should be documented. SA_ERR_HPI_INVALID_DATA is returned when attempting to add an invalid area type.

7.4.4 saHpiIdrAreaDelete()

This function is used to delete an Inventory Data Area, including the Area header and all fields contained with the area, from a particular Inventory Data Repository.

Prototype

```
SaErrorT SAHPI_API saHpiIdrAreaDelete(
    SAHPI_IN SaHpiSessionIdT        SessionId,
    SAHPI_IN SaHpiResourceIdT       ResourceId,
    SAHPI_IN SaHpiIdrIdT            IdrId,
    SAHPI_IN SaHpiEntryIdT          AreaId
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

ResourceId – [in] Resource identified for this operation.

IdrId – [in] Identifier for the Inventory Data Repository.

AreaId – [in] Identifier of Area entry to delete from the IDR.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_CAPABILITY` is returned if the resource does not support an inventory data repository, as indicated by `SAHPI_CAPABILITY_INVENTORY_DATA` in the resource's RPT entry.

`SA_ERR_HPI_NOT_PRESENT` is returned if the:

- IDR is not present.
- Area identified by the *AreaId* parameter does not exist within the IDR.

`SA_ERR_HPI_INVALID_PARAMS` is returned when the *AreaId* is an invalid reserved value such as `SAHPI_LAST_ENTRY`.

`SA_ERR_HPI_READ_ONLY` is returned if the:

- IDA is read-only and therefore cannot be deleted.
- IDA contains a read-only *Field* and therefore cannot be deleted.
- IDR is read-only as deletions are not permitted for an area from a read-only IDR.

Remarks

Deleting an Inventory Data Area also deletes all fields contained within the area.

In some implementations, certain *Areas* are intrinsically read-only. The *ReadOnly* flag, in the area header, indicates if the *Area* is read-only.

If the Inventory Data Area is not read-only, but contains a *Field* that is read-only, the *Area* cannot be deleted. An attempt to delete an *Area* that contains a read-only *Field* will return an error.

7.4.5 saHpiIdrFieldGet()

This function returns the Inventory Data Field information from a particular Inventory Data Area and Repository.

Prototype

```
SaErrorT SAHPI_API saHpiIdrFieldGet(
    SAHPI_IN SaHpiSessionIdT         SessionId,
    SAHPI_IN SaHpiResourceIdT        ResourceId,
    SAHPI_IN SaHpiIdrIdT             IdrId,
    SAHPI_IN SaHpiEntryIdT           AreaId,
    SAHPI_IN SaHpiIdrFieldTypeT      FieldType,
    SAHPI_IN SaHpiEntryIdT           FieldId,
    SAHPI_OUT SaHpiEntryIdT          *NextFieldId,
    SAHPI_OUT SaHpiIdrFieldT         *Field
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

ResourceId – [in] Resource identified for this operation.

IdrId – [in] Identifier for the Inventory Data Repository.

AreaId – [in] Area identifier for the IDA.

FieldType – [in] Type of Inventory Data Field.

FieldId – [in] Identifier of Field to retrieve from the IDA. Reserved *FieldId* values:

- `SAHPI_FIRST_ENTRY` Get first entry.
- `SAHPI_LAST_ENTRY` Reserved as a delimiter for end of list. Not a valid *FieldId*.

NextFieldId – [out] Pointer to location to store the *FieldId* of next field of the requested type in the IDA.

Field – [out] Pointer to Inventory Data Field into which the field information will be placed.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_CAPABILITY` is returned if the resource does not support an inventory data repository, as indicated by `SAHPI_CAPABILITY_INVENTORY_DATA` in the resource's RPT entry.

`SA_ERR_HPI_NOT_PRESENT` is returned if the:

- IDR is not present.
- Area identified by *AreaId* is not present.
- *FieldType* parameter is set to `SAHPI_IDR_FIELDTYPE_UNSPECIFIED`, and the field specified by the *FieldId* parameter does not exist in the IDA.
- *FieldType* parameter is set to a specific field type, but a field matching both the *FieldId* parameter and *FieldType* parameter does not exist.

`SA_ERR_HPI_INVALID_PARAMS` is returned if:

- *FieldType* is not one of the valid enumerated values for this type.

- The *AreaId* or *FieldId* is an invalid reserved value such as `SAHPI_LAST_ENTRY`.
- The *NextFieldId* pointer is passed in as NULL.
- The *Field* pointer is passed in as NULL.

Remarks

This function allows retrieval of an Inventory Data Field by one of two ways: by *FieldId* regardless of type or by *FieldType* and *FieldId*.

To retrieve all fields contained within an IDA, set the *FieldType* parameter to `SAHPI_IDR_FIELDTYPE_UNSPECIFIED`, and set the *FieldId* parameter to `SAHPI_FIRST_ENTRY` for the first call. For each subsequent call, set the *FieldId* parameter to the value returned in the *NextFieldId* parameter. Continue calling this function until the *NextFieldId* parameter contains the value `SAHPI_LAST_ENTRY`.

To retrieve fields of a specific type within an IDA, set the *FieldType* parameter to a valid *Field* type enumeration. Use the *FieldId* parameter in the same manner described above to retrieve all fields of the specified type. Set the *FieldId* parameter to `SAHPI_FIRST_ENTRY` to retrieve the first field of that type. Use the value returned in *NextFieldId* to retrieve the remaining fields of that type until `SAHPI_LAST_ENTRY` is returned.

7.4.6 saHpiIdrFieldAdd()

This function is used to add a field to the specified Inventory Data Area with a specific Inventory Data Repository.

Prototype

```
SaErrorT SAHPI_API saHpiIdrFieldAdd(
    SAHPI_IN SaHpiSessionIdT          SessionId,
    SAHPI_IN SaHpiResourceIdT         ResourceId,
    SAHPI_IN SaHpiIdrIdT              IdrId,
    SAHPI_INOUT SaHpiIdrFieldT        *Field
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

ResourceId – [in] Resource identified for this operation.

IdrId – [in] Identifier for the Inventory Data Repository.

Field– [in/out] Pointer to Inventory Data Field, which contains the new field information to add to the IDA.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_CAPABILITY` is returned if the resource does not support an inventory data repository, as indicated by `SAHPI_CAPABILITY_INVENTORY_DATA` in the resource's RPT entry.

`SA_ERR_HPI_NOT_PRESENT` is returned if the:

- IDR is not present.
- Area identified by *Field→AreaId* does not exist within the IDR.

`SA_ERR_HPI_INVALID_DATA` is returned if the *Field* data is incorrectly formatted or does not meet the restrictions for the implementation.

`SA_ERR_HPI_OUT_OF_SPACE` is returned if the IDR does not have enough free space to allow the addition of the field.

`SA_ERR_HPI_READ_ONLY` is returned if the:

- Area identified by *Field→AreaId* is read-only and does not allow modification to its fields.
- IDR is identified by *IdrId*, is read-only, and does not allow modification to its fields.

`SA_ERR_HPI_INVALID_PARAMS` is returned if the:

- The *Field* type is not one of the valid field type enumerated values.
- Field type, *Field→Type*, is set to `SAHPI_IDR_FIELDTYPE_UNSPECIFIED.`
- `SaHpiTextBufferT` structure passed as part of the *Field* parameter is not valid. This occurs when:
  - The *DataType* is not one of the enumerated values for that type, or
  - The data field contains characters that are not legal according to the value of *DataType*, or

- The *Language* is not one of the enumerated values for that type when the *DataType* is SAHPI_TL_TYPE_UNICODE or SAHPI_TL_TYPE_TEXT.

- *Field* pointer is passed in as NULL.

Remarks

The *FieldId* element within the *Field* parameter is not used by this function. Instead, on successful completion, the *FieldId* field is set to the new value associated with the added *Field*.

The *ReadOnly* element in the *Field* parameter is not used by this function. On successful completion, the *ReadOnly* element in the new *Field* will always be False.

Addition of a read-only Inventory Data Field is not allowed; therefore the *ReadOnly* element in the SaHpiIdrFieldT parameter is ignored.

SAHPI_IDR_FIELDTYPE_UNSPECIFIED is not a valid field type, and should not be used with this function. If SAHPI_IDR_FIELDTYPE_UNSPECIFIED is specified as the field type, an error code of SA_ERR_HPI_INVALID_DATA is returned. This field type is only valid when used with the saHpiIdrFieldGet() function to retrieve fields of an unspecified type.

Some implementations have restrictions on what fields are valid in specific areas and/or the data format of some fields. These restrictions should be documented. SA_ERR_HPI_INVALID_DATA is returned when the field type or field data does not meet the implementation-specific restrictions.

The *AreaId* element within the *Field* parameter identifies the specific IDA into which the new field is added.

7.4.7 saHpiIdrFieldSet()

This function is used to update an Inventory Data Field for a particular Inventory Data Area and Repository.

Prototype

```
SaErrorT SAHPI_API saHpiIdrFieldSet(
    SAHPI_IN SaHpiSessionIdT          SessionId,
    SAHPI_IN SaHpiResourceIdT         ResourceId,
    SAHPI_IN SaHpiIdrIdT              IdrId,
    SAHPI_IN SaHpiIdrFieldT           *Field
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

ResourceId – [in] Resource identified for this operation.

IdrId – [in] Identifier for the Inventory Data Repository.

Field – [in] Pointer to Inventory Data Field, which contains the updated field information.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_CAPABILITY` is returned if the resource does not support an inventory data repository, as indicated by `SAHPI_CAPABILITY_INVENTORY_DATA` in the resource's RPT entry.

`SA_ERR_HPI_NOT_PRESENT` is returned if the:

- IDR is not present.
- Area identified by *Field→AreaId* does not exist within the IDR or if the *Field* does not exist within the Inventory Data Area.

`SA_ERR_HPI_INVALID_PARAMS` is returned if the:

- *Field* pointer is passed in as NULL.
- Field type, *Field→Type*, is not set to one of the valid field type enumerated values.
- Field type, *Field→Type*, is set to `SAHPI_IDR_FIELDTYPE_UNSPECIFIED`.

`SA_ERR_HPI_INVALID_DATA` is returned if the field data is incorrectly formatted or does not meet the restrictions for the implementation.

`SA_ERR_HPI_OUT_OF_SPACE` is returned if the IDR does not have enough free space to allow the modification of the field due to an increase in the field size.

`SA_ERR_HPI_READ_ONLY` is returned if the:

- Field is read-only and does not allow modifications.
- Area is read-only and does not allow modifications to its fields.
- IDR is read-only and does not allow modifications to its fields.

`SA_ERR_HPI_INVALID_PARAMS` is returned if the `SaHpiTextBufferT` structure passed as part of the *Field* parameter is not valid. This occurs when:

- The *DataType* is not one of the enumerated values for that type, or
- The data field contains characters that are not legal according to the value of *DataType*, or
- The *Language* is not one of the enumerated values for that type when the *DataType* is SAHPI_TL_TYPE_UNICODE or SAHPI_TL_TYPE_TEXT.

Remarks

This function allows modification of both the *FieldType* and *Field* data of a given Inventory Data Field. This function does not allow modification of the read-only attribute of the Inventory Data Field; therefore after a successful update, the *ReadOnly* element in the updated *Field* will always be False. The input value for *ReadOnly* is ignored.

SAHPI_IDR_FIELDTYPE_UNSPECIFIED is not a valid field type, and should not be used with this function. If SAHPI_IDR_FIELDTYPE_UNSPECIFIED is specified as the new field type, an error code of SA_ERR_HPI_INVALID_DATA is returned. This field type is only valid when used with the saHpiIdrFieldGet() function to retrieve fields of an unspecified type.

Some implementations have restrictions on what fields are valid in specific areas and/or the data format of some fields. These restrictions should be documented. SA_ERR_HPI_INVALID_DATA is returned when the field type or field data does not meet the implementation-specific restrictions.

In some implementations, certain *Fields* are intrinsically read-only. The *ReadOnly* flag, in the field structure, indicates if the *Field* is read-only.

The *Field* to update is identified by the *AreaId* and *FieldId* elements within the *Field* parameter.

7.4.8 saHpiIdrFieldDelete()

This function is used to delete an Inventory Data Field from a particular Inventory Data Area and Repository.

Prototype

```
SaErrorT SAHPI_API saHpiIdrFieldDelete(
    SAHPI_IN SaHpiSessionIdT          SessionId,
    SAHPI_IN SaHpiResourceIdT         ResourceId,
    SAHPI_IN SaHpiIdrIdT              IdrId,
    SAHPI_IN SaHpiEntryIdT            AreaId,
    SAHPI_IN SaHpiEntryIdT            FieldId
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

ResourceId – [in] Resource identified for this operation.

IdrId – [in] Identifier for the Inventory Data Repository.

AreaId – [in] Area identifier for the IDA.

FieldId – [in] Identifier of Field to delete from the IDA.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_CAPABILITY` is returned if the resource does not support an inventory data repository, as indicated by `SAHPI_CAPABILITY_INVENTORY_DATA` in the resource's RPT entry.

`SA_ERR_HPI_NOT_PRESENT` is returned if the:

- IDR is not present.
- Area identified by the *AreaId* parameter does not exist within the IDR, or if the Field identified by the *FieldId* parameter does not exist within the IDA.

`SA_ERR_HPI_READ_ONLY` is returned if the field, the IDA, or the IDR is read-only and does not allow deletion.

`SA_ERR_HPI_INVALID_PARAMS` is returned when the *AreaId* or *FieldId* is an invalid reserved value such as `SAHPI_LAST_ENTRY`.

Remarks

In some implementations, certain fields are intrinsically read-only. The *ReadOnly* flag, in the field structure, indicates if the field is read-only.

7.5 Watchdog Timers

Many high availability platforms contain watchdog timers to provide a means of monitoring the overall health of the software system. The HPI provides a standardized set of functions to access these watchdog timers. Additionally, the HPI provides a means to configure selected actions that will be taken when the watchdog timer expires. These actions include generating various sorts of interrupts, and performing power off, power cycle, and reset actions.

As a watchdog timer counts down and expires, up to two different actions may be performed: a “pre-timer interrupt” action and a “watchdog expiration” action. Each of these actions may be separately configured for the watchdog timer, and they may be set to both happen when the watchdog timer expires, or the pre-timer interrupt may be set to happen a fixed time interval before the watchdog timer expires.

The pre-timer interrupt action may be set to “NMI”, “SMI”, “Message Interrupt”, “OEM”, or “None”. Setting the pre-timer interrupt action to “None” must be supported on all platforms. The other actions are not necessarily supported and the specific action taken is dependent on the platform capabilities. An event will be generated when the pre-timer expires, unless the pre-timer interrupt action is “None” and the pre-timer interval is zero, in which case it is implementation-dependent whether or not an event is generated.

One of three watchdog timer expiration actions may be selected. These actions are “Reset”, “Power Down”, and “Power Cycle”. Not all of these options may be supported on all platforms, and what kind of reset action is taken (warm, cold, etc.) when “Reset” is selected is platform dependent. It is also possible to specify “No action,” meaning that no reset, power down, or power cycle is done when the timer expires. “No action” is required to be supported on all platforms. Even with no action configured, it is still possible to generate an HPI event, as described below.

The HPI watchdog timer function set provides a means for the calling software to get, set, and reset the watchdog. These functions are valid for resources that have the watchdog timer capability set in their corresponding RPT entries.

More than one watchdog timer may be supported per resource. Each of the watchdog timer APIs include a “watchdog number” parameter to address a specific timer accessed through that resource. If the RPT entry for a resource indicates that it supports watchdog timers, then there must be at least one watchdog timer hosted by the resource, with the watchdog number of `SAHPI_DEFAULT_WATCHDOG_NUM`. If additional watchdog timers are hosted by the resource, they may have any watchdog number, and may be located by watchdog records in the RDR repository. The watchdog records in the RDR repository identify which entity a particular watchdog timer is associated with. Power, interrupt, or reset actions will operate on that entity when the watchdog times out.

A watchdog timer is configured with the `saHpiWatchdogTimerSet()` function. An HPI User passes a watchdog timer structure that includes information on the expiration interval for the watchdog timer, the pre-timer interrupt interval, pre-timer and expiration actions, event generation support, and information identifying the current user of the watchdog timer.

The last of these items provides support for the fact that there may be multiple users of the watchdog over time. A timer use value is set whenever the watchdog is set, and separate flags are maintained indicating timer expiration while any particular timer use was active. There are no restrictions on the use of the timer use values, but several have pre-defined meanings. Care should be taken against using timer use values inconsistently with their pre-defined meanings because an HPI User may make assumptions about the meaning of a watchdog timeout based on these definitions.

Configuring the watchdog timer with the `saHpiWatchdogTimerSet()` function does not automatically start it running. Generally, a subsequent call to `saHpiWatchdogTimerReset()` is required to start the watchdog timer. However, if the watchdog timer is already running when the `saHpiWatchdogTimerSet()` function is called, it may be configured to continue running with the new configuration, without requiring a separate `saHpiWatchdogTimerReset()` function call.

If the Watchdog has been configured to issue a Pre-Timeout interrupt, and that interrupt has already occurred, the `saHpiWatchdogTimerReset()` function will not reset the watchdog counter. The only way to stop a Watchdog from timing out once a Pre-Timeout interrupt has occurred is to use the `saHpiWatchdogTimerSet()` function to reset and/or stop the timer.

A watchdog timer may be configured to generate events as it counts down past a pre-timer interrupt point and when it expires. Setting the *Log* field in the `SaHpiWatchdogT` structure to True enables event generation.

If event generation is configured and the pre-timer interrupt interval is non-zero, an event will be generated when the pre-timer interrupt point is reached before the watchdog timer expires. This event will have the *WatchdogAction* field set to `SAHPI_WAE_TIMER_INT` indicating that the pre-timeout interrupt point is reached and will include information on what pre-timer interrupt action is being taken in the *WatchdogPreTimerAction* field. The severity associated with a watchdog pre-timer interrupt event is `SAHPI_MAJOR` regardless of the pre-timer interrupt action.

If event generation is configured and the watchdog timer is not stopped, an event will be generated when the timer expires (in addition to a pre-timer interrupt event). This event will have the *WatchdogAction* field set to a value other than `SAHPI_WAE_TIMER_INT` indicating the expiration of the timer and the action taken upon expiration (may be `SAHPI_WAE_NO_ACTION`). When the *WatchdogAction* field is not `SAHPI_WAE_TIMER_INT`, the *WatchdogPreTimerAction* field is ignored. The severity associated with a watchdog timer expiration event when the action is `SAHPI_WAE_NO_ACTION` is `SAHPI_INFORMATIONAL`; otherwise the severity of the event is `SAHPI_MAJOR`.

If event generation is configured and the pre-timer interrupt interval is zero and the pre-timer interrupt action is not `SAHPI_WPI_NONE`, two events will be generated when the watchdog timer expires. If the pre-timer interrupt action is `SAHPI_WPI_NONE` and the pre-timer interrupt interval is zero, it is implementation-specific whether a pre-timer interrupt event is generated. The *WatchdogAction* field in the event structure determines the type of watchdog event; pre-timer interrupt or timer expiration. When the pre-timer action and timer action occur concurrently, there is no guarantee on the ordering of events or that the pre-timer interrupt action occurred before the timer action.

Refer to Appendix A for an example of watchdog usage.

7.5.1 saHpiWatchdogTimerGet()

This function retrieves the current watchdog timer settings and configuration.

Prototype

```
SaErrorT SAHPI_API saHpiWatchdogTimerGet (
    SAHPI_IN  SaHpiSessionIdT        SessionId,
    SAHPI_IN  SaHpiResourceIdT       ResourceId,
    SAHPI_IN  SaHpiWatchdogNumT      WatchdogNum,
    SAHPI_OUT SaHpiWatchdogT         *Watchdog
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

ResourceId – [in] Resource identified for this operation.

WatchdogNum – [in] Watchdog number that specifies the watchdog timer on a resource.

Watchdog – [out] Pointer to watchdog data structure.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_CAPABILITY` is returned if the resource does not support a watchdog timer, as indicated by `SAHPI_CAPABILITY_WATCHDOG` in the resource's RPT entry.

`SA_ERR_HPI_NOT_PRESENT` is returned if the *WatchdogNum* is not present.

`SA_ERR_HPI_INVALID_PARAMS` is returned when the *Watchdog* pointer is passed in as NULL.

Remarks

See the description of the `SaHpiWatchdogT` structure in Section 8.11 on page 180 for details on what information is returned by this function.

When the watchdog action is `SAHPI_WA_RESET`, the type of reset (warm or cold) is implementation-dependent.

7.5.2 saHpiWatchdogTimerSet()

This function provides a method for initializing the watchdog timer configuration.

Once the appropriate configuration has been set using `saHpiWatchdogTimerSet()`, an HPI User must then call `saHpiWatchdogTimerReset()` to initially start the watchdog timer, unless the watchdog timer was already running prior to calling `saHpiWatchdogTimerSet()` and the *Running* field in the passed *Watchdog* structure is True.

Prototype

```
SaErrorT SAHPI_API saHpiWatchdogTimerSet (
    SAHPI_IN SaHpiSessionIdT       SessionId,
    SAHPI_IN SaHpiResourceIdT      ResourceId,
    SAHPI_IN SaHpiWatchdogNumT     WatchdogNum,
    SAHPI_IN SaHpiWatchdogT        *Watchdog
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

ResourceId – [in] Resource identified for this operation.

WatchdogNum – [in] The watchdog number specifying the specific watchdog timer on a resource.

Watchdog – [in] Pointer to watchdog data structure.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_CAPABILITY` is returned if the resource does not support a watchdog timer, as indicated by `SAHPI_CAPABILITY_WATCHDOG` in the resource's RPT entry.

`SA_ERR_HPI_NOT_PRESENT` is returned if the *WatchdogNum* is not present.

`SA_ERR_HPI_INVALID_PARAMS` is returned when the:

- *Watchdog->TimerUse* is not one of the valid enumerated values for this type.
- *Watchdog->TimerAction* is not one of the valid enumerated values for this type.
- *Watchdog->PretimerInterrupt* is not one of the valid enumerated values for this type.
- *Watchdog* pointer is passed in as NULL.

`SA_ERR_HPI_INVALID_DATA` is returned when the:

- *Watchdog->PreTimeoutInterval* is outside the acceptable range for the implementation.
- *Watchdog->InitialCount* is outside the acceptable range for the implementation.
- Value of *Watchdog->PreTimeoutInterval* is greater than the value of *Watchdog->InitialCount.*
- *Watchdog->PretimerInterrupt* is set to an unsupported value. See remarks.
- *Watchdog->TimerAction* is set to an unsupported value. See remarks.
- *Watchdog->TimerUse* is set to an unsupported value. See remarks.

Remarks

Configuring the watchdog timer with the `saHpiWatchdogTimerSet()` function does not start the timer running. If the timer was previously stopped when this function is called, then it will remain stopped, and must be started by calling `saHpiWatchdogTimerReset()`. If the timer was previously running, then it will continue to run if the *Watchdog->Running* field passed is True, or will be stopped if the *Watchdog->Running* field passed is False. If it continues to run, it will restart its count-down from the newly configured initial count value.

If the initial counter value is set to 0, then any configured pre-timer interrupt action or watchdog timer expiration action will be taken immediately when the watchdog timer is started. This provides a mechanism for software to force an immediate recovery action should that be dependent on a Watchdog timeout occurring.

See the description of the `SaHpiWatchdogT` structure for more details on the effects of this command related to specific data passed in that structure.

Some implementations impose a limit on the acceptable ranges for the *PreTimeoutInterval* or *InitialCount*. These limitations must be documented. `SA_ERR_HPI_INVALID_DATA` is returned if an attempt is made to set a value outside of the supported range.

Some implementations cannot accept all of the enumerated values for *TimerUse*, *TimerAction*, or *PretimerInterrupt*. These restrictions should be documented. `SA_ERR_HPI_INVALID_DATA` is returned if an attempt is made to select an unsupported option.

When the watchdog action is set to `SAHPI_WA_RESET`, the type of reset (warm or cold) is implementation-dependent.

7.5.3 saHpiWatchdogTimerReset()

This function provides a method to start or restart the watchdog timer from the initial countdown value.

Prototype

```
SaErrorT SAHPI_API saHpiWatchdogTimerReset (
    SAHPI_IN SaHpiSessionIdT       SessionId,
    SAHPI_IN SaHpiResourceIdT      ResourceId,
    SAHPI_IN SaHpiWatchdogNumT     WatchdogNum
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

ResourceId – [in] Resource identified for this operation.

WatchdogNum – [in] The watchdog number specifying the specific watchdog timer on a resource.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_CAPABILITY` is returned if the resource does not support a watchdog timer, as indicated by `SAHPI_CAPABILITY_WATCHDOG` in the resource's RPT entry.

`SA_ERR_HPI_NOT_PRESENT` is returned if the *WatchdogNum* is not present.

`SA_ERR_HPI_INVALID_REQUEST` is returned if the current watchdog state does not allow a reset.

Remarks

If the Watchdog has been configured to issue a Pre-Timeout interrupt, and that interrupt has already occurred, the `saHpiWatchdogTimerReset()` function will not reset the watchdog counter. The only way to stop a Watchdog from timing out once a Pre-Timeout interrupt has occurred is to use the `saHpiWatchdogTimerSet()` function to reset and/or stop the timer.

7.6 Annunciators

This section describes the Annunciator Management Instrument and its uses.

A common requirement of high-availability systems is the ability to report fault conditions and other status information to system technicians, operators, or other supervising management systems, via some sort of display panels, LEDs, sirens, etc.

A platform may provide annunciation of fault conditions based on alarm entries in the DAT, as described in Section 3.2.1 on page 16 and in Section 6.6 on page 65. While this facility provides for complete platform-independence for an HPI User, it also limits the HPI User's ability to control what alarms are being announced, or how alarms are announced.

An HPI implementation may provide more control to an HPI User over what conditions are announced and how, while still maintaining a high degree of portability by defining Annunciator management instruments.

An Annunciator management instrument holds a set of individual *announcements.* The function of the Annunciator is to communicate the contents of its current set via whatever platform-specific mechanism is associated with that particular Annunciator management instrument. For example, if announcements are to be communicated via lighting a set of LEDs on a front-panel display, the Annunciator management instrument may analyze its current set of announcements and turn on a single LED reflecting the most severe condition found, or turn on a "System Ok" LED if there are currently no items in the set. A different Annunciator may continuously scroll each announcement in its set on a text display, as well as turning on LEDs and setting dry-contact relays to reflect the severity of conditions present. A third Annunciator may announce items in its set by sending messages to a proprietary management system, or by sending emails or pages to a system technician.

The Annunciator provides a common interface to these varied mechanisms for announcing conditions, so an HPI User is not burdened by platform-to-platform differences. However, the current content of any Annunciator management instrument is not defined by the HPI standard in the same way that the contents of the DAT are defined. Thus, the HPI implementation and HPI Users can exert more control over what conditions should be announced. Further, a platform can contain multiple Annunciator management instruments, each reflecting a different physical announcement device in the platform. By exposing each separately, HPI Users and the HPI implementation can control which conditions are handled by each announcement device.

An Annunciator management instrument may be implemented using other HPI controls that are in "auto" mode; for example, digital controls to turn LEDs on and off, stream controls to sound audible alerts, and/or text controls to display detailed information. However, Annunciators may also operate directly to report conditions using mechanisms that are not themselves visible directly in the HPI interface.

Over time, announcements are added to and deleted from an Annunciator's current set of announcements. This may be done automatically by the HPI implementation to reflect platform fault conditions, or by an HPI User via the HPI interface. When announcements are added or deleted automatically by the HPI implementation, it is implementation-specific which announcements are added or removed.

Each Annunciator management instrument has a current mode that indicates whether announcements are added and removed automatically by the HPI implementation, by an HPI User, or both. The mode may be set to one of three values, with the following meanings:

- `SAHPI_ANNUNCIATOR_MODE_AUTO` – the HPI implementation automatically adds and deletes announcements; an HPI User is not permitted to add or delete announcements.
- `SAHPI_ANNUNCIATOR_MODE_USER` – an HPI User may add and delete announcements; the HPI implementation will not automatically add or delete announcements.
- `SAHPI_ANNUNCIATOR_MODE_SHARED` – the HPI implementation automatically adds and deletes announcements, and an HPI User may also add and delete announcements.

The initial mode of each Annunciator is implementation-specific. An HPI User may change the mode of Annunciators with the `saHpiAnnunciatorModeSet()` function. However, the mode may be configured to be "Read-only", in which case an HPI User will not be able to change the mode.

When the mode is "User" or "Shared", HPI Users may add or delete any types of announcements in the Annunciator's current set – not just User announcements. This is allowed so that an HPI User can exert complete control over what conditions are being announced, if that is required. To distinguish between announcements added to an Annunciator automatically and those added by an HPI User, an *AddedByUser* field in the announcement indicates the source of the announcement in the set.

Each announcement in an Annunciator's current set contains a severity level, details describing the specific condition that is being reported, an *Acknowledged* flag, a timestamp indicating when the announcement was added to the set, and an *EntryId* that uniquely identifies the particular status item within the set.

EntryIds are assigned to announcements as they are added to the set as well as a *Timestamp*. After an announcement is deleted from the current set, its *EntryId* may be reused for a newly added announcement as long as the new announcement will have a different timestamp than any previously deleted announcement using the same *EntryId*. Thus, the *EntryId* and *Timestamp* together will uniquely identify any announcement, which was ever present in the Annunciator's set.

The actual meaning of the *Acknowledged* flag is arbitrary, and the actions taken by the platform when announcements are flagged as "acknowledged" or "unacknowledged" are implementation-specific. The intended use of the flag is to indicate whether a particular announcement in the current set has been recognized by whomever or whatever is inspecting the LEDs, displays, etc., that are being driven by the Annunciator management instrument. Thus, when announcements are added to the current set, generally the flag should be set to indicate that the condition is "unacknowledged". Later, either as the result of an HPI User function call, or due to some implementation-dependent action (such as pressing an "acknowledge" button on a front-panel display), the flag can be changed to indicate that the announcement is now "acknowledged."

The ability to acknowledge announcements is not controlled by the Annunciator "mode" setting (Auto, User, or Shared). Any announcement may be acknowledged by the HPI implementation, or via the `saHpiAnnunciatorAcknowledge()` function call, regardless of the current mode setting for the Annunciator.

Resources that contain Annunciator management instruments contain the `SAHPI_CAPABILITY_ANNUNCIATOR` capability flag in their RPT entries, and support the following functions. More than one Annunciator management instrument may be supported by a resource, if there are separate discrete announcing mechanisms controlled by the resource, each of which should potentially announce different sets of conditions.

Each Annunciator management instrument will have an RDR that identifies the particular annunciator, and provides fixed configuration information including the annunciator output type. The Entity Path in the Annunciator RDR should indicate the platform entity with which the Annunciator is associated. For example, if an Annunciator is designed to announce conditions related to a particular system chassis, then the entity path for that Annunciator should be the entity path for the system chassis. If an annunciator is not associated with any particular system entity, it may have an "empty" Entity Path (first entity type in the path is `SAHPI_ENT_ROOT`). Each Annunciator management instrument supported by a resource has a unique number, used to address that particular Annunciator in the following API functions.

7.6.1 saHpiAnnunciatorGetNext()

This function allows retrieval of an announcement from the current set of announcements held in the Annunciator.

Prototype

```
SaErrorT SAHPI_API saHpiAnnunciatorGetNext(
    SAHPI_IN SaHpiSessionIdT            SessionId,
    SAHPI_IN SaHpiResourceIdT           ResourceId,
    SAHPI_IN SaHpiAnnunciatorNumT       AnnunciatorNum,
    SAHPI_IN SaHpiSeverityT             Severity,
    SAHPI_IN SaHpiBoolT                 UnacknowledgedOnly,
    SAHPI_INOUT SaHpiAnnouncementT      *Announcement
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

ResourceId – [in] Resource identified for this operation.

AnnunciatorNum – [in] Annunciator number for the addressed Annunciator.

Severity – [in] Severity level of announcements to retrieve. Set to `SAHPI_ALL_SEVERITIES` to retrieve announcement of any severity; otherwise, set to requested severity level.

UnacknowledgedOnly – [in] Set to True to indicate only unacknowledged announcements should be returned. Set to False to indicate either an acknowledged or unacknowledged announcement may be returned.

Announcement – [in/out] Pointer to the structure to hold the returned announcement. Also, on input, *Announcement->EntryId* and *Announcement->Timestamp* are used to identify the "previous" announcement.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_CAPABILITY` is returned if the resource does not support Annunciators, as indicated by `SAHPI_CAPABILITY_ANNUNCIATOR` in the resource's RPT entry.

`SA_ERR_HPI_INVALID_PARAMS` is returned when *Severity* is not one of the valid enumerated values for this type.

`SA_ERR_HPI_NOT_PRESENT` is returned if:

- The *AnnunciatorNum* passed does not address a valid Annunciator supported by the resource.
- There are no announcements (or no unacknowledged announcements if *UnacknowledgedOnly* is True) that meet the specified *Severity* in the current set after the announcement identified by *Announcement->EntryId* and *Announcement->Timestamp*.
- There are no announcements (or no unacknowledged announcements if *UnacknowledgedOnly* is True) that meet the specified *Severity* in the current set if the passed *Announcement->EntryId* field was set to `SAHPI_FIRST_ENTRY`.

`SA_ERR_HPI_INVALID_PARAMS` is returned when the *Announcement* parameter is passed in as NULL.

`SA_ERR_HPI_INVALID_DATA` is returned if the passed *Announcement->EntryId* matches an announcement in the current set, but the passed *Announcement->Timestamp* does not match the timestamp of that announcement.

Remarks

All announcements contained in the set are maintained in the order in which they were added. This function will return the next announcement meeting the specifications given by an HPI User that was added to the set after the announcement whose *EntryId* and timestamp is passed by an HPI User. If `SAHPI_FIRST_ENTRY` is passed as the *EntryId*, the first announcement in the set meeting the specifications given by an HPI User is returned. This function should operate even if the announcement associated with the *EntryId* and *Timestamp* passed by an HPI User has been deleted.

Selection can be restricted to only announcements of a specified severity, and/or only unacknowledged announcements. To retrieve all announcements contained meeting specific requirements, call `saHpiAnnunciatorGetNext()` with the *Status->EntryId* field set to `SAHPI_FIRST_ENTRY` and the *Severity* and *UnacknowledgedOnly* parameters set to select what announcements should be returned. Then, repeatedly call `saHpiAnnunciatorGetNext()` passing the previously returned announcement as the *Announcement* parameter, and the same values for *Severity* and *UnacknowledgedOnly* until the function returns with the error code `SA_ERR_HPI_NOT_PRESENT`.

`SAHPI_FIRST_ENTRY` and `SAHPI_LAST_ENTRY` are reserved *EntryId* values, and will never be assigned to an announcement.

The elements *EntryId* and *Timestamp* are used in the *Announcement* parameter to identify the "previous" announcement; the next announcement added to the table after this announcement that meets the *Severity* and *UnacknowledgedOnly* requirements will be returned. *Announcement->EntryId* may be set to `SAHPI_FIRST_ENTRY` to select the first announcement in the current set meeting the *Severity* and *UnacknowledgedOnly* requirements. If *Announcement->EntryId* is `SAHPI_FIRST_ENTRY`, then *Announcement->Timestamp* is ignored.

7.6.2 saHpiAnnunciatorGet()

This function allows retrieval of a specific announcement in the Annunciator's current set by referencing its *EntryId*.

Prototype

```
SaErrorT SAHPI_API saHpiAnnunciatorGet(
    SAHPI_IN SaHpiSessionIdT            SessionId,
    SAHPI_IN SaHpiResourceIdT           ResourceId,
    SAHPI_IN SaHpiAnnunciatorNumT       AnnunciatorNum,
    SAHPI_IN SaHpiEntryIdT              EntryId,
    SAHPI_OUT SaHpiAnnouncementT        *Announcement
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

ResourceId – [in] Resource identified for this operation.

AnnunciatorNum – [in] Annunciator number for the addressed Annunciator.

EntryId – [in] Identifier of the announcement to retrieve from the Annunciator.

Announcement – [out] Pointer to the structure to hold the returned announcement.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_CAPABILITY` is returned if the resource does not support Annunciators, as indicated by `SAHPI_CAPABILITY_ANNUNCIATOR` in the resource's RPT entry.

`SA_ERR_HPI_NOT_PRESENT` is returned if:

- The *AnnunciatorNum* passed does not address a valid Annunciator supported by the resource.
- The requested *EntryId* does not correspond to an announcement contained in the Annunciator.

`SA_ERR_HPI_INVALID_PARAMS` is returned when the *Announcement* parameter is passed in as NULL.

Remarks

`SAHPI_FIRST_ENTRY` and `SAHPI_LAST_ENTRY` are reserved *EntryId* values, and will never be assigned to announcements.

7.6.3 saHpiAnnunciatorAcknowledge()

This function allows an HPI User to acknowledge a single announcement or a group of announcements by severity.

Prototype

```
SaErrorT SAHPI_API saHpiAnnunciatorAcknowledge(
    SAHPI_IN SaHpiSessionIdT            SessionId,
    SAHPI_IN SaHpiResourceIdT           ResourceId,
    SAHPI_IN SaHpiAnnunciatorNumT       AnnunciatorNum,
    SAHPI_IN SaHpiEntryIdT              EntryId,
    SAHPI_IN SaHpiSeverityT             Severity
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

ResourceId – [in] Resource identified for this operation.

AnnunciatorNum – [in] Annunciator number for the addressed Annunciator.

EntryId – [in] Entry identifier of the announcement to acknowledge. Reserved *EntryId* values:

- `SAHPI_ENTRY_UNSPECIFIED` Ignore this parameter.

Severity – [in] Severity level of announcements to acknowledge. Ignored unless *EntryId* is `SAHPI_ENTRY_UNSPECIFIED`.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_CAPABILITY` is returned if the resource does not support Annunciators, as indicated by `SAHPI_CAPABILITY_ANNUNCIATOR` in the resource's RPT entry.

`SA_ERR_HPI_NOT_PRESENT` is returned if:

- The *AnnunciatorNum* passed does not address a valid Annunciator supported by the resource.
- An announcement identified by the *EntryId* parameter does not exist in the current set.

`SA_ERR_HPI_INVALID_PARAMS` is returned if *EntryId* is `SAHPI_ENTRY_UNSPECIFIED` and *Severity* is not one of the valid enumerated values for this type.

Remarks

Announcements are acknowledged by one of two ways: a single announcement by *EntryId* regardless of severity or as a group of announcements by severity regardless of *EntryId*.

An HPI User acknowledges an announcement to influence the platform-specific annunciation provided by the Annunciator management instrument.

An acknowledged announcement will have the *Acknowledged* field set to True.

To acknowledge all announcements contained within the current set, set the *Severity* parameter to `SAHPI_ALL_SEVERITIES`, and set the *EntryId* parameter to `SAHPI_ENTRY_UNSPECIFIED`.

To acknowledge all announcements of a specific severity, set the *Severity* parameter to the appropriate value, and set the *EntryId* parameter to `SAHPI_ENTRY_UNSPECIFIED`.

To acknowledge a single announcement, set the *EntryId* parameter to a value other than `SAHPI_ENTRY_UNSPECIFIED`. The *EntryId* must be a valid identifier for an announcement present in the current set.

If an announcement has been previously acknowledged, acknowledging it again has no effect. However, this is not an error.

If the *EntryId* parameter has a value other than `SAHPI_ENTRY_UNSPECIFIED`, the *Severity* parameter is ignored.

If the *EntryId* parameter is passed as `SAHPI_ENTRY_UNSPECIFIED`, and no announcements are present that meet the requested *Severity*, this function will have no effect. However, this is not an error.

`SAHPI_ENTRY_UNSPECIFIED` is defined as the same value as `SAHPI_FIRST_ENTRY`, so using either symbol will have the same effect. However, `SAHPI_ENTRY_UNSPECIFIED` should be used with this function for clarity.

7.6.4 saHpiAnnunciatorAdd()

This function is used to add an announcement to the set of items held by an Annunciator management instrument.

Prototype

```
SaErrorT SAHPI_API saHpiAnnunciatorAdd(
    SAHPI_IN SaHpiSessionIdT            SessionId,
    SAHPI_IN SaHpiResourceIdT           ResourceId,
    SAHPI_IN SaHpiAnnunciatorNumT       AnnunciatorNum,
    SAHPI_INOUT SaHpiAnnouncementT      *Announcement
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

ResourceId – [in] Resource identified for this operation.

AnnunciatorNum – [in] Annunciator number for the addressed Annunciator.

Announcement – [in/out] Pointer to structure that contains the new announcement to add to the set.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_CAPABILITY` is returned if the resource does not support Annunciators, as indicated by `SAHPI_CAPABILITY_ANNUNCIATOR` in the resource's RPT entry.

`SA_ERR_HPI_NOT_PRESENT` is returned if the *AnnunciatorNum* passed does not address a valid Annunciator supported by the resource.

`SA_ERR_HPI_INVALID_PARAMS` is returned when:

- The *Announcement* pointer is passed in as NULL.
- The *Announcement->Severity* passed is not valid.
- The *Announcement->StatusCond* structure passed in is not valid.

`SA_ERR_HPI_OUT_OF_SPACE` is returned if the Annunciator is not able to add an additional announcement due to resource limits.

`SA_ERR_HPI_READ_ONLY` is returned if the Annunciator is in auto mode.

Remarks

The *EntryId*, *Timestamp*, and *AddedByUser* fields within the *Announcement* parameter are not used by this function. Instead, on successful completion, these fields are set to new values associated with the added announcement. *AddedByUser* will always be set to True.

7.6.5 saHpiAnnunciatorDelete()

This function allows an HPI User to delete a single announcement or a group of announcements from the current set of an Annunciator. Announcements may be deleted individually by specifying a specific *EntryId*, or they may be deleted as a group by specifying a severity.

Prototype

```
SaErrorT SAHPI_API saHpiAnnunciatorDelete(
    SAHPI_IN SaHpiSessionIdT            SessionId,
    SAHPI_IN SaHpiResourceIdT           ResourceId,
    SAHPI_IN SaHpiAnnunciatorNumT       AnnunciatorNum,
    SAHPI_IN SaHpiEntryIdT              EntryId,
    SAHPI_IN SaHpiSeverityT             Severity
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

ResourceId – [in] Resource identified for this operation.

AnnunciatorNum – [in] Annunciator number for the addressed Annunciator.

EntryId – [in] Entry identifier of the announcement to delete. Reserved *EntryId* values:

- `SAHPI_ENTRY_UNSPECIFIED` Ignore this parameter.

Severity – [in] Severity level of announcements to delete. Ignored unless *EntryId* is `SAHPI_ENTRY_UNSPECIFIED`.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_CAPABILITY` is returned if the resource does not support Annunciators, as indicated by `SAHPI_CAPABILITY_ANNUNCIATOR` in the resource's RPT entry.

`SA_ERR_HPI_INVALID_PARAMS` is returned if *EntryId* is `SAHPI_ENTRY_UNSPECIFIED` and *Severity* is not one of the valid enumerated values for this type.

`SA_ERR_HPI_NOT_PRESENT` is returned if:

- The *AnnunciatorNum* passed does not address a valid Annunciator supported by the resource
- An announcement identified by the *EntryId* parameter does not exist in the current set of the Annunciator.

`SA_ERR_HPI_READ_ONLY` is returned if the Annunciator is in auto mode.

Remarks

To delete a single, specific announcement, set the *EntryId* parameter to a value representing an actual announcement in the current set. The *Severity* parameter is ignored when the *EntryId* parameter is set to a value other than `SAHPI_ENTRY_UNSPECIFIED`.

To delete a group of announcements, set the *EntryId* parameter to `SAHPI_ENTRY_UNSPECIFIED`, and set the *Severity* parameter to identify which severity of announcements should be deleted. To clear all announcements contained within the Annunciator, set the *Severity* parameter to `SAHPI_ALL_SEVERITIES`.

If the *EntryId* parameter is passed as `SAHPI_ENTRY_UNSPECIFIED`, and no announcements are present that meet the specified *Severity*, this function will have no effect. However, this is not an error.

`SAHPI_ENTRY_UNSPECIFIED` is defined as the same value as `SAHPI_FIRST_ENTRY`, so using either symbol will have the same effect. However, `SAHPI_ENTRY_UNSPECIFIED` should be used with this function for clarity.

7.6.6 saHpiAnnunciatorModeGet()

This function allows an HPI User to retrieve the current operating mode of an Annunciator. The mode indicates whether or not an HPI User is allowed to add or delete items in the set, and whether or not the HPI implementation will automatically add or delete items in the set.

Prototype

```
SaErrorT SAHPI_API saHpiAnnunciatorModeGet(
    SAHPI_IN SaHpiSessionIdT            SessionId,
    SAHPI_IN SaHpiResourceIdT           ResourceId,
    SAHPI_IN SaHpiAnnunciatorNumT       AnnunciatorNum,
    SAHPI_OUT SaHpiAnnunciatorModeT     *Mode
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

ResourceId – [in] Resource identified for this operation.

AnnunciatorNum – [in] Annunciator number for the addressed Annunciator.

Mode – [out] Pointer to location to store the current operating mode of the Annunciator where the returned value will be one of the following:

- `SAHPI_ANNUNCIATOR_MODE_AUTO` – the HPI implementation may add or delete announcements in the set; an HPI User may not add or delete announcements in the set.
- `SAHPI_ANNUNCIATOR_MODE_USER` – the HPI implementation may not add or delete announcements in the set; an HPI User may add or delete announcements in the set.
- `SAHPI_ANNUNCIATOR_MODE_SHARED` – both the HPI implementation and an HPI User may add or delete announcements in the set.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_CAPABILITY` is returned if the resource does not support Annunciators, as indicated by `SAHPI_CAPABILITY_ANNUNCIATOR` in the resource's RPT entry.

`SA_ERR_HPI_NOT_PRESENT` is returned if the *AnnunciatorNum* passed does not address a valid Annunciator supported by the resource.

`SA_ERR_HPI_INVALID_PARAMS` is returned if *Mode* is passed in as NULL.

Remarks

None.

7.6.7 saHpiAnnunciatorModeSet()

This function allows an HPI User to change the current operating mode of an Annunciator. The mode indicates whether or not an HPI User is allowed to add or delete announcements in the set, and whether or not the HPI implementation will automatically add or delete announcements in the set.

Prototype

```
SaErrorT SAHPI_API saHpiAnnunciatorModeSet(
    SAHPI_IN SaHpiSessionIdT            SessionId,
    SAHPI_IN SaHpiResourceIdT           ResourceId,
    SAHPI_IN SaHpiAnnunciatorNumT       AnnunciatorNum,
    SAHPI_IN SaHpiAnnunciatorModeT      Mode
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

ResourceId – [in] Resource identified for this operation.

AnnunciatorNum – [in] Annunciator number for the addressed Annunciator.

Mode – [out] Mode to set for the Annunciator:

- `SAHPI_ANNUNCIATOR_MODE_AUTO` – the HPI implementation may add or delete announcements in the set; an HPI User may not add or delete announcements in the set.
- `SAHPI_ANNUNCIATOR_MODE_USER` – the HPI implementation may not add or delete announcements in the set; an HPI User may add or delete announcements in the set.
- `SAHPI_ANNUNCIATOR_MODE_SHARED` – both the HPI implementation and an HPI User may add or delete announcements in the set.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_CAPABILITY` is returned if the resource does not support Annunciators, as indicated by `SAHPI_CAPABILITY_ANNUNCIATOR` in the resource's RPT entry.

`SA_ERR_HPI_NOT_PRESENT` is returned if the *AnnunciatorNum* passed does not address a valid Annunciator supported by the resource.

`SA_ERR_HPI_INVALID_PARAMS` is returned if *Mode* is not a valid enumerated value for this type.

`SA_ERR_HPI_READ_ONLY` is returned if mode changing is not permitted for this Annunciator.

Remarks

None.

7.7 Managed Hot Swap

Resources which support standard hot swap events and functions will have the FRU capability (`SAHPI_CAPABILITY_FRU`) set, and possibly the Managed Hot Swap capability (`SAHPI_CAPABILITY_MANAGED_HOTSWAP`) set in their RPT entry. Including the FRU capability indicates that the resource may be inserted or removed during normal operation, and that hot swap events will be issued when insertions or removals occur. Including the Managed Hot Swap capability indicates that the resource follows the full hot swap model and usage described in this section and can be managed using the functions defined below. A resource with the FRU capability set but without the Managed Hot Swap capability set follows the simplified hot swap model described in this section. Table 6 summarizes the meaning of these two capability flags.

Table 6. Hot Swap Capabilities

Capability Flags	Hot Swap Model Supported	Hot Swap Events Issued	Hot Swap Functions Supported
SAHPI_CAPABILITY_FRU=0 SAHPI_CAPABILITY_MANAGED_HOTSWAP=0	None	None	None
SAHPI_CAPABILITY_FRU=1 SAHPI_CAPABILITY_MANAGED_HOTSWAP=0	Simplified	For transition between Active- and Not-Present states only	None
SAHPI_CAPABILITY_FRU=1 SAHPI_CAPABILITY_MANAGED_HOTSWAP=1	Full	For transition between any states in the full hot-swap model	All
SAHPI_CAPABILITY_FRU=0 SAHPI_CAPABILITY_MANAGED_HOTSWAP=1	Illegal configuration	N/A	N/A

When a FRU entity is inserted into or removed from the system, the corresponding resource is added to or removed from all domains in which that resource is visible, and the RPTs in those domains are updated accordingly.

The hot swap model describes the behavior of a resource as it enters and exits an HPI domain and the associated events that it (or the domain controller) generates to indicate state changes. Figure 10 depicts the full hot swap model defined by HPI. In the abstracted hot swap model, the hot swap functions are directed to the resource associated with the actual FRU being inserted or removed and not to the container of the FRU (such as a slot in CompactPCI). The hot swap model is not intended to imply an implementation, but is intended to support existing hot swap implementations.

A resource may follow the full hot swap model using all five states, or it may follow a simplified model shown in Figure 11 and transition between the NOT PRESENT and ACTIVE states. The full model is appropriate for resources associated with FRUs that require processing to occur as they are inserted or removed from the system. The simplified model is appropriate for resources associated with FRUs that do not require insertion or removal processing, and simply need to alert the management software of their comings and goings.

Figure 10. Full Hot Swap State Model

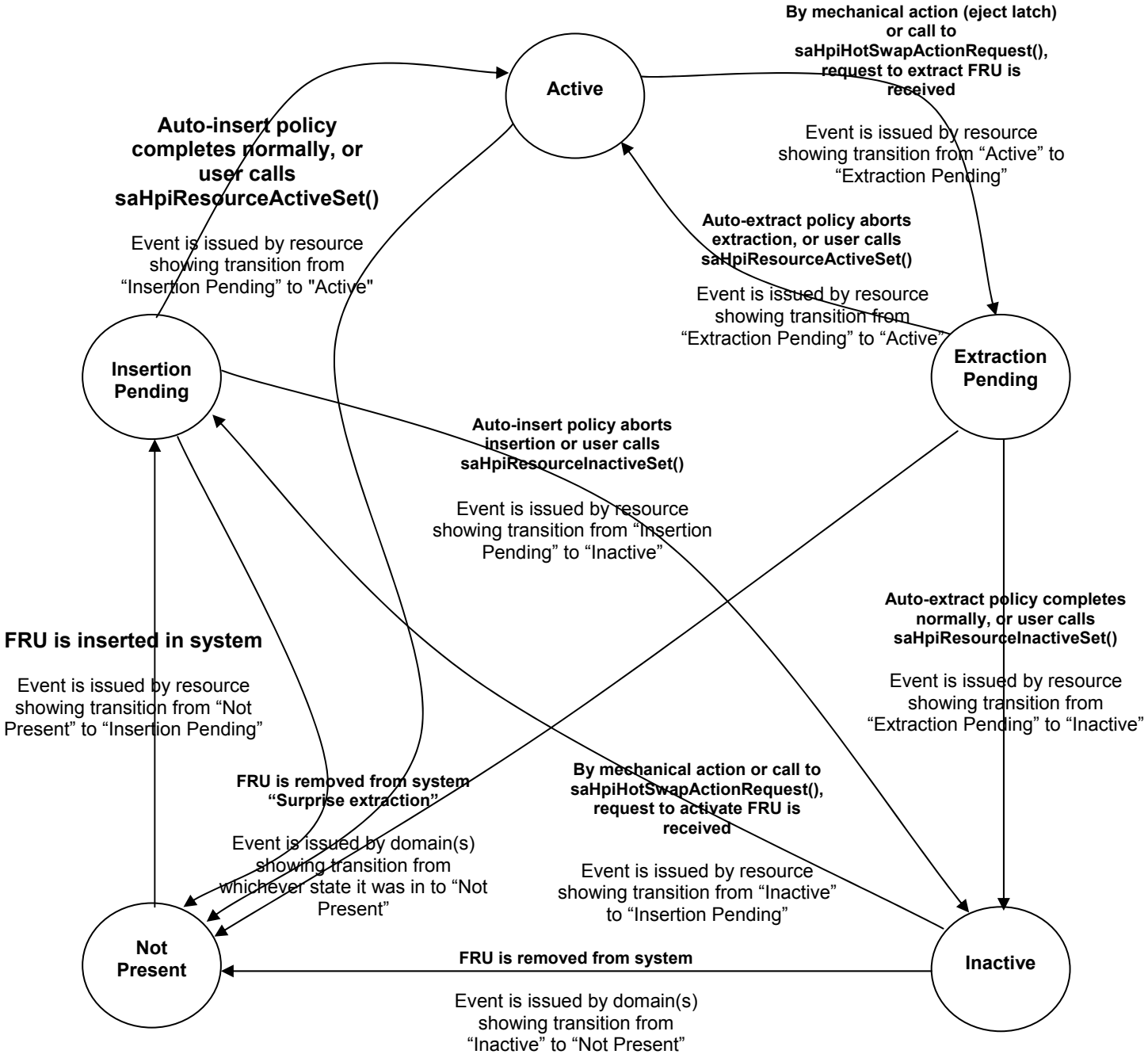
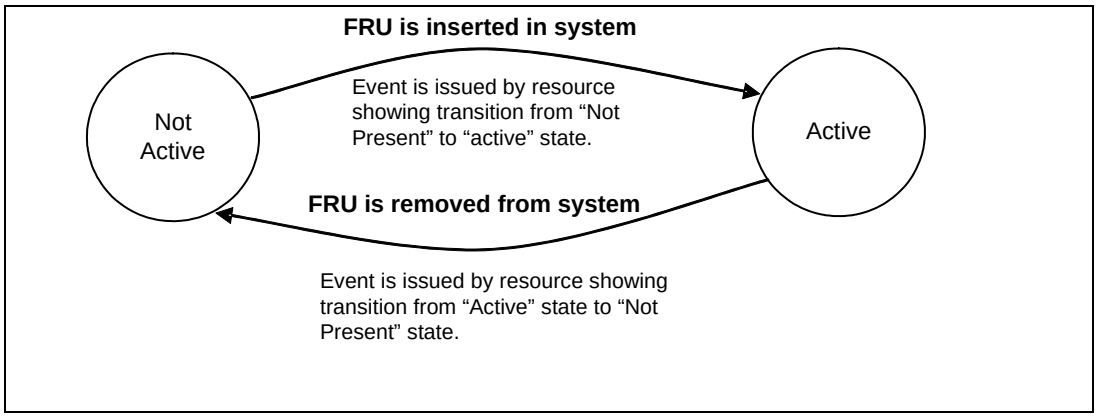


Figure 11. Simplified Hot Swap Model



The state of a resource as it enters a domain is unknown. Implementations may generate events, but an HPI User should not rely on events to determine the health upon insertion, as HPI implementations are not required to generate events to assert existing sensor states at the time of insertion.

HPI implementations may generate hot swap events that re-announce the current hot swap state of a resource. These events will have matching values for the previous and current hot swap states.

7.7.1 Hot Swap States

The HPI Hot Swap model defines five states of a resource during hot swap that result in a state transition, and the event that indicates the state transition. Hot swap events indicating normal state transitions are issued with a severity of `SAHPI_INFORMATIONAL`. Events which indicate a "surprise extraction" are issued with the severity set in the *ResourceSeverity* field of the RPT entry for the resource. A "surprise extraction" is a transition to the NOT PRESENT state from any state other than INACTIVE. HPI implementations may generate hot swap events that re-announce the current hot swap state of a resource. These events should have matching values for the previous and current hot swap states, and should be issued with a severity of `SAHPI_INFORMATIONAL`.

The five hot swap states listed below describe the life cycle of a resource supporting the Managed Hot Swap capability.

A resource which supports the "Simplified Hot Swap Model" (FRU capability set, but Managed Hot Swap capability not set) supports only the NOT PRESENT and ACTIVE states. These resources transition directly between the NOT PRESENT and the ACTIVE state and issue a Hot Swap event when such a transition occurs.

NOT PRESENT

The NOT PRESENT state is actually a virtual state that represents a resource that is not currently present in the domain, because the FRU associated with that resource is not currently present in the system. A resource is in this state before the FRU is physically inserted into the system or if it has been removed from the system. A resource that supports the full hot swap model typically transitions to this state from the INACTIVE state, but a resource can transition to this state from any state due to a surprise extraction of the FRU. The HPI implementation should generate a hot swap event with HotSwapState = `SAHPI_HS_STATE_NOT_PRESENT` when the resource transitions from any state to the NOT PRESENT state.

The event can indicate a normal transition from INACTIVE to the NOT PRESENT state or a surprise extraction from any other state. Normal transition events are issued with a severity of `SAHPI_INFORMATIONAL`. A resource following the simplified hot swap model always indicates the transition to the NOT PRESENT state as a surprise extraction.

When a resource associated with a FRU fails, many systems will detect this as a NOT PRESENT state for the FRU, and report it as a transition to the NOT PRESENT state (a normal transition if the resource was in the INACTIVE state, or a surprise extraction if it was in a different state). If the resource is subsequently restored to functionality, these systems will then detect the presence of the FRU, and report another state transition from the NOT PRESENT state to the current hot swap state for the restored resource. If the current hot swap state is not the normal "initial" state for the FRU when it is inserted (i.e., INSERTION PENDING for a resource that uses the full hot swap model or ACTIVE for a resource that uses the simplified hot swap model), then the severity of the "unusual" hot swap event showing a transition from NOT PRESENT to a different state should be the severity contained in the *ResourceSeverity* field in the resource's RPT entry.

INSERTION PENDING

The INSERTION PENDING state is entered after a resource that supports the full hot swap model has been added to the domain, as a result of the associated FRU being physically inserted into the system. This state indicates the resource is transitioning from a NOT PRESENT state or INACTIVE state into the ACTIVE state. When transitioning into the INSERTION PENDING state, the resource should generate a hot swap event with HotSwapState = `SAHPI_HS_STATE_INSERTION_PENDING`. The event can be generated when the ejector latch is shut (CompactPCI) or the device is seated in a slot.

Upon receiving the event, an HPI User has the opportunity to discover the capabilities of the resource before allowing the FRU associated with the resource to power on and become an active component in the system. During this state, the FRU can be commanded to power on or de-assert reset.

ACTIVE

The ACTIVE state indicates that a resource is now an active member of the domain.

After a FRU completes the hardware connection process, the associated resource enters the ACTIVE state. This does not mean that the FRU is now active at the software level, but merely indicates that the FRU is now active in the system and that it should not be abruptly removed. The HPI implementation generates a hot swap event with HotSwapState = `SAHPI_HS_STATE_ACTIVE` when the resource transitions to the ACTIVE state.

EXTRACTION PENDING

The EXTRACTION PENDING state indicates that the resource, which supports the full hot swap model, has requested extraction of the associated FRU. Typically, a resource enters an EXTRACTION PENDING state when an ejector latch is opened (PICMG 2.1) or when a hot swap button is pressed. The HPI implementation should generate a hot swap event with HotSwapState = `SAHPI_HS_STATE_EXTRACTION_PENDING` when a resource that supports Managed Hot Swap requests extraction.

Upon receiving the event, an HPI User has the opportunity to unload software drivers, relocate processes, or unmount file systems (software disconnect) before allowing a FRU to power down and disconnect from the system.

INACTIVE

The INACTIVE state indicates that the FRU, which supports the full hot swap model, is no longer active in the system, and that it has completed the extraction process. When a FRU completes the hardware disconnection process, it is logically and electrically disconnected or isolated from the platform but still physically located in the platform, so the associated resource remains in the domain. Typically, a FRU will be powered off or held in reset when in this state. The HPI implementation should generate a hot swap event with HotSwapState = `SAHPI_HS_STATE_INACTIVE` when the resource is transitioning to an INACTIVE state.

7.7.2 Hot Swap Auto Insertion and Auto Extraction Capabilities

A resource supporting hot swap typically supports default policies for insertion and extraction. On insertion, the default policy *may* be for the resource to turn the associated FRU's local power on and to de-assert reset. On extraction, the default policy *may* be for the resource to power off the FRU and turn on a hot swap indicator. These policies will automatically start after a configurable timeout period once the resource transitions to the INSERTION PENDING or EXTRACTION PENDING state. At the end of executing one of these policies, the resource will automatically change the hotswap state to `SAHPI_HS_STATE_ACTIVE` or `SAHPI_HS_STATE_INACTIVE`.

During the timeout period before the auto-insert or auto-extract policy begins, an HPI User can call `saHpiHotSwapPolicyCancel()` to request that the automatic insertion or extraction policy not be run. If the auto insertion or auto extraction policy is not run, then an HPI User must take all required actions to bring up or shut down the FRU, and must call `saHpiResourceActiveSet()` or `saHpiResourceInactiveSet()` to cause the transition out of the INSERTION PENDING or EXTRACTION PENDING state.

Because a resource that supports the simplified hot swap model will never transition into INSERTION PENDING or EXTRACTION PENDING states, the `saHpiHotSwapPolicyCancel()` function is not applicable to those resources.

7.7.3 Using Hot Swap

After receiving a hot swap event for a resource that supports the full hot swap model, an HPI User can cancel the default policy (using `saHpiHotSwapPolicyCancel()`, as described above) of the hot swap process for that resource. After canceling the auto-insertion policy, an HPI User can discover the capabilities of a resource before the FRU is allowed to power on and become an active component in the system. An HPI User can also cancel the auto-extraction policy after receiving the extraction event (again by using `saHpiHotSwapPolicyCancel()`). This allows an HPI User to complete the software disconnection process before a FRU is allowed to power down or be removed from the system.

An HPI User can initiate resource hot swap actions using the `saHpiHotSwapActionRequest()` function, assuming the FRU is physically installed in the system. This function allows an HPI User to simulate the actions of installing a FRU, or requesting extraction of a FRU.

After receiving a hot swap event for a resource that supports the simplified hot swap model, an HPI User should update its internal data, adjusting its processing to the insertion or removal of the FRU.

7.7.4 Hot Swap Functions

HPI defines a set of functions for managing the hot swap connection/disconnection process of resources that follow the full hot swap model. These functions are supported by resources that have the Managed Hot Swap capability set in their RPT entries.

Resources that support managed hot swap will often also support the Reset and Power functions described in Section 7.9 on page 154 and Section 7.10 on page 157. However, support for these functions is indicated by separate resource *Capability* flags, and is not implied by the Managed Hot Swap capability, nor is the Managed Hot Swap capability required for a resource to support those functions.

7.7.5 saHpiHotSwapPolicyCancel()

A resource supporting hot swap typically supports default policies for insertion and extraction. On insertion, the default policy may be for the resource to turn the associated FRU's local power on and to de-assert reset. On extraction, the default policy may be for the resource to immediately power off the FRU and turn on a hot swap indicator. This function allows an HPI User, after receiving a hot swap event with *HotSwapState* equal to `SAHPI_HS_STATE_INSERTION_PENDING` or `SAHPI_HS_STATE_EXTRACTION_PENDING`, to prevent the default policy from being executed.

Prototype

```
SaErrorT SAHPI_API saHpiHotSwapPolicyCancel (
    SAHPI_IN SaHpiSessionIdT      SessionId,
    SAHPI_IN SaHpiResourceIdT     ResourceId
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

ResourceId – [in] Resource identified for this operation.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_CAPABILITY` is returned if the resource does not support managed hot swap, as indicated by `SAHPI_CAPABILITY_MANAGED_HOTSWAP` in the resource's RPT entry.

`SA_ERR_HPI_INVALID_REQUEST` is returned if the resource is:

- Not in the INSERTION PENDING or EXTRACTION PENDING state.
- Processing an auto-insertion or auto-extraction policy.

Remarks

Each time the resource transitions to either the INSERTION PENDING or EXTRACTION PENDING state, the default policies will execute after the configured timeout period, unless cancelled using `saHpiHotSwapPolicyCancel()` after the transition to INSERTION PENDING or EXTRACTION PENDING state and before the timeout expires.The default policy cannot be canceled once it has begun to execute.

Because a resource that supports the simplified hot swap model will never transition into INSERTION PENDING or EXTRACTION PENDING states, this function is not applicable to those resources.

7.7.6 saHpiResourceActiveSet()

This function allows an HPI User to request a resource to transition to the ACTIVE state from the INSERTION PENDING or EXTRACTION PENDING state.

Prototype

```
SaErrorT SAHPI_API saHpiResourceActiveSet (
    SAHPI_IN SaHpiSessionIdT     SessionId,
    SAHPI_IN SaHpiResourceIdT    ResourceId
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

ResourceId – [in] Resource identified for this operation.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_CAPABILITY` is returned if the resource does not support managed hot swap, as indicated by `SAHPI_CAPABILITY_MANAGED_HOTSWAP` in the resource's RPT entry.

`SA_ERR_HPI_INVALID_REQUEST` is returned if the resource is:

- Not in the INSERTION PENDING or EXTRACTION PENDING state.
- Processing an auto-insertion or auto-extraction policy.

Remarks

During insertion, a resource supporting hot swap will generate an event to indicate that it is in the INSERTION PENDING state. If an HPI User calls `saHpiHotSwapPolicyCancel()` before the resource begins an auto-insert operation, then the resource will remain in INSERTION PENDING state while an HPI User acts on the resource to integrate it into the system. During this state, an HPI User can instruct the resource to power on the associated FRU, to de-assert reset, or to turn off its hot swap indicator using the `saHpiResourcePowerStateSet()`, `saHpiResourceResetStateSet()`, or `saHpiHotSwapIndicatorStateSet()` functions, respectively, if the resource has those associated capabilities. Once an HPI User has completed with the integration of the FRU, this function must be called to signal that the resource should now transition into the ACTIVE state.

An HPI User may also use this function to request a resource to return to the ACTIVE state from the EXTRACTION PENDING state in order to reject an extraction request.

Because a resource that supports the simplified hot swap model will never transition into INSERTION PENDING or EXTRACTION PENDING states, this function is not applicable to those resources.

Only valid if resource is in INSERTION PENDING or EXTRACTION PENDING state and an auto-insert or auto-extract policy action has not been initiated.

7.7.7 saHpiResourceInactiveSet()

This function allows an HPI User to request a resource to transition to the INACTIVE state from the INSERTION PENDING or EXTRACTION PENDING state.

Prototype

```
SaErrorT SAHPI_API saHpiResourceInactiveSet (
    SAHPI_IN SaHpiSessionIdT     SessionId,
    SAHPI_IN SaHpiResourceIdT    ResourceId
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

ResourceId – [in] Resource identified for this operation.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_CAPABILITY` is returned if the resource does not support managed hot swap, as indicated by `SAHPI_CAPABILITY_MANAGED_HOTSWAP` in the resource's RPT entry.

`SA_ERR_HPI_INVALID_REQUEST` is returned if the resource is:

- Not in the INSERTION PENDING or EXTRACTION PENDING state.
- Processing an auto-insertion or auto-extraction policy.

Remarks

During extraction, a resource supporting hot swap will generate an event to indicate that it is in the EXTRACTION PENDING state. If an HPI User calls `saHpiHotSwapPolicyCancel()` before the resource begins an auto-extract operation, then the resource will remain in EXTRACTION PENDING state while an HPI User acts on the resource to isolate the associated FRU from the system. During this state, an HPI User can instruct the resource to power off the FRU, to assert reset, or to turn on its hot swap indicator using the `saHpiResourcePowerStateSet()`, `saHpiResourceResetStateSet()`, or `saHpiHotSwapIndicatorStateSet()` functions, respectively, if the resource has these associated capabilities. Once an HPI User has completed the shutdown of the FRU, this function must be called to signal that the resource should now transition into the INACTIVE state.

An HPI User may also use this function to request a resource to return to the INACTIVE state from the INSERTION PENDING state to abort a hot-swap insertion action.

Because a resource that supports the simplified hot swap model will never transition into INSERTION PENDING or EXTRACTION PENDING states, this function is not applicable to those resources.

Only valid if resource is in EXTRACTION PENDING or INSERTION PENDING state and an auto-extract or auto-insert policy action has not been initiated.

7.7.8 saHpiAutoInsertTimeoutGet()

This function allows an HPI User to request the auto-insert timeout value within a specific domain. This value indicates how long the resource will wait before the default auto-insertion policy is invoked.

Prototype

```
SaErrorT SAHPI_API saHpiAutoInsertTimeoutGet(
    SAHPI_IN  SaHpiSessionIdT     SessionId,
    SAHPI_OUT SaHpiTimeoutT       *Timeout
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

Timeout – [out] Pointer to location to store the number of nanoseconds to wait before autonomous handling of the hot swap event. Reserved time out values:

- `SAHPI_TIMEOUT_IMMEDIATE` indicates autonomous handling is immediate.
- `SAHPI_TIMEOUT_BLOCK` indicates autonomous handling does not occur.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_INVALID_PARAMS` is returned if the *Timeout* pointer is passed in as NULL.

Remarks

Each domain maintains a single auto-insert timeout value and it is applied to all contained resources upon insertion, which support managed hot swap. Further information on the auto-insert timeout can be found in the function `saHpiAutoInsertTimeoutSet()`.

7.7.9 saHpiAutoInsertTimeoutSet()

This function allows an HPI User to configure a timeout for how long to wait before the default auto-insertion policy is invoked on a resource within a specific domain.

Prototype

```
SaErrorT SAHPI_API saHpiAutoInsertTimeoutSet(
    SAHPI_IN SaHpiSessionIdT        SessionId,
    SAHPI_IN SaHpiTimeoutT          Timeout
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

Timeout – [in] The number of nanoseconds to wait before autonomous handling of the hot swap event. Reserved time out values:

- `SAHPI_TIMEOUT_IMMEDIATE` indicates proceed immediately to autonomous handling.
- `SAHPI_TIMEOUT_BLOCK` indicates prevent autonomous handling.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_READ_ONLY` is returned if the auto-insert timeout for the domain is fixed as indicated by the `SAHPI_DOMAIN_CAP_AUTOINSERT_READ_ONLY` flag in the *DomainInfo* structure.

`SA_ERR_HPI_INVALID_PARAMS` is returned when the *Timeout* parameter is not set to `SAHPI_TIMEOUT_BLOCK`, `SAHPI_TIMEOUT_IMMEDIATE` or a positive value.

Remarks

This function accepts a parameter instructing each resource to impose a delay before performing its default hot swap policy for auto-insertion. The parameter may be set to `SAHPI_TIMEOUT_IMMEDIATE` to direct resources to proceed immediately to auto–insertion, or to `SAHPI_TIMEOUT_BLOCK` to prevent auto-insertion from ever occurring. If the parameter is set to another value, then it defines the number of nanoseconds between the time a hot swap event with HotSwapState = `SAHPI_HS_STATE_INSERTION_PENDING` is generated, and the time that the auto-insertion policy will be invoked for that resource. If, during this time period, a `saHpiHotSwapPolicyCancel()` function call is processed, the timer will be stopped, and the auto-insertion policy will not be invoked. Each domain maintains a single auto-insert timeout value and it is applied to all contained resources upon insertion, which support managed hot swap.

Once the auto-insertion policy begins, an HPI User will not be allowed to cancel the insertion policy; hence, the timeout should be set appropriately to allow for this condition. Note that the timeout period begins when the hot swap event with HotSwapState = `SAHPI_HS_STATE_INSERTION_PENDING` is initially generated; not when it is received by an HPI User with a `saHpiEventGet()` function call, or even when it is placed in a session event queue.

A resource may exist in multiple domains, which themselves may have different auto-insertion timeout values. Upon insertion, the resource will begin its auto-insertion policy based on the smallest auto-insertion timeout value. As an example, if a resource is inserted into two domains, one with an auto-insertion timeout of 5 seconds, and one with an auto-insertion timeout of 10 seconds, the resource will begin its auto-insertion policy at 5 seconds.

An implementation may enforce a fixed, preset timeout value. In such cases, the SAHPI_DOMAIN_CAP_AUTOINSERT_READ_ONLY flag will be set to indicate that an HPI User cannot change the auto-insert *Timeout* value. SA_ERR_HPI_READ_ONLY is returned if an HPI User attempts to change a read-only timeout.

7.7.10 saHpiAutoExtractTimeoutGet()

This function allows an HPI User to request the timeout for how long a resource will wait before the default auto-extraction policy is invoked.

Prototype

```
SaErrorT SAHPI_API saHpiAutoExtractTimeoutGet(
    SAHPI_IN  SaHpiSessionIdT      SessionId,
    SAHPI_IN  SaHpiResourceIdT     ResourceId,
    SAHPI_OUT SaHpiTimeoutT        *Timeout
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

ResourceId – [in] Resource identified for this operation.

Timeout – [out] Pointer to location to store the number of nanoseconds to wait before autonomous handling of the hot swap event. Reserved time out values:

- `SAHPI_TIMEOUT_IMMEDIATE` indicates autonomous handling is immediate.
- `SAHPI_TIMEOUT_BLOCK` indicates autonomous handling does not occur.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_CAPABILITY` is returned if the resource does not support managed hot swap, as indicated by `SAHPI_CAPABILITY_MANAGED_HOTSWAP` in the resource's RPT entry.

`SA_ERR_HPI_INVALID_PARAMS` is returned if the *Timeout* pointer is passed in as NULL.

Remarks

Further information on auto-extract timeouts is detailed in `saHpiAutoExtractTimeoutSet()`.

7.7.11 saHpiAutoExtractTimeoutSet()

This function allows an HPI User to configure a timeout for how long to wait before the default auto-extraction policy is invoked.

Prototype

```
SaErrorT SAHPI_API saHpiAutoExtractTimeoutSet(
    SAHPI_IN  SaHpiSessionIdT      SessionId,
    SAHPI_IN  SaHpiResourceIdT     ResourceId,
    SAHPI_IN  SaHpiTimeoutT        Timeout
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

ResourceId – [in] Resource identified for this operation.

Timeout – [in] The number of nanoseconds to wait before autonomous handling of the hot swap event. Reserved timeout values:

- `SAHPI_TIMEOUT_IMMEDIATE` indicates proceed immediately to autonomous handling.
- `SAHPI_TIMEOUT_BLOCK` indicates prevent autonomous handling.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_CAPABILITY` is returned if the resource does not support managed hot swap, as indicated by `SAHPI_CAPABILITY_MANAGED_HOTSWAP` in the resource's RPT entry.

`SA_ERR_HPI_INVALID_PARAMS` is returned when the *Timeout* parameter is not set to `SAHPI_TIMEOUT_BLOCK`, `SAHPI_TIMEOUT_IMMEDIATE` or a positive value.

`SA_ERR_HPI_READ_ONLY` is returned if the auto-extract timeout for the resource is fixed, as indicated by the `SAHPI_HS_CAPABILITY_AUTOEXTRACT_READ_ONLY` in the resource's RPT entry.

Remarks

This function accepts a parameter instructing the resource to impose a delay before performing its default hot swap policy for auto-extraction. The parameter may be set to `SAHPI_TIMEOUT_IMMEDIATE` to direct the resource to proceed immediately to auto-extraction, or to `SAHPI_TIMEOUT_BLOCK` to prevent auto-extraction from ever occurring on a resource. If the parameter is set to another value, then it defines the number of nanoseconds between the time a hot swap event with HotSwapState = `SAHPI_HS_STATE_EXTRACTION_PENDING` is generated and the time that the auto-extraction policy will be invoked for the resource. If, during this time period, a `saHpiHotSwapPolicyCancel()` function call is processed, the timer will be stopped, and the auto-extraction policy will not be invoked.

Once the auto-extraction policy begins, an HPI User will not be allowed to cancel the extraction policy; hence, the timeout should be set appropriately to allow for this condition. Note that the timeout period begins when the hot swap event with HotSwapState = `SAHPI_HS_STATE_EXTRACTION_PENDING` is initially generated; not when it is received by a HPI User with a `saHpiEventGet()` function call, or even when it is placed in a session event queue.

The auto-extraction policy is set at the resource level and is only supported by resources supporting the Managed Hot Swap capability. The auto-extraction timeout value cannot be modified if the resource's "Hot Swap AutoExtract Read-only" capability is set. After discovering that a newly inserted resource supports Managed Hot Swap, and read-write auto-extract timeouts, an HPI User may use this function to change the timeout of the auto-extraction policy for that resource. If a resource supports the simplified hot swap model, setting this timer has no effect since the resource will transition directly to NOT PRESENT state on an extraction.

An implementation may enforce a fixed, preset timeout value. In such cases, the `SAHPI_HS_CAPABILITY_AUTOEXTRACT_READ_ONLY` flag will be set to indicate that an HPI User cannot change the auto-extract *Timeout* value. `SA_ERR_HPI_READ_ONLY` is returned if an HPI User attempts to change a read-only timeout.

7.7.12 saHpiHotSwapStateGet()

This function allows an HPI User to retrieve the current hot swap state of a resource.

Prototype

```
SaErrorT SAHPI_API saHpiHotSwapStateGet (
    SAHPI_IN  SaHpiSessionIdT       SessionId,
    SAHPI_IN  SaHpiResourceIdT      ResourceId,
    SAHPI_OUT SaHpiHsStateT         *State
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

ResourceId – [in] Resource identified for this operation.

State – [out] Pointer to location to store returned state information.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_CAPABILITY` is returned if the resource does not support managed hot swap, as indicated by `SAHPI_CAPABILITY_MANAGED_HOTSWAP` in the resource's RPT entry.

`SA_ERR_HPI_INVALID_PARAMS` is returned if the *State* pointer is passed in as NULL.

Remarks

The returned state will be one of the following four states:

- `SAHPI_HS_STATE_INSERTION_PENDING`
- `SAHPI_HS_STATE_ACTIVE`
- `SAHPI_HS_STATE_EXTRACTION_PENDING`
- `SAHPI_HS_STATE_INACTIVE`

The state `SAHPI_HS_STATE_NOT_PRESENT` will never be returned, because a resource that is not present cannot be addressed by this function in the first place.

7.7.13 saHpiHotSwapActionRequest()

This function allows an HPI User to invoke an insertion or extraction process via software.

Prototype

```
SaErrorT SAHPI_API saHpiHotSwapActionRequest (
    SAHPI_IN SaHpiSessionIdT     SessionId,
    SAHPI_IN SaHpiResourceIdT    ResourceId,
    SAHPI_IN SaHpiHsActionT      Action
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

ResourceId – [in] Resource identified for this operation.

Action – [in] Requested action:

- `SAHPI_HS_ACTION_INSERTION`
- `SAHPI_HS_ACTION_EXTRACTION`

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_CAPABILITY` is returned if the resource does not support managed hot swap, as indicated by `SAHPI_CAPABILITY_MANAGED_HOTSWAP` in the resource's RPT entry.

`SA_ERR_HPI_INVALID_REQUEST` is returned if the resource is not in an appropriate hot swap state, or if the requested action is inappropriate for the current hot swap state. See the Remarks section below.

`SA_ERR_HPI_INVALID_PARAMS` is returned when *Action* is not one of the valid enumerated values for this type.

Remarks

A resource supporting hot swap typically requires a physical action on the associated FRU to invoke an insertion or extraction process. An insertion process is invoked by physically inserting the FRU into a chassis. Physically opening an ejector latch or pressing a button invokes the extraction process. This function provides an alternative means to invoke an insertion or extraction process via software.

This function may only be called:

- To request an insertion action when the resource is in INACTIVE state.
- To request an extraction action when the resource is in the ACTIVE state.

The function may not be called when the resource is in any other state.

7.7.14 saHpiHotSwapIndicatorStateGet()

This function allows an HPI User to retrieve the state of the hot swap indicator. A FRU associated with a hot-swappable resource may include a hot swap indicator such as a blue LED. This indicator signifies that the FRU is ready for removal.

Prototype

```
SaErrorT SAHPI_API saHpiHotSwapIndicatorStateGet (
    SAHPI_IN  SaHpiSessionIdT            SessionId,
    SAHPI_IN  SaHpiResourceIdT           ResourceId,
    SAHPI_OUT SaHpiHsIndicatorStateT     *State
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

ResourceId – [in] Resource identified for this operation.

State – [out] Pointer to location to store state of hot swap indicator.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_CAPABILITY` is returned if the resource does not support:

- Managed hot swap, as indicated by `SAHPI_CAPABILITY_MANAGED_HOTSWAP` in the resource's RPT entry.
- A hot swap indicator on the FRU as indicated by the `SAHPI_HS_CAPABILITY_INDICATOR_SUPPORTED` in the resource’s RPT entry.

`SA_ERR_HPI_INVALID_PARAMS` is returned if the *State* pointer is passed in as NULL.

Remarks

The returned state is either `SAHPI_HS_INDICATOR_OFF` or `SAHPI_HS_INDICATOR_ON`. This function will return the state of the indicator, regardless of what hot swap state the resource is in.

Not all resources supporting managed hot swap will necessarily support this function. Whether or not a resource supports the hot swap indicator is specified in the Hot Swap Capabilities field of the RPT entry.

7.7.15 saHpiHotSwapIndicatorStateSet()

This function allows an HPI User to set the state of the hot swap indicator. A FRU associated with a hot-swappable resource may include a hot swap indicator such as a blue LED. This indicator signifies that the FRU is ready for removal.

Prototype

```
SaErrorT SAHPI_API saHpiHotSwapIndicatorStateSet (
    SAHPI_IN SaHpiSessionIdT           SessionId,
    SAHPI_IN SaHpiResourceIdT          ResourceId,
    SAHPI_IN SaHpiHsIndicatorStateT    State
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

ResourceId – [in] Resource identified for this operation.

State – [in] State of hot swap indicator to be set.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_CAPABILITY` is returned if the resource does not support:

- Managed hot swap, as indicated by `SAHPI_CAPABILITY_MANAGED_HOTSWAP` in the resource's RPT entry.
- A hot swap indicator on the FRU as indicated by the `SAHPI_HS_CAPABILITY_INDICATOR_SUPPORTED` in the resource's RPT entry.

`SA_ERR_HPI_INVALID_PARAMS` is returned when *State* is not one of the valid enumerated values for this type.

Remarks

Valid states include `SAHPI_HS_INDICATOR_OFF` or `SAHPI_HS_INDICATOR_ON`. This function will set the indicator regardless of what hot swap state the resource is in, though it is recommended that this function be used only in conjunction with moving the resource to the appropriate hot swap state.

Not all resources supporting managed hot swap will necessarily support this function. Whether or not a resource supports the hot swap indicator is specified in the Hot Swap Capabilities field of the RPT entry.

7.8 Configuration

This function is valid for any and all resources that have the Configuration capability (`SAHPI_CAPABILITY_CONFIGURATION`) set in their corresponding RPT entries.

A resource maintains three sets of configuration items for the resource:

a) The current operational configuration settings

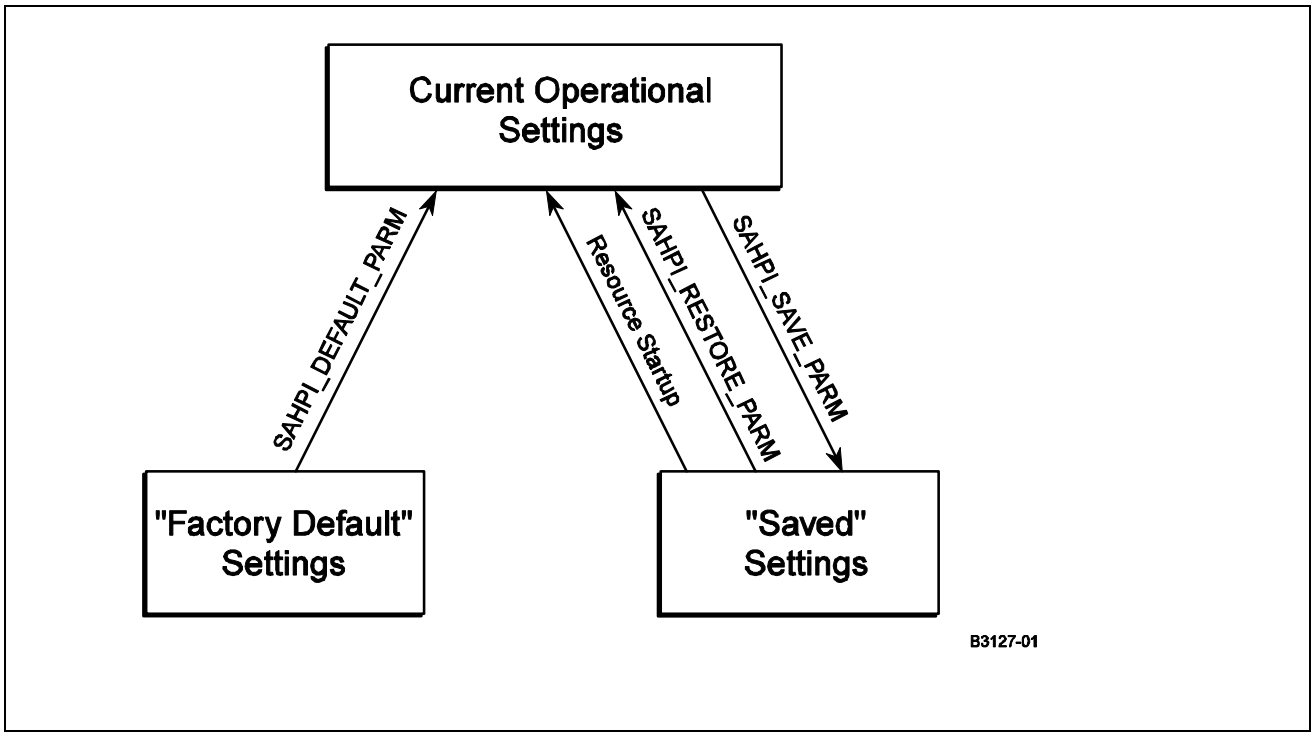
b) A set of "saved" configuration settings stored in non-volatile memory

c) The "factory default" configuration settings

Both the saved and factory default configuration settings are stored in non-volatile memory. Initially, when the resource is manufactured, the saved and factory default settings are the same. When the resource starts, e.g., at system startup or resource hot-swap insertion, the saved settings are used as the initial operational settings for the resource. During operation, the operational configuration settings may be changed by various actions of the platform, and by HPI Users making HPI API calls. The `saHpiParmControl()` API can be used to update the saved settings in non-volatile memory to the current values of the operational settings, so that these settings will be used the next time the resource is started. The `saHpiParmControl()` API can also be used to reset the current operational settings to either the "saved" settings or the "factory default" settings while the resource is operational. Figure 12 shows how these three sets of configuration settings are used when the resource starts, and when `saHpiParmControl()` is called with the various values for the *Action* parameter.

Note that when a resource is physically removed from a system and then re-inserted, there can be no assurances as to the state of the saved configuration settings.

Figure 12. Configuration Settings



The resource configuration items that are saved and restored with the `saHpiParmControl()` function are in three categories; mandatory, optional, and implementation-specific.

Every HPI resource that supports the Configuration capability must save and restore the mandatory configuration items via the `saHpiParmControl()` API. The mandatory configuration items are:

a) Auto-extract timeout value

b) Sensor enable states

c) Sensor event enable states

d) Sensor assert and deassert masks

e) Sensor threshold values and hysteresis values

Optional configuration settings may be saved and restored by a resource via the `saHpiParmControl()` API. Documentation for an HPI implementation should specify if these items are saved and restored, and if not saved, whether their settings are in volatile or non-volatile memory. The optional configuration items are:

a) Control state and mode settings

b) Watchdog configuration settings

c) Inventory Data Repository data. If Inventory Data Repository data is not saved and restored, but is immediately persisted by the resource, this will be indicated by a flag in the IDR resource data record.

HPI implementations may also save and restore implementation specific items that are not exported via the HPI interface when the `saHpiParmControl()` API is called. Documentation for an HPI implementation should describe these items, if any exist.

7.8.1 saHpiParmControl()

This function allows an HPI User to save and restore parameters associated with a specific resource.

Prototype

```
SaErrorT SAHPI_API saHpiParmControl (
    SAHPI_IN SaHpiSessionIdT      SessionId,
    SAHPI_IN SaHpiResourceIdT     ResourceId,
    SAHPI_IN SaHpiParmActionT     Action
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

ResourceId – [in] Resource identified for this operation.

Action – [in] Action to perform on resource parameters.

- `SAHPI_DEFAULT_PARM` Restores the factory default settings for a specific resource. Factory defaults include sensor thresholds and configurations, and resource- specific configuration parameters.
- `SAHPI_SAVE_PARM` Stores the resource configuration parameters in non-volatile storage. Resource configuration parameters stored in non-volatile storage will survive power cycles and resource resets.
- `SAHPI_RESTORE_PARM` Restores resource configuration parameters from non-volatile storage. Resource configuration parameters include sensor thresholds and sensor configurations, as well as resource-specific parameters.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_CAPABILITY` is returned if the resource does not support parameter control, as indicated by `SAHPI_CAPABILITY_CONFIGURATION` in the resource's RPT entry.

`SA_ERR_HPI_INVALID_PARAMS` is returned when *Action* is not set to a proper value.

Remarks

Resource-specific parameters should be documented in an implementation guide for the HPI implementation.

When this API is called with `SAHPI_RESTORE_PARM` as the action prior to having made a call with this API where the action parameter was set to `SAHPI_SAVE_PARM`, the default parameters will be restored.

7.9 Reset Management

These functions are valid for resources that have the Reset capability (`SAHPI_CAPABILITY_RESET`) set in their corresponding RPT entries.

Entities may be reset for a variety of reasons. A failed entity may be reset to bring it to a known state. In these cases, either a warm reset or a cold reset may be performed. A warm reset preserves entity state, whereas a cold reset does not. Both of these reset types are pulsed asserted and then de-asserted by the HPI implementation. This allows the HPI implementation to hold the reset asserted for the appropriate length of time, as needed by each entity.

`saHpiResourceResetStateSet()` can also be used for insertion and extraction scenarios. A typical resource supporting hot swap will have to ability to control local reset on the entity that it manages. During insertion, a resource can be instructed to assert reset, while the entity powers on. During extraction a resource can be requested to assert reset before the entity is powered off. `SAHPI_RESET_ASSERT` is used to hold the entity in reset. `SAHPI_RESET_DEASSERT` is used to bring an entity out of the reset state.

7.9.1 saHpiResourceResetStateGet()

This function gets the reset state of an entity, allowing an HPI User to determine if the entity is being held with its reset asserted. If a resource manages multiple entities, this function will address the entity which is identified in the RPT entry for the resource.

Prototype

```
SaErrorT SAHPI_API saHpiResourceResetStateGet (
    SAHPI_IN SaHpiSessionIdT        SessionId,
    SAHPI_IN SaHpiResourceIdT       ResourceId,
    SAHPI_OUT SaHpiResetActionT     *ResetAction
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

ResourceId – [in] Resource identified for this operation.

ResetAction – [out] The current reset state of the entity. Valid reset states are:

- `SAHPI_RESET_ASSERT`: The entity's reset is asserted, e.g., for hot swap insertion/extraction purposes.
- `SAHPI_RESET_DEASSERT`: The entity's reset is not asserted.

Return Value

`SA_OK` is returned if the resource has reset control, and the reset state has successfully been determined; otherwise, an error code is returned.

`SA_ERR_HPI_CAPABILITY` is returned if the resource does not support reset control as indicated by `SAHPI_CAPABILITY_RESET` in the resource's RPT entry.

`SA_ERR_HPI_INVALID_PARAMS` is returned if the *ResetAction* pointer is passed in as NULL.

Remarks

`SAHPI_COLD_RESET` and `SAHPI_WARM_RESET` are pulsed resets, and are not valid values to be returned in *ResetAction*. If the entity is not being held in reset (using `SAHPI_RESET_ASSERT`), the appropriate value is `SAHPI_RESET_DEASSERT`.

7.9.2 saHpiResourceResetStateSet()

This function directs the resource to perform the specified reset type on the entity that it manages. If a resource manages multiple entities, this function addresses the entity that is identified in the RPT entry for the resource.

Prototype

```
SaErrorT SAHPI_API saHpiResourceResetStateSet (
    SAHPI_IN SaHpiSessionIdT       SessionId,
    SAHPI_IN SaHpiResourceIdT      ResourceId,
    SAHPI_IN SaHpiResetActionT     ResetAction
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

ResourceId – [in] Resource identified for this operation.

ResetAction – [in] Type of reset to perform on the entity. Valid reset actions are:

- `SAHPI_COLD_RESET`: Perform a 'Cold Reset' on the entity (pulse), leaving reset de-asserted,
- `SAHPI_WARM_RESET`: Perform a 'Warm Reset' on the entity (pulse), leaving reset de-asserted,
- `SAHPI_RESET_ASSERT`: Put the entity into reset state and hold reset asserted, e.g., for hot swap insertion/extraction purposes,
- `SAHPI_RESET_DEASSERT`: Bring the entity out of the reset state.

Return Value

`SA_OK` is returned if the resource has reset control, and the requested reset action has succeeded; otherwise, an error code is returned.

`SA_ERR_HPI_CAPABILITY` is returned if the resource does not support resource reset control, as indicated by `SAHPI_CAPABILITY_RESET` in the resource's RPT entry.

`SA_ERR_HPI_INVALID_PARAMS` is returned when the *ResetAction* is not set to a proper value.

`SA_ERR_HPI_INVALID_CMD` is returned if the requested *ResetAction* is `SAHPI_RESET_ASSERT` and the resource does not support this action.

`SA_ERR_HPI_INVALID_REQUEST` is returned if the *ResetAction* is `SAHPI_COLD_RESET` or `SAHPI_WARM_RESET` and reset is currently asserted.

Remarks

Some resources may not support holding the entity in reset. If this is the case, the resource should return `SA_ERR_HPI_INVALID_CMD` if the `SAHPI_RESET_ASSERT` action is requested. All resources that have the `SAHPI_CAPABILITY_RESET` flag set in their RPTs should support `SAHPI_COLD_RESET` and `SAHPI_WARM_RESET`. However, it is not required that these actions be different. That is, some resources may only have one sort of reset action (e.g., a "cold" reset) which is executed when either `SAHPI_COLD_RESET` or `SAHPI_WARM_RESET` is requested.

The `SAHPI_RESET_ASSERT` is used to hold an entity in reset, and `SAHPI_RESET_DEASSERT` is used to bring the entity out of an asserted reset state.

7.10 Power Management

These functions are valid for resources that have the Power capability (`SAHPI_CAPABILITY_POWER`) set in their corresponding RPT entries.

Entities may be powered on, powered off or power-cycled. It is legal to set any power state regardless of the current state. If the power is already in the state being set, the function will have no effect and will return normally (assuming there are no other errors). A power cycle operation is equivalent to a power off followed, after an implementation-dependent delay, by a power on. If the entity is already powered off, a power cycle operation will result in turning the power on.

`saHpiResourcePowerStateSet()` can also be used for insertion and extraction scenarios. A typical resource supporting hot swap will also support power control for the FRU entity. During insertion, a resource can be instructed to power on the FRU at an appropriate time. During extraction a resource can be requested to power off the FRU.

7.10.1 saHpiResourcePowerStateGet()

This function gets the power state of an entity, allowing an HPI User to determine if the entity is currently powered on or off. If a resource manages multiple entities, this function will address the entity which is identified in the RPT entry for the resource.

Prototype

```
SaErrorT SAHPI_API saHpiResourcePowerStateGet (
    SAHPI_IN  SaHpiSessionIdT       SessionId,
    SAHPI_IN  SaHpiResourceIdT      ResourceId,
    SAHPI_OUT SaHpiPowerStateT      *State
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

ResourceId – [in] Resource identified for this operation.

State – [out] The current power state of the resource. Valid power states are:

- `SAHPI_POWER_OFF`: The entity's primary power is OFF,
- `SAHPI_POWER_ON`: The entity's primary power is ON.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_CAPABILITY` is returned if the resource does not support power management, as indicated by `SAHPI_CAPABILITY_POWER` in the resource's RPT entry.

`SA_ERR_HPI_INVALID_PARAMS` is returned if the *State* pointer is passed in as NULL.

Remarks

`SAHPI_POWER_CYCLE` is a pulsed power operation and is not a valid return for the power state.

7.10.2 saHpiResourcePowerStateSet()

This function directs the resource to perform the power control action on the entity that it manages. If a resource manages multiple entities, this function addresses the entity that is identified in the RPT entry for the resource.

Prototype

```
SaErrorT SAHPI_API saHpiResourcePowerStateSet (
    SAHPI_IN SaHpiSessionIdT       SessionId,
    SAHPI_IN SaHpiResourceIdT      ResourceId,
    SAHPI_IN SaHpiPowerStateT      State
);
```

Parameters

SessionId – [in] Identifier for a session context previously obtained using `saHpiSessionOpen()`.

ResourceId – [in] Resource identified for this operation.

State – [in] the requested power control action. Valid values are:

- `SAHPI_POWER_OFF`: The entity's primary power is turned OFF,
- `SAHPI_POWER_ON`: The entity's primary power is turned ON,
- `SAHPI_POWER_CYCLE`: The entity's primary power is turned OFF, then turned ON.

Return Value

`SA_OK` is returned on successful completion; otherwise, an error code is returned.

`SA_ERR_HPI_CAPABILITY` is returned if the resource does not support power management, as indicated by `SAHPI_CAPABILITY_POWER` in the resource's RPT entry.

`SA_ERR_HPI_INVALID_PARAMS` is returned when *State* is not one of the valid enumerated values for this type.

Remarks

This function controls the hardware power on a FRU entity regardless of what hot-swap state the resource is in. For example, it is legal (and may be desirable) to cycle power on the FRU even while it is in ACTIVE state in order to attempt to clear a fault condition. Similarly, a resource could be instructed to power on a FRU even while it is in INACTIVE state, for example, in order to run off-line diagnostics.

Not all resources supporting hot swap will necessarily support this function. In particular, resources that use the simplified hot swap model may not have the ability to control FRU power.

This function may also be supported for non-FRU entities if power control is available for those entities.

8 Data Type Definitions

8.1 Basic Data Types and Values

```
/*****************************************************************************
*****************************************************************************
**********                                                         **********
**********                 Basic Data Types and Values             **********
**********                                                         **********
*****************************************************************************
*****************************************************************************/

/* General Types – need to be specified correctly for the host architecture */

/*
**   It is recommended that these types be defined such that the data sizes
**   and alignment of each data type are as indicated.  The only requirement
**   for source compatibility is that the types be defined to be able to
**   contain at least the required data (e.g., at least signed 8-bit values
**   must be contained in the data type defined as SaHpiInt8T, etc.)
**   Following the full recommendations for data size and alignment, however,
**   may promote more binary compatibility.
*/

typedef <. . .> SaHpiUint8T;   /* unsigned 8-bit data, 1-byte alignment   */
typedef <. . .> SaHpiUint16T;  /* unsigned 16-bit data, 2-byte alignment  */
typedef <. . .> SaHpiUint32T;  /* unsigned 32-bit data, 4-byte alignment  */
typedef <. . .> SaHpiUint64T;  /* unsigned 64-bit data, 8-byte alignment  */
typedef <. . .> SaHpiInt8T;    /* signed 8-bit data, 1-byte alignment     */
typedef <. . .> SaHpiInt16T;   /* signed 16-bit data, 2-byte alignment    */
typedef <. . .> SaHpiInt32T;   /* signed 32-bit data, 4-byte alignment    */
typedef <. . .> SaHpiInt64T;   /* signed 64-bit data, 8-byte alignment    */
typedef <. . .> SaHpiFloat64T; /* 64-bit floating point, 8-byte alignment */

typedef SaHpiUint8T     SaHpiBoolT;
#define SAHPI_TRUE      1       /* While SAHPI_TRUE = 1, any non-zero
                                   value is also considered to be True
                                   and HPI Users/Implementers of this
                                   specification should not test for
                                   equality against SAHPI_TRUE. */

#define SAHPI_FALSE     0

/* Platform, O/S, or Vendor dependent */
#define SAHPI_API
#define SAHPI_IN
#define SAHPI_OUT
#define SAHPI_INOUT

/*
** Identifier for the manufacturer
**
** This is the IANA-assigned private enterprise number for the
** manufacturer of the resource or FRU, or of the manufacturer
** defining an OEM control or event type. A list of current
** IANA-assigned private enterprise numbers may be obtained at
**
**     http://www.iana.org/assignments/enterprise-numbers
**
** If a manufacturer does not currently have an assigned number, one
** may be obtained by following the instructions located at
**
**     http://www.iana.org/cgi-bin/enterprise.pl
*/
typedef SaHpiUint32T SaHpiManufacturerIdT;
#define SAHPI_MANUFACTURER_ID_UNSPECIFIED (SaHpiManufacturerIdT)0

/* Version Types  */
```

```
typedef SaHpiUint32T SaHpiVersionT;

/*
** Interface Version
**
** The interface version is the version of the actual interface and not the
** version of the implementation. It is a 24 bit value where
** the most significant 8 bits represent the compatibility level
** (with letters represented as the corresponding numbers);
** the next 8 bits represent the major version number; and
** the least significant 8 bits represent the minor version number.
*/
#define SAHPI_INTERFACE_VERSION (SaHpiVersionT)0x020101  /* B.01.01 */

/*
** Return Codes
**
** SaErrorT is defined in the HPI specification.  In the future a
** common SAF types definition may be created to contain this type. At
** that time, this typedef should be removed. Each of the return codes
** is defined in Section 4.1 of the specification.
*/
typedef SaHpiInt32T SaErrorT; /* Return code */

/*
** SA_OK:
*/
#define SA_OK                          (SaErrorT)0x0000

/* This value is the base for all HPI-specific error codes. */
#define SA_HPI_ERR_BASE                -1000

#define SA_ERR_HPI_ERROR               (SaErrorT)(SA_HPI_ERR_BASE - 1)
#define SA_ERR_HPI_UNSUPPORTED_API     (SaErrorT)(SA_HPI_ERR_BASE - 2)
#define SA_ERR_HPI_BUSY                (SaErrorT)(SA_HPI_ERR_BASE - 3)
#define SA_ERR_HPI_INTERNAL_ERROR      (SaErrorT)(SA_HPI_ERR_BASE - 4)
#define SA_ERR_HPI_INVALID_CMD         (SaErrorT)(SA_HPI_ERR_BASE - 5)
#define SA_ERR_HPI_TIMEOUT             (SaErrorT)(SA_HPI_ERR_BASE - 6)
#define SA_ERR_HPI_OUT_OF_SPACE        (SaErrorT)(SA_HPI_ERR_BASE - 7)
#define SA_ERR_HPI_OUT_OF_MEMORY       (SaErrorT)(SA_HPI_ERR_BASE - 8)
#define SA_ERR_HPI_INVALID_PARAMS      (SaErrorT)(SA_HPI_ERR_BASE - 9)
#define SA_ERR_HPI_INVALID_DATA        (SaErrorT)(SA_HPI_ERR_BASE - 10)
#define SA_ERR_HPI_NOT_PRESENT         (SaErrorT)(SA_HPI_ERR_BASE - 11)
#define SA_ERR_HPI_NO_RESPONSE         (SaErrorT)(SA_HPI_ERR_BASE - 12)
#define SA_ERR_HPI_DUPLICATE           (SaErrorT)(SA_HPI_ERR_BASE - 13)
#define SA_ERR_HPI_INVALID_SESSION     (SaErrorT)(SA_HPI_ERR_BASE - 14)
#define SA_ERR_HPI_INVALID_DOMAIN      (SaErrorT)(SA_HPI_ERR_BASE - 15)
#define SA_ERR_HPI_INVALID_RESOURCE    (SaErrorT)(SA_HPI_ERR_BASE - 16)
#define SA_ERR_HPI_INVALID_REQUEST     (SaErrorT)(SA_HPI_ERR_BASE - 17)
#define SA_ERR_HPI_ENTITY_NOT_PRESENT  (SaErrorT)(SA_HPI_ERR_BASE - 18)
#define SA_ERR_HPI_READ_ONLY           (SaErrorT)(SA_HPI_ERR_BASE - 19)
#define SA_ERR_HPI_CAPABILITY          (SaErrorT)(SA_HPI_ERR_BASE - 20)
#define SA_ERR_HPI_UNKNOWN             (SaErrorT)(SA_HPI_ERR_BASE - 21)

/*
** Domain, Session and Resource Type Definitions
*/

/* Domain ID. */
typedef SaHpiUint32T SaHpiDomainIdT;

/* The SAHPI_UNSPECIFIED_DOMAIN_ID value is used to specify the default
** domain.
*/
#define SAHPI_UNSPECIFIED_DOMAIN_ID (SaHpiDomainIdT) 0xFFFFFFFF

/* Session ID. */
typedef SaHpiUint32T SaHpiSessionIdT;

/* Resource identifier. */
typedef SaHpiUint32T SaHpiResourceIdT;
```

```
/* The SAHPI_UNSPECIFIED_RESOURCE_ID value is used to specify the Domain
** Event Log and to specify that there is no resource for such things as HPI
** User events/alarms.
*/
#define SAHPI_UNSPECIFIED_RESOURCE_ID (SaHpiResourceIdT) 0xFFFFFFFF

/* Table Related Type Definitions  */
typedef SaHpiUint32T SaHpiEntryIdT;
#define SAHPI_FIRST_ENTRY (SaHpiEntryIdT)0x00000000
#define SAHPI_LAST_ENTRY  (SaHpiEntryIdT)0xFFFFFFFF
#define SAHPI_ENTRY_UNSPECIFIED SAHPI_FIRST_ENTRY

/*
** Time Related Type Definitions
**
** An HPI time value represents the local time as the number of nanoseconds
** from 00:00:00, January 1, 1970, in a 64-bit signed integer. This format
** is sufficient to represent times with nano-second resolution from the
** year 1678 to 2262. Every API which deals with time values must define
** the timezone used.
**
** It should be noted that although nano-second resolution is supported
** in the data type, the actual resolution provided by an implementation
** may be more limited than this.
**
** The value -2**63, which is 0x8000000000000000, is used to indicate
** “unknown/unspecified time”.
**
** Conversion to/from POSIX and other common time representations is
** relatively straightforward. The following code framgment converts
** between SaHpiTimeT and time_t:
**
**     time_t tt1, tt2;
**     SaHpiTimeT saHpiTime;
**
**     time(&tt1);
**     saHpiTime = (SaHpiTimeT) tt1 * 1000000000;
**     tt2 = saHpiTime / 1000000000;
**
** The following fragment converts between SaHpiTimeT and a struct timeval:
**
**     struct timeval tv1, tv2;
**     SaHpiTimeT saHpiTime;
**
**     gettimeofday(&tv1, NULL);
**     saHpiTime = (SaHpiTimeT) tv1.tv_sec * 1000000000 + tv1.tv_usec * 1000;
**     tv2.tv_sec = saHpiTime / 1000000000;
**     tv2.tv_usec = saHpiTime % 1000000000 / 1000;
**
** The following fragment converts between SaHpiTimeT and a struct timespec:
**
**     struct timespec ts1, ts2;
**     SaHpiTimeT saHpiTime;
**
**     clock_gettime(CLOCK_REALTIME, &ts1);
**     saHpiTime = (SaHpiTimeT) ts1.tv_sec * 1000000000 + ts1.tv_nsec;
**     ts2.tv_sec = saHpiTime / 1000000000;
**     ts2.tv_nsec = saHpiTime % 1000000000;
**
** Note, however, that since time_t is (effectively) universally 32 bits,
** all of these conversions will cease to work on January 18, 2038.
**
** Some subsystems may need the flexibility to report either absolute or
** relative (eg. to system boot) times. This will typically be in the
** case of a board which may or may not, depending on the system setup,
** have an idea of absolute time. For example, some boards may have
** “time of day” clocks which start at zero, and never get set to the
** time of day.
**
** In these cases, times which represent “current” time (in events, for
```

```
** example) can be reported based on the clock value, whether it has been
** set to the actual date/time, or whether it represents the elapsed time
** since boot. If it is the time since boot, the value will be (for 27
** years) less than 0x0C00000000000000, which is Mon May 26 16:58:48 1997.
** If the value is greater than this, then it can be assumed to be an
** absolute time.
**
** There is no practical need within the interface for expressing dates prior
** to the publication of this specification (which is more than five years
** after the “break point” between relative and absolute time). Thus, in all
** instances a time value should be interpreted as “relative” times if the
** value is less than or equal to SAHPI_TIME_MAX_RELATIVE (but not equal to
** SAHPI_TIME_UNSPECIFIED which always means the time is not available), or
** “absolute” times if the value is greater than SAHPI_TIME_MAX_RELATIVE.
*/
typedef SaHpiInt64T SaHpiTimeT;    /* Time in nanoseconds */

/* Unspecified or unknown time */
#define SAHPI_TIME_UNSPECIFIED     (SaHpiTimeT) 0x8000000000000000LL

/* Maximum time that can be specified as relative */
#define SAHPI_TIME_MAX_RELATIVE    (SaHpiTimeT) 0x0C00000000000000LL
typedef SaHpiInt64T SaHpiTimeoutT; /* Timeout in nanoseconds */

/* Non-blocking call */
#define SAHPI_TIMEOUT_IMMEDIATE    (SaHpiTimeoutT) 0x0000000000000000LL

/* Blocking call, wait indefinitely for call to complete */
#define SAHPI_TIMEOUT_BLOCK        (SaHpiTimeoutT) -1LL

/*
** Language
**
** This enumeration lists all of the languages that can be associated with text.
**
** SAHPI_LANG_UNDEF indicates that the language is unspecified or
** unknown.
*/
typedef enum {
    SAHPI_LANG_UNDEF = 0, SAHPI_LANG_AFAR, SAHPI_LANG_ABKHAZIAN,
    SAHPI_LANG_AFRIKAANS, SAHPI_LANG_AMHARIC, SAHPI_LANG_ARABIC,
    SAHPI_LANG_ASSAMESE, SAHPI_LANG_AYMARA, SAHPI_LANG_AZERBAIJANI,
    SAHPI_LANG_BASHKIR, SAHPI_LANG_BYELORUSSIAN, SAHPI_LANG_BULGARIAN,
    SAHPI_LANG_BIHARI, SAHPI_LANG_BISLAMA, SAHPI_LANG_BENGALI,
    SAHPI_LANG_TIBETAN, SAHPI_LANG_BRETON, SAHPI_LANG_CATALAN,
    SAHPI_LANG_CORSICAN, SAHPI_LANG_CZECH, SAHPI_LANG_WELSH,
    SAHPI_LANG_DANISH, SAHPI_LANG_GERMAN, SAHPI_LANG_BHUTANI,
    SAHPI_LANG_GREEK, SAHPI_LANG_ENGLISH, SAHPI_LANG_ESPERANTO,
    SAHPI_LANG_SPANISH, SAHPI_LANG_ESTONIAN, SAHPI_LANG_BASQUE,
    SAHPI_LANG_PERSIAN, SAHPI_LANG_FINNISH, SAHPI_LANG_FIJI,
    SAHPI_LANG_FAEROESE, SAHPI_LANG_FRENCH, SAHPI_LANG_FRISIAN,
    SAHPI_LANG_IRISH, SAHPI_LANG_SCOTSGAELIC, SAHPI_LANG_GALICIAN,
    SAHPI_LANG_GUARANI, SAHPI_LANG_GUJARATI, SAHPI_LANG_HAUSA,
    SAHPI_LANG_HINDI, SAHPI_LANG_CROATIAN, SAHPI_LANG_HUNGARIAN,
    SAHPI_LANG_ARMENIAN, SAHPI_LANG_INTERLINGUA, SAHPI_LANG_INTERLINGUE,
    SAHPI_LANG_INUPIAK, SAHPI_LANG_INDONESIAN, SAHPI_LANG_ICELANDIC,
    SAHPI_LANG_ITALIAN, SAHPI_LANG_HEBREW, SAHPI_LANG_JAPANESE,
    SAHPI_LANG_YIDDISH, SAHPI_LANG_JAVANESE, SAHPI_LANG_GEORGIAN,
    SAHPI_LANG_KAZAKH, SAHPI_LANG_GREENLANDIC, SAHPI_LANG_CAMBODIAN,
    SAHPI_LANG_KANNADA, SAHPI_LANG_KOREAN, SAHPI_LANG_KASHMIRI,
    SAHPI_LANG_KURDISH, SAHPI_LANG_KIRGHIZ, SAHPI_LANG_LATIN,
    SAHPI_LANG_LINGALA, SAHPI_LANG_LAOTHIAN, SAHPI_LANG_LITHUANIAN,
    SAHPI_LANG_LATVIANLETTISH, SAHPI_LANG_MALAGASY, SAHPI_LANG_MAORI,
    SAHPI_LANG_MACEDONIAN, SAHPI_LANG_MALAYALAM, SAHPI_LANG_MONGOLIAN,
    SAHPI_LANG_MOLDAVIAN, SAHPI_LANG_MARATHI, SAHPI_LANG_MALAY,
    SAHPI_LANG_MALTESE, SAHPI_LANG_BURMESE, SAHPI_LANG_NAURU,
    SAHPI_LANG_NEPALI, SAHPI_LANG_DUTCH, SAHPI_LANG_NORWEGIAN,
    SAHPI_LANG_OCCITAN, SAHPI_LANG_AFANOROMO, SAHPI_LANG_ORIYA,
    SAHPI_LANG_PUNJABI, SAHPI_LANG_POLISH, SAHPI_LANG_PASHTOPUSHTO,
    SAHPI_LANG_PORTUGUESE, SAHPI_LANG_QUECHUA, SAHPI_LANG_RHAETOROMANCE,
    SAHPI_LANG_KIRUNDI, SAHPI_LANG_ROMANIAN, SAHPI_LANG_RUSSIAN,
```

```
    SAHPI_LANG_KINYARWANDA, SAHPI_LANG_SANSKRIT, SAHPI_LANG_SINDHI,
    SAHPI_LANG_SANGRO, SAHPI_LANG_SERBOCROATIAN, SAHPI_LANG_SINGHALESE,
    SAHPI_LANG_SLOVAK, SAHPI_LANG_SLOVENIAN, SAHPI_LANG_SAMOAN,
    SAHPI_LANG_SHONA, SAHPI_LANG_SOMALI, SAHPI_LANG_ALBANIAN,
    SAHPI_LANG_SERBIAN, SAHPI_LANG_SISWATI, SAHPI_LANG_SESOTHO,
    SAHPI_LANG_SUDANESE, SAHPI_LANG_SWEDISH, SAHPI_LANG_SWAHILI,
    SAHPI_LANG_TAMIL, SAHPI_LANG_TELUGU, SAHPI_LANG_TAJIK,
    SAHPI_LANG_THAI, SAHPI_LANG_TIGRINYA, SAHPI_LANG_TURKMEN,
    SAHPI_LANG_TAGALOG, SAHPI_LANG_SETSWANA, SAHPI_LANG_TONGA,
    SAHPI_LANG_TURKISH, SAHPI_LANG_TSONGA, SAHPI_LANG_TATAR,
    SAHPI_LANG_TWI, SAHPI_LANG_UKRAINIAN, SAHPI_LANG_URDU,
    SAHPI_LANG_UZBEK, SAHPI_LANG_VIETNAMESE, SAHPI_LANG_VOLAPUK,
    SAHPI_LANG_WOLOF, SAHPI_LANG_XHOSA, SAHPI_LANG_YORUBA,
    SAHPI_LANG_CHINESE, SAHPI_LANG_ZULU
} SaHpiLanguageT;

/*
** Text Buffers
**
** These structures are used for defining the type of data in the text buffer
** and the length of the buffer. Text buffers are used in the inventory data,
** RDR, RPT, etc. for variable length strings of data.
**
** The encoding of the Data field in the SaHpiTextBufferT structure is defined
** by the value of the DataType field in the buffer.  The following table
** describes the various encodings:
**
**    DataType                    Encoding
**    --------                    --------
**
**   SAHPI_TL_TYPE_UNICODE        16-bit Unicode, least significant byte first.
**                                Buffer must contain even number of bytes.
**
**   SAHPI_TL_TYPE_BCDPLUS        8-bit ASCII, '0'-'9' or space, dash, period,
**                                colon, comma, or underscore only.
**
**   SAHPI_TL_TYPE_ASCII6         8-bit ASCII, reduced set, 0x20=0x5f only.
**
**   SAHPI_TL_TYPE_TEXT           8-bit ASCII+Latin 1
**
**   SAHPI_TL_TYPE_BINARY         8-bit bytes, any values legal
**
** Note: "ASCII+Latin 1" is derived from the first 256 characters of
**       Unicode 2.0. The first 256 codes of Unicode follow ISO 646 (ASCII)
**       and ISO 8859/1 (Latin 1). The Unicode "C0 Controls and Basic Latin"
**       set defines the first 128 8-bit characters (00h-7Fh) and the
**       "C1 Controls and Latin 1 Supplement" defines the second 128 (80h-FFh).
**
** Note: The SAHPI_TL_TYPE_BCDPLUS and SAHPI_TL_TYPE_ASCII6 encodings
**       use normal ASCII character encodings, but restrict the allowed
**       characters to a subset of the entire ASCII character set. These
**       encodings are used when the target device contains restrictions
**       on which characters it can store or display.  SAHPI_TL_TYPE_BCDPLUS
**       data may be stored externally as 4-bit values, and
**       SAHPI_TL_TYPE_ASCII6 may be stored externally as 6-bit values.
**       But, regardless of how the data is stored externally, it is
**       encoded as 8-bit ASCII in the SaHpiTextBufferT structure passed
**       across the HPI.
*/

#define SAHPI_MAX_TEXT_BUFFER_LENGTH  255

typedef enum {
    SAHPI_TL_TYPE_UNICODE = 0,     /* 2-byte UNICODE characters; DataLength
                                      must be even. */
    SAHPI_TL_TYPE_BCDPLUS,        /* String of ASCII characters, '0'-'9', space,
                                      dash, period, colon, comma or underscore
                                      ONLY */
    SAHPI_TL_TYPE_ASCII6,         /* Reduced ASCII character set: 0x20-0x5F
                                      ONLY */
    SAHPI_TL_TYPE_TEXT,           /* ASCII+Latin 1 */
```

```
    SAHPI_TL_TYPE_BINARY          /* Binary data, any values legal */
} SaHpiTextTypeT;

typedef struct {
    SaHpiTextTypeT DataType;
    SaHpiLanguageT Language;      /* Language the text is in. */
    SaHpiUint8T    DataLength;    /* Bytes used in Data buffer  */
    SaHpiUint8T    Data[SAHPI_MAX_TEXT_BUFFER_LENGTH];  /* Data buffer */
} SaHpiTextBufferT;

/*
** Instrument Id
**
** The following data type is used for all management instrument identifiers -
** sensor numbers, control numbers, watchdog timer numbers, etc.
**
*/

typedef SaHpiUint32T SaHpiInstrumentIdT;
```

8.2 Entities

```
/*****************************************************************************
*****************************************************************************
**********                                                         **********
**********                          Entities                       **********
**********                                                         **********
*****************************************************************************
*****************************************************************************/
/*
** Entity Types
**
** Entities are used to associate specific hardware components with sensors,
** controls, watchdogs, or resources. Entities are defined with an entity
** type enumeration, and an entity location number (to identify
** the physical location of a particular type of entity).
**
** Entities are uniquely identified in a system with an ordered series of
** Entity Type / Entity Location pairs called an “Entity Path”. Each subsequent
** Entity Type/Entity Location in the path is the next higher “containing”
** entity. The “root” of the Entity Path (the outermost level of containment)
** is designated with an Entity Type of SAHPI_ENT_ROOT if the entire Entity Path
** is fewer than SAHPI_MAX_ENTITY_PATH entries in length.
**
** Enumerated Entity Types include those types enumerated by the IPMI Consortium
** for IPMI-managed entities, as well as additional types defined by the
** HPI specification. Room is left in the enumeration for the inclusion of
** Entity Types taken from other lists, if needed in the future.
*/
/* Base values for entity types from various sources. */
#define SAHPI_ENT_IPMI_GROUP 0
#define SAHPI_ENT_SAFHPI_GROUP 0x10000
#define SAHPI_ENT_ROOT_VALUE 0xFFFF
typedef enum
{
    SAHPI_ENT_UNSPECIFIED = SAHPI_ENT_IPMI_GROUP,
    SAHPI_ENT_OTHER,
    SAHPI_ENT_UNKNOWN,
    SAHPI_ENT_PROCESSOR,
    SAHPI_ENT_DISK_BAY,           /* Disk or disk bay  */
    SAHPI_ENT_PERIPHERAL_BAY,
    SAHPI_ENT_SYS_MGMNT_MODULE,   /* System management module  */
    SAHPI_ENT_SYSTEM_BOARD,       /* Main system board, may also be
                                     processor board and/or internal
                                     expansion board */
    SAHPI_ENT_MEMORY_MODULE,      /* Board holding memory devices */
    SAHPI_ENT_PROCESSOR_MODULE,   /* Holds processors, use this
                                     designation when processors are not
                                     mounted on system board */
    SAHPI_ENT_POWER_SUPPLY,       /* Main power supply (supplies) for the
```

```
                                  system */
    SAHPI_ENT_ADD_IN_CARD,
    SAHPI_ENT_FRONT_PANEL_BOARD,   /* Control panel  */
    SAHPI_ENT_BACK_PANEL_BOARD,
    SAHPI_ENT_POWER_SYSTEM_BOARD,
    SAHPI_ENT_DRIVE_BACKPLANE,
    SAHPI_ENT_SYS_EXPANSION_BOARD, /* System internal expansion board
                                  (contains expansion slots). */
    SAHPI_ENT_OTHER_SYSTEM_BOARD,  /* Part of board set          */
    SAHPI_ENT_PROCESSOR_BOARD,     /* Holds 1 or more processors. Includes
                                  boards that hold SECC modules) */
    SAHPI_ENT_POWER_UNIT,          /* Power unit / power domain (typically
                                  used as a pre-defined logical entity
                                  for grouping power supplies)*/
    SAHPI_ENT_POWER_MODULE,        /* Power module / DC-to-DC converter.
                                  Use this value for internal
                                  converters. Note: You should use
                                  entity ID (power supply) for the
                                  main power supply even if the main
                                  supply is a DC-to-DC converter */
    SAHPI_ENT_POWER_MGMNT,         /* Power management/distribution
                                  board */
    SAHPI_ENT_CHASSIS_BACK_PANEL_BOARD,
    SAHPI_ENT_SYSTEM_CHASSIS,
    SAHPI_ENT_SUB_CHASSIS,
    SAHPI_ENT_OTHER_CHASSIS_BOARD,
    SAHPI_ENT_DISK_DRIVE_BAY,
    SAHPI_ENT_PERIPHERAL_BAY_2,
    SAHPI_ENT_DEVICE_BAY,
    SAHPI_ENT_COOLING_DEVICE,      /* Fan/cooling device */
    SAHPI_ENT_COOLING_UNIT,        /* Can be used as a pre-defined logical
                                  entity for grouping fans or other
                                  cooling devices. */
    SAHPI_ENT_INTERCONNECT,        /* Cable / interconnect */
    SAHPI_ENT_MEMORY_DEVICE,       /* This Entity ID should be used for
                                  replaceable memory devices, e.g.
                                  DIMM/SIMM. It is recommended that
                                  Entity IDs not be used for
                                  individual non-replaceable memory
                                  devices. Rather, monitoring and
                                  error reporting should be associated
                                  with the FRU [e.g. memory card]
                                  holding the memory. */
    SAHPI_ENT_SYS_MGMNT_SOFTWARE, /* System Management Software  */
    SAHPI_ENT_BIOS,
    SAHPI_ENT_OPERATING_SYSTEM,
    SAHPI_ENT_SYSTEM_BUS,
    SAHPI_ENT_GROUP,              /* This is a logical entity for use with
                                  Entity Association records. It is
                                  provided to allow a sensor data
                                  record to point to an entity-
                                  association record when there is no
                                  appropriate pre-defined logical
                                  entity for the entity grouping.
                                  This Entity should not be used as a
                                  physical entity. */
    SAHPI_ENT_REMOTE,             /* Out of band management communication
                                  device */
    SAHPI_ENT_EXTERNAL_ENVIRONMENT,
    SAHPI_ENT_BATTERY,
    SAHPI_ENT_CHASSIS_SPECIFIC    = SAHPI_ENT_IPMI_GROUP + 0x90,
    SAHPI_ENT_BOARD_SET_SPECIFIC  = SAHPI_ENT_IPMI_GROUP + 0xB0,
    SAHPI_ENT_OEM_SYSINT_SPECIFIC = SAHPI_ENT_IPMI_GROUP + 0xD0,
    SAHPI_ENT_ROOT = SAHPI_ENT_ROOT_VALUE,
    SAHPI_ENT_RACK = SAHPI_ENT_SAFHPI_GROUP,
    SAHPI_ENT_SUBRACK,
    SAHPI_ENT_COMPACTPCI_CHASSIS,
    SAHPI_ENT_ADVANCEDTCA_CHASSIS,
    SAHPI_ENT_RACK_MOUNTED_SERVER,
    SAHPI_ENT_SYSTEM_BLADE,
    SAHPI_ENT_SWITCH,                  /* Network switch, such as a
```

```
                                        rack-mounted ethernet or fabric
                                        switch. */
    SAHPI_ENT_SWITCH_BLADE,             /* Network switch, as above, but in
                                        a bladed system. */
    SAHPI_ENT_SBC_BLADE,
    SAHPI_ENT_IO_BLADE,
    SAHPI_ENT_DISK_BLADE,
    SAHPI_ENT_DISK_DRIVE,
    SAHPI_ENT_FAN,
    SAHPI_ENT_POWER_DISTRIBUTION_UNIT,
    SAHPI_ENT_SPEC_PROC_BLADE,          /* Special Processing Blade,
                                        including DSP */
    SAHPI_ENT_IO_SUBBOARD,              /* I/O Sub-Board, including
                                        PMC I/O board */
    SAHPI_ENT_SBC_SUBBOARD,             /* SBC Sub-Board, including PMC
                                        SBC board */
    SAHPI_ENT_ALARM_MANAGER,            /* Chassis alarm manager board */
    SAHPI_ENT_SHELF_MANAGER,            /* Blade-based shelf manager */
    SAHPI_ENT_DISPLAY_PANEL,            /* Display panel, such as an
                                        alarm display panel. */
    SAHPI_ENT_SUBBOARD_CARRIER_BLADE,   /* Includes PMC Carrier Blade --
                                        Use only if "carrier" is only
                                        function of blade. Else use
                                        primary function (SBC_BLADE,
                                        DSP_BLADE, etc.). */
    SAHPI_ENT_PHYSICAL_SLOT             /* Indicates the physical slot into
                                        which a blade is inserted. */
} SaHpiEntityTypeT;

typedef SaHpiUint32T SaHpiEntityLocationT;

typedef struct {
    SaHpiEntityTypeT     EntityType;
    SaHpiEntityLocationT EntityLocation;
} SaHpiEntityT;

#define SAHPI_MAX_ENTITY_PATH 16

typedef struct {
    SaHpiEntityT  Entry[SAHPI_MAX_ENTITY_PATH];
} SaHpiEntityPathT;
```

8.3 Events, Part 1

```
/*****************************************************************************
*****************************************************************************
**********                                                         **********
**********                      Events, part 1                     **********
**********                                                         **********
*****************************************************************************
*****************************************************************************/
/*
** Category
**
** Sensor events contain an event category and event state. Depending on the
** event category, the event states take on different meanings for events
** generated by specific sensors.
**
** It is recommended that implementations map their sensor specific
** event categories into the set of categories listed here.  When such a mapping
** is impractical or impossible, the SAHPI_EC_SENSOR_SPECIFIC category should
** be used.
**
** The SAHPI_EC_GENERIC category can be used for discrete sensors which have
** state meanings other than those identified with other event categories.
*/
typedef SaHpiUint8T SaHpiEventCategoryT;

#define SAHPI_EC_UNSPECIFIED     (SaHpiEventCategoryT)0x00 /* Unspecified */
#define SAHPI_EC_THRESHOLD       (SaHpiEventCategoryT)0x01 /* Threshold
```

```
                                                        events */
#define SAHPI_EC_USAGE          (SaHpiEventCategoryT)0x02 /* Usage state
                                                        events */
#define SAHPI_EC_STATE          (SaHpiEventCategoryT)0x03 /* Generic state
                                                        events */
#define SAHPI_EC_PRED_FAIL      (SaHpiEventCategoryT)0x04 /* Predictive fail
                                                        events */
#define SAHPI_EC_LIMIT          (SaHpiEventCategoryT)0x05 /* Limit events */
#define SAHPI_EC_PERFORMANCE    (SaHpiEventCategoryT)0x06 /* Performance
                                                        events    */
#define SAHPI_EC_SEVERITY       (SaHpiEventCategoryT)0x07 /* Severity
                                                        indicating
                                                        events */
#define SAHPI_EC_PRESENCE       (SaHpiEventCategoryT)0x08 /* Device presence
                                                        events */
#define SAHPI_EC_ENABLE         (SaHpiEventCategoryT)0x09 /* Device enabled
                                                        events */
#define SAHPI_EC_AVAILABILITY   (SaHpiEventCategoryT)0x0A /* Availability
                                                        state events */
#define SAHPI_EC_REDUNDANCY     (SaHpiEventCategoryT)0x0B /* Redundancy
                                                        state events */
#define SAHPI_EC_SENSOR_SPECIFIC (SaHpiEventCategoryT)0x7E /* Sensor-
                                                        specific events */
#define SAHPI_EC_GENERIC        (SaHpiEventCategoryT)0x7F /* OEM defined
                                                        events */

/*
** Event States
**
** The following event states are specified relative to the categories listed
** above. The event types are only valid for their given category. Each set of
** events is labeled as to which category it belongs to.
** Each event will have only one event state associated with it. When retrieving
** the event status or event enabled status a bit mask of all applicable event
** states is used. Similarly, when setting the event enabled status a bit mask
** of all applicable event states is used.
*/
typedef SaHpiUint16T SaHpiEventStateT;

/*
** SaHpiEventCategoryT == <any>
*/
#define SAHPI_ES_UNSPECIFIED (SaHpiEventStateT)0x0000

/*
** SaHpiEventCategoryT == SAHPI_EC_THRESHOLD
** When using these event states, the event state should match
** the event severity (for example SAHPI_ES_LOWER_MINOR should have an
** event severity of SAHPI_MINOR).
*/
#define SAHPI_ES_LOWER_MINOR (SaHpiEventStateT)0x0001
#define SAHPI_ES_LOWER_MAJOR (SaHpiEventStateT)0x0002
#define SAHPI_ES_LOWER_CRIT  (SaHpiEventStateT)0x0004
#define SAHPI_ES_UPPER_MINOR (SaHpiEventStateT)0x0008
#define SAHPI_ES_UPPER_MAJOR (SaHpiEventStateT)0x0010
#define SAHPI_ES_UPPER_CRIT  (SaHpiEventStateT)0x0020

/* SaHpiEventCategoryT == SAHPI_EC_USAGE */
#define SAHPI_ES_IDLE   (SaHpiEventStateT)0x0001
#define SAHPI_ES_ACTIVE (SaHpiEventStateT)0x0002
#define SAHPI_ES_BUSY   (SaHpiEventStateT)0x0004

/* SaHpiEventCategoryT == SAHPI_EC_STATE */
#define SAHPI_ES_STATE_DEASSERTED (SaHpiEventStateT)0x0001
#define SAHPI_ES_STATE_ASSERTED   (SaHpiEventStateT)0x0002

/* SaHpiEventCategoryT == SAHPI_EC_PRED_FAIL */
#define SAHPI_ES_PRED_FAILURE_DEASSERT (SaHpiEventStateT)0x0001
#define SAHPI_ES_PRED_FAILURE_ASSERT   (SaHpiEventStateT)0x0002

/* SaHpiEventCategoryT == SAHPI_EC_LIMIT */
```

```
#define SAHPI_ES_LIMIT_NOT_EXCEEDED (SaHpiEventStateT)0x0001
#define SAHPI_ES_LIMIT_EXCEEDED     (SaHpiEventStateT)0x0002

/* SaHpiEventCategoryT == SAHPI_EC_PERFORMANCE */
#define SAHPI_ES_PERFORMANCE_MET   (SaHpiEventStateT)0x0001
#define SAHPI_ES_PERFORMANCE_LAGS  (SaHpiEventStateT)0x0002

/*
** SaHpiEventCategoryT == SAHPI_EC_SEVERITY
** When using these event states, the event state should match
** the event severity
*/
#define SAHPI_ES_OK                  (SaHpiEventStateT)0x0001
#define SAHPI_ES_MINOR_FROM_OK       (SaHpiEventStateT)0x0002
#define SAHPI_ES_MAJOR_FROM_LESS     (SaHpiEventStateT)0x0004
#define SAHPI_ES_CRITICAL_FROM_LESS  (SaHpiEventStateT)0x0008
#define SAHPI_ES_MINOR_FROM_MORE     (SaHpiEventStateT)0x0010
#define SAHPI_ES_MAJOR_FROM_CRITICAL (SaHpiEventStateT)0x0020
#define SAHPI_ES_CRITICAL            (SaHpiEventStateT)0x0040
#define SAHPI_ES_MONITOR             (SaHpiEventStateT)0x0080
#define SAHPI_ES_INFORMATIONAL       (SaHpiEventStateT)0x0100

/* SaHpiEventCategoryT == SAHPI_EC_PRESENCE */
#define SAHPI_ES_ABSENT  (SaHpiEventStateT)0x0001
#define SAHPI_ES_PRESENT (SaHpiEventStateT)0x0002

/* SaHpiEventCategoryT == SAHPI_EC_ENABLE */
#define SAHPI_ES_DISABLED (SaHpiEventStateT)0x0001
#define SAHPI_ES_ENABLED  (SaHpiEventStateT)0x0002

/* SaHpiEventCategoryT == SAHPI_EC_AVAILABILITY */
#define SAHPI_ES_RUNNING       (SaHpiEventStateT)0x0001
#define SAHPI_ES_TEST          (SaHpiEventStateT)0x0002
#define SAHPI_ES_POWER_OFF     (SaHpiEventStateT)0x0004
#define SAHPI_ES_ON_LINE       (SaHpiEventStateT)0x0008
#define SAHPI_ES_OFF_LINE      (SaHpiEventStateT)0x0010
#define SAHPI_ES_OFF_DUTY      (SaHpiEventStateT)0x0020
#define SAHPI_ES_DEGRADED      (SaHpiEventStateT)0x0040
#define SAHPI_ES_POWER_SAVE    (SaHpiEventStateT)0x0080
#define SAHPI_ES_INSTALL_ERROR (SaHpiEventStateT)0x0100

/* SaHpiEventCategoryT == SAHPI_EC_REDUNDANCY */
#define SAHPI_ES_FULLY_REDUNDANT                  (SaHpiEventStateT)0x0001
#define SAHPI_ES_REDUNDANCY_LOST                  (SaHpiEventStateT)0x0002
#define SAHPI_ES_REDUNDANCY_DEGRADED              (SaHpiEventStateT)0x0004
#define SAHPI_ES_REDUNDANCY_LOST_SUFFICIENT_RESOURCES \
                                                  (SaHpiEventStateT)0x0008
#define SAHPI_ES_NON_REDUNDANT_SUFFICIENT_RESOURCES \
                                                  (SaHpiEventStateT)0x0010
#define SAHPI_ES_NON_REDUNDANT_INSUFFICIENT_RESOURCES \
                                                  (SaHpiEventStateT)0x0020
#define SAHPI_ES_REDUNDANCY_DEGRADED_FROM_FULL    (SaHpiEventStateT)0x0040
#define SAHPI_ES_REDUNDANCY_DEGRADED_FROM_NON     (SaHpiEventStateT)0x0080

/*
** SaHpiEventCategoryT == SAHPI_EC_GENERIC || SAHPI_EC_SENSOR_SPECIFIC
** These event states are implementation-specific.
*/
#define SAHPI_ES_STATE_00  (SaHpiEventStateT)0x0001
#define SAHPI_ES_STATE_01  (SaHpiEventStateT)0x0002
#define SAHPI_ES_STATE_02  (SaHpiEventStateT)0x0004
#define SAHPI_ES_STATE_03  (SaHpiEventStateT)0x0008
#define SAHPI_ES_STATE_04  (SaHpiEventStateT)0x0010
#define SAHPI_ES_STATE_05  (SaHpiEventStateT)0x0020
#define SAHPI_ES_STATE_06  (SaHpiEventStateT)0x0040
#define SAHPI_ES_STATE_07  (SaHpiEventStateT)0x0080
#define SAHPI_ES_STATE_08  (SaHpiEventStateT)0x0100
#define SAHPI_ES_STATE_09  (SaHpiEventStateT)0x0200
#define SAHPI_ES_STATE_10  (SaHpiEventStateT)0x0400
#define SAHPI_ES_STATE_11  (SaHpiEventStateT)0x0800
#define SAHPI_ES_STATE_12  (SaHpiEventStateT)0x1000
```

```
#define SAHPI_ES_STATE_13  (SaHpiEventStateT)0x2000
#define SAHPI_ES_STATE_14  (SaHpiEventStateT)0x4000
```

8.4 Sensors

```
/*****************************************************************************
*****************************************************************************
**********                                                         **********
**********                        Sensors                          **********
**********                                                         **********
*****************************************************************************
*****************************************************************************/

/* Sensor Number */
typedef SaHpiInstrumentIdT SaHpiSensorNumT;
/* The following specifies the named range for standard sensor numbers. */
#define SAHPI_STANDARD_SENSOR_MIN   (SaHpiSensorNumT)0x00000100
#define SAHPI_STANDARD_SENSOR_MAX   (SaHpiSensorNumT)0x000001FF

/* Type of Sensor */
typedef enum {
    SAHPI_TEMPERATURE = 0x01,
    SAHPI_VOLTAGE,
    SAHPI_CURRENT,
    SAHPI_FAN,
    SAHPI_PHYSICAL_SECURITY,
    SAHPI_PLATFORM_VIOLATION,
    SAHPI_PROCESSOR,
    SAHPI_POWER_SUPPLY,
    SAHPI_POWER_UNIT,
    SAHPI_COOLING_DEVICE,
    SAHPI_OTHER_UNITS_BASED_SENSOR,
    SAHPI_MEMORY,
    SAHPI_DRIVE_SLOT,
    SAHPI_POST_MEMORY_RESIZE,
    SAHPI_SYSTEM_FW_PROGRESS,
    SAHPI_EVENT_LOGGING_DISABLED,
    SAHPI_RESERVED1,
    SAHPI_SYSTEM_EVENT,
    SAHPI_CRITICAL_INTERRUPT,
    SAHPI_BUTTON,
    SAHPI_MODULE_BOARD,
    SAHPI_MICROCONTROLLER_COPROCESSOR,
    SAHPI_ADDIN_CARD,
    SAHPI_CHASSIS,
    SAHPI_CHIP_SET,
    SAHPI_OTHER_FRU,
    SAHPI_CABLE_INTERCONNECT,
    SAHPI_TERMINATOR,
    SAHPI_SYSTEM_BOOT_INITIATED,
    SAHPI_BOOT_ERROR,
    SAHPI_OS_BOOT,
    SAHPI_OS_CRITICAL_STOP,
    SAHPI_SLOT_CONNECTOR,
    SAHPI_SYSTEM_ACPI_POWER_STATE,
    SAHPI_RESERVED2,
    SAHPI_PLATFORM_ALERT,
    SAHPI_ENTITY_PRESENCE,
    SAHPI_MONITOR_ASIC_IC,
    SAHPI_LAN,
    SAHPI_MANAGEMENT_SUBSYSTEM_HEALTH,
    SAHPI_BATTERY,
    SAHPI_OPERATIONAL = 0xA0,
    SAHPI_OEM_SENSOR=0xC0
}  SaHpiSensorTypeT;

/*
** Sensor Reading Type
**
** These definitions list the available data types that can be
** used for sensor readings.
```

```
**
*/

#define SAHPI_SENSOR_BUFFER_LENGTH 32

typedef enum {
      SAHPI_SENSOR_READING_TYPE_INT64,
      SAHPI_SENSOR_READING_TYPE_UINT64,
      SAHPI_SENSOR_READING_TYPE_FLOAT64,
      SAHPI_SENSOR_READING_TYPE_BUFFER    /* 32 byte array. The format of
                                             the buffer is implementation-
                                             specific. Sensors that use
                                             this reading type may not have
                                             thresholds that are settable
                                             or readable. */
} SaHpiSensorReadingTypeT;

typedef union {
    SaHpiInt64T          SensorInt64;
    SaHpiUint64T         SensorUint64;
    SaHpiFloat64T        SensorFloat64;
    SaHpiUint8T          SensorBuffer[SAHPI_SENSOR_BUFFER_LENGTH];
} SaHpiSensorReadingUnionT;

/*
** Sensor Reading
**
** The sensor reading data structure is returned from a call to get
** sensor reading. The structure is also used when setting and getting sensor
** threshold values and reporting sensor ranges.
**
** IsSupported is set when a sensor reading/threshold value is available.
** Otherwise, if no reading or threshold is supported, this flag is set to
** False.
**
*/

typedef struct {
      SaHpiBoolT                  IsSupported;
      SaHpiSensorReadingTypeT     Type;
      SaHpiSensorReadingUnionT     Value;
} SaHpiSensorReadingT;


/* Sensor Event Mask Actions – used with saHpiSensorEventMasksSet() */

typedef enum {
    SAHPI_SENS_ADD_EVENTS_TO_MASKS,
    SAHPI_SENS_REMOVE_EVENTS_FROM_MASKS
} SaHpiSensorEventMaskActionT;

/* Value to use for AssertEvents or DeassertEvents parameter
   in saHpiSensorEventMasksSet() to set or clear all supported
   event states for a sensor in the mask */

#define SAHPI_ALL_EVENT_STATES (SaHpiEventStateT)0xFFFF

/*
** Threshold Values
** This structure encompasses all of the thresholds that can be set.
** These are set and read with the same units as sensors report in
** saHpiSensorReadingGet().  When hysteresis is not constant over the
** range of sensor values, it is calculated at the nominal sensor reading,
** as given in the Range field of the sensor RDR.
**
** Thresholds are required to be set in-order (such that the setting for
** UpCritical is greater than or equal to the setting for UpMajor, etc.).*/

typedef struct {
    SaHpiSensorReadingT LowCritical;      /* Lower Critical Threshold */
    SaHpiSensorReadingT LowMajor;         /* Lower Major Threshold */
```

```
    SaHpiSensorReadingT LowMinor;         /* Lower Minor Threshold */
    SaHpiSensorReadingT UpCritical;       /* Upper critical Threshold */
    SaHpiSensorReadingT UpMajor;          /* Upper major Threshold */
    SaHpiSensorReadingT UpMinor;          /* Upper minor Threshold */
    SaHpiSensorReadingT PosThdHysteresis; /* Positive Threshold Hysteresis */
    SaHpiSensorReadingT NegThdHysteresis; /* Negative Threshold Hysteresis */
}SaHpiSensorThresholdsT;
```

8.5 Sensor Resource Data Records

```
/*****************************************************************************
*****************************************************************************
**********                                                         **********
**********                 Sensor Resource Data Records            **********
**********                                                         **********
*****************************************************************************
*****************************************************************************/

/*
**  Sensor Range
** Sensor range values can include minimum, maximum, normal minimum, normal
** maximum, and nominal values.
**
** Sensor thresholds cannot be set outside of the range defined by SAHPI_SRF_MIN
** through SAHPI_SRF_MAX, if these limits are present (as indicated by the
** SaHpiSensorRangeFlagsT).  If the MIN limit is not present, no lower bound
** will be enforced on sensor thresholds.  If the MAX limit is not present, no
** upper bound will be enforced on sensor thresholds.
*/
typedef SaHpiUint8T SaHpiSensorRangeFlagsT;
#define SAHPI_SRF_MIN        (SaHpiSensorRangeFlagsT)0x10
#define SAHPI_SRF_MAX        (SaHpiSensorRangeFlagsT)0x08
#define SAHPI_SRF_NORMAL_MIN (SaHpiSensorRangeFlagsT)0x04
#define SAHPI_SRF_NORMAL_MAX (SaHpiSensorRangeFlagsT)0x02
#define SAHPI_SRF_NOMINAL    (SaHpiSensorRangeFlagsT)0x01

typedef struct {
    SaHpiSensorRangeFlagsT  Flags;
    SaHpiSensorReadingT     Max;
    SaHpiSensorReadingT     Min;
    SaHpiSensorReadingT     Nominal;
    SaHpiSensorReadingT     NormalMax;
    SaHpiSensorReadingT     NormalMin;
} SaHpiSensorRangeT;

/*
** Sensor Units
** This is a list of all the sensor units supported by HPI.
*/
typedef enum {
    SAHPI_SU_UNSPECIFIED = 0, SAHPI_SU_DEGREES_C, SAHPI_SU_DEGREES_F,
    SAHPI_SU_DEGREES_K, SAHPI_SU_VOLTS, SAHPI_SU_AMPS,
    SAHPI_SU_WATTS, SAHPI_SU_JOULES, SAHPI_SU_COULOMBS,
    SAHPI_SU_VA, SAHPI_SU_NITS, SAHPI_SU_LUMEN,
    SAHPI_SU_LUX, SAHPI_SU_CANDELA, SAHPI_SU_KPA,
    SAHPI_SU_PSI, SAHPI_SU_NEWTON, SAHPI_SU_CFM,
    SAHPI_SU_RPM, SAHPI_SU_HZ, SAHPI_SU_MICROSECOND,
    SAHPI_SU_MILLISECOND, SAHPI_SU_SECOND, SAHPI_SU_MINUTE,
    SAHPI_SU_HOUR, SAHPI_SU_DAY, SAHPI_SU_WEEK,
    SAHPI_SU_MIL, SAHPI_SU_INCHES, SAHPI_SU_FEET,
    SAHPI_SU_CU_IN, SAHPI_SU_CU_FEET, SAHPI_SU_MM,
    SAHPI_SU_CM, SAHPI_SU_M, SAHPI_SU_CU_CM,
    SAHPI_SU_CU_M, SAHPI_SU_LITERS, SAHPI_SU_FLUID_OUNCE,
    SAHPI_SU_RADIANS, SAHPI_SU_STERADIANS, SAHPI_SU_REVOLUTIONS,
    SAHPI_SU_CYCLES, SAHPI_SU_GRAVITIES, SAHPI_SU_OUNCE,
    SAHPI_SU_POUND, SAHPI_SU_FT_LB, SAHPI_SU_OZ_IN,
    SAHPI_SU_GAUSS, SAHPI_SU_GILBERTS, SAHPI_SU_HENRY,
    SAHPI_SU_MILLIHENRY, SAHPI_SU_FARAD, SAHPI_SU_MICROFARAD,
    SAHPI_SU_OHMS, SAHPI_SU_SIEMENS, SAHPI_SU_MOLE,
    SAHPI_SU_BECQUEREL, SAHPI_SU_PPM, SAHPI_SU_RESERVED,
```

```
    SAHPI_SU_DECIBELS, SAHPI_SU_DBA, SAHPI_SU_DBC,
    SAHPI_SU_GRAY, SAHPI_SU_SIEVERT, SAHPI_SU_COLOR_TEMP_DEG_K,
    SAHPI_SU_BIT, SAHPI_SU_KILOBIT, SAHPI_SU_MEGABIT,
    SAHPI_SU_GIGABIT, SAHPI_SU_BYTE, SAHPI_SU_KILOBYTE,
    SAHPI_SU_MEGABYTE, SAHPI_SU_GIGABYTE, SAHPI_SU_WORD,
    SAHPI_SU_DWORD, SAHPI_SU_QWORD, SAHPI_SU_LINE,
    SAHPI_SU_HIT, SAHPI_SU_MISS, SAHPI_SU_RETRY,
    SAHPI_SU_RESET, SAHPI_SU_OVERRUN, SAHPI_SU_UNDERRUN,
    SAHPI_SU_COLLISION, SAHPI_SU_PACKETS, SAHPI_SU_MESSAGES,
    SAHPI_SU_CHARACTERS, SAHPI_SU_ERRORS, SAHPI_SU_CORRECTABLE_ERRORS,
    SAHPI_SU_UNCORRECTABLE_ERRORS
} SaHpiSensorUnitsT;

/*
** Modifier Unit Use
** This type defines how the modifier unit is used. For example: base unit ==
** meter, modifier unit == seconds, and modifier unit use ==
** SAHPI_SMUU_BASIC_OVER_MODIFIER. The resulting unit would be meters per
** second.
*/
typedef enum {
    SAHPI_SMUU_NONE = 0,
    SAHPI_SMUU_BASIC_OVER_MODIFIER,  /* Basic Unit / Modifier Unit */
    SAHPI_SMUU_BASIC_TIMES_MODIFIER  /* Basic Unit * Modifier Unit */
} SaHpiSensorModUnitUseT;

/*
** Data Format
** When IsSupported is False, the sensor does not support data readings
** (it only supports event states).  A False setting for this flag
** indicates that the rest of the structure is not meaningful.
**
** This structure encapsulates all of the various types that make up the
** definition of sensor data. For reading type of
** SAHPI_SENSOR_READING_TYPE_BUFFER, the rest of the structure
** (beyond ReadingType) is not meaningful.
**
** The Accuracy Factor is expressed as a floating point percentage
** (e.g. 0.05 = 5%) and represents statistically how close the measured
** reading is to the actual value. It is an interpreted value that
** figures in all sensor accuracies, resolutions, and tolerances.
*/

typedef struct {
    SaHpiBoolT                IsSupported;    /* Indicates if sensor data
                                                 readings are supported.*/
    SaHpiSensorReadingTypeT   ReadingType;    /* Type of value for sensor
                                                 reading. */
    SaHpiSensorUnitsT         BaseUnits;      /* Base units (meters, etc.)  */
    SaHpiSensorUnitsT         ModifierUnits;  /* Modifier unit (second, etc.)*/
    SaHpiSensorModUnitUseT    ModifierUse;    /* Modifier use(m/sec, etc.)  */
    SaHpiBoolT                Percentage;     /* Is value a percentage */
    SaHpiSensorRangeT         Range;          /* Valid range of sensor */
    SaHpiFloat64T             AccuracyFactor; /* Accuracy */
} SaHpiSensorDataFormatT;

/*
** Threshold Support
**
** These types define what threshold values are readable and writable.
** Thresholds are read/written in the same ReadingType as is used for sensor
** readings.
*/
typedef SaHpiUint8T SaHpiSensorThdMaskT;
#define SAHPI_STM_LOW_MINOR      (SaHpiSensorThdMaskT)0x01
#define SAHPI_STM_LOW_MAJOR      (SaHpiSensorThdMaskT)0x02
#define SAHPI_STM_LOW_CRIT       (SaHpiSensorThdMaskT)0x04
#define SAHPI_STM_UP_MINOR       (SaHpiSensorThdMaskT)0x08
#define SAHPI_STM_UP_MAJOR       (SaHpiSensorThdMaskT)0x10
#define SAHPI_STM_UP_CRIT        (SaHpiSensorThdMaskT)0x20
#define SAHPI_STM_UP_HYSTERESIS  (SaHpiSensorThdMaskT)0x40
```

```
#define SAHPI_STM_LOW_HYSTERESIS (SaHpiSensorThdMaskT)0x80

typedef struct {
    SaHpiBoolT            IsAccessible; /* True if the sensor
                                           supports readable or writable
                                           thresholds. If False,
                                           rest of structure is not
                                           meaningful. Sensors that have the
                                           IsAccessible flag set must also
                                           support the threshold event category.
                                           A sensor of reading type SAHPI_
                                           SENSOR_READING_TYPE_BUFFER cannot
                                           have accessible thresholds.*/
    SaHpiSensorThdMaskT   ReadThold;    /* Readable thresholds */
    SaHpiSensorThdMaskT   WriteThold;   /* Writable thresholds */
    SaHpiBoolT            Nonlinear;    /* If this flag is set, hysteresis
                                           values are calculated at the nominal
                                           sensor value. */
} SaHpiSensorThdDefnT;

/*
** Event Control
**
** This type defines how sensor event messages can be controlled (can be turned
** off and on for each type of event, etc.).
*/
typedef enum {
    SAHPI_SEC_PER_EVENT = 0,    /* Event message control per event,
                                   or by entire sensor; sensor event enable
                                   status can be changed, and assert/deassert
                                   masks can be changed */
    SAHPI_SEC_READ_ONLY_MASKS,  /* Control for entire sensor only; sensor
                                   event enable status can be changed, but
                                   assert/deassert masks cannot be changed */
    SAHPI_SEC_READ_ONLY         /* Event control not supported; sensor event
                                   enable status cannot be changed and
                                   assert/deassert masks cannot be changed */
} SaHpiSensorEventCtrlT;

/*
** Record
**
** This is the sensor resource data record which describes all of the static
** data associated with a sensor.
*/
typedef struct {
    SaHpiSensorNumT         Num;           /* Sensor Number/Index */
    SaHpiSensorTypeT        Type;          /* General Sensor Type */
    SaHpiEventCategoryT     Category;      /* Event category */
    SaHpiBoolT              EnableCtrl;    /* True if HPI User can enable
                                              or disable sensor via
                                              saHpiSensorEnableSet() */
    SaHpiSensorEventCtrlT   EventCtrl;     /* How events can be controlled */
    SaHpiEventStateT        Events;        /* Bit mask of event states
                                              supported */
    SaHpiSensorDataFormatT  DataFormat;    /* Format of the data */
    SaHpiSensorThdDefnT     ThresholdDefn; /* Threshold Definition */
    SaHpiUint32T            Oem;           /* Reserved for OEM use */
} SaHpiSensorRecT;
```

8.6 Aggregate Status

```
/*****************************************************************************
*****************************************************************************
**********                                                         **********
**********                    Aggregate Status                     **********
**********                                                         **********
*****************************************************************************
*****************************************************************************/
```

```
/* These are the default sensor numbers for aggregate status. */
#define SAHPI_DEFAGSENS_OPER (SaHpiSensorNumT)0x00000100
#define SAHPI_DEFAGSENS_PWR  (SaHpiSensorNumT)0x00000101
#define SAHPI_DEFAGSENS_TEMP (SaHpiSensorNumT)0x00000102

/* The following specifies the named range for aggregate status. */
#define SAHPI_DEFAGSENS_MIN   (SaHpiSensorNumT)0x00000100
#define SAHPI_DEFAGSENS_MAX   (SaHpiSensorNumT)0x0000010F
```

8.7 Controls

```
/*****************************************************************************
*****************************************************************************
**********                                                         **********
**********                        Controls                         **********
**********                                                         **********
*****************************************************************************
*****************************************************************************/

/* Control Number  */
typedef SaHpiInstrumentIdT SaHpiCtrlNumT;

/*
** Type of Control
**
** This enumerated type defines the different types of generic controls.
*/
typedef enum {
    SAHPI_CTRL_TYPE_DIGITAL = 0x00,
    SAHPI_CTRL_TYPE_DISCRETE,
    SAHPI_CTRL_TYPE_ANALOG,
    SAHPI_CTRL_TYPE_STREAM,
    SAHPI_CTRL_TYPE_TEXT,
    SAHPI_CTRL_TYPE_OEM = 0xC0
} SaHpiCtrlTypeT;

/*
** Control State Type Definitions
*/

/*
** Digital Control State Definition
**
** Defines the types of digital control states.
** Any of the four states may be set using saHpiControlSet().
** Only ON or OFF are appropriate returns from saHpiControlGet().
** (PULSE_ON and PULSE_OFF are transitory and end in OFF and ON states,
** respectively.)
** OFF – the control is off
** ON  - the control is on
** PULSE_OFF – the control is briefly turned off, and then turned back on
** PULSE_ON  - the control is briefly turned on, and then turned back off
**
*/
typedef enum {
    SAHPI_CTRL_STATE_OFF = 0,
    SAHPI_CTRL_STATE_ON,
    SAHPI_CTRL_STATE_PULSE_OFF,
    SAHPI_CTRL_STATE_PULSE_ON
} SaHpiCtrlStateDigitalT;

/*
** Discrete Control State Definition
*/
typedef SaHpiUint32T SaHpiCtrlStateDiscreteT;

/*
** Analog Control State Definition
*/
typedef SaHpiInt32T  SaHpiCtrlStateAnalogT;
```

```
/*
** Stream Control State Definition
*/
#define SAHPI_CTRL_MAX_STREAM_LENGTH 4
typedef struct {
    SaHpiBoolT   Repeat;       /* Repeat flag */
    SaHpiUint32T StreamLength; /* Length of the data, in bytes,
                                  stored in the stream. */
    SaHpiUint8T  Stream[SAHPI_CTRL_MAX_STREAM_LENGTH];
} SaHpiCtrlStateStreamT;

/*
** Text Control State Definition
*/
typedef SaHpiUint8T SaHpiTxtLineNumT;

/* Reserved number for sending output to all lines */
#define SAHPI_TLN_ALL_LINES (SaHpiTxtLineNumT)0x00

typedef struct {
    SaHpiTxtLineNumT    Line; /* Operate on line # */
    SaHpiTextBufferT    Text; /* Text to display */
} SaHpiCtrlStateTextT;

/*
** OEM Control State Definition
*/
#define SAHPI_CTRL_MAX_OEM_BODY_LENGTH 255
typedef struct {
    SaHpiManufacturerIdT MId;
    SaHpiUint8T BodyLength;
    SaHpiUint8T Body[SAHPI_CTRL_MAX_OEM_BODY_LENGTH]; /* OEM Specific */
} SaHpiCtrlStateOemT;

typedef union {
    SaHpiCtrlStateDigitalT  Digital;
    SaHpiCtrlStateDiscreteT Discrete;
    SaHpiCtrlStateAnalogT   Analog;
    SaHpiCtrlStateStreamT   Stream;
    SaHpiCtrlStateTextT     Text;
    SaHpiCtrlStateOemT      Oem;
} SaHpiCtrlStateUnionT;

typedef struct {
    SaHpiCtrlTypeT          Type;       /* Type of control */
    SaHpiCtrlStateUnionT    StateUnion; /* Data for control type */
} SaHpiCtrlStateT;
/*
** Control Mode Type Definition
**
** Controls may be in either AUTO mode or MANUAL mode.
**
*/
typedef enum {
    SAHPI_CTRL_MODE_AUTO,
    SAHPI_CTRL_MODE_MANUAL
} SaHpiCtrlModeT;
```

8.8 Control Resource Data Records

```
/*****************************************************************************
*****************************************************************************
**********                                                         **********
**********                Control Resource Data Records            **********
**********                                                         **********
*****************************************************************************
*****************************************************************************/

/*
```

```
** Output Type
**
**  This enumeration defines the what the control's output will be.
*/
typedef enum {
    SAHPI_CTRL_GENERIC = 0,
    SAHPI_CTRL_LED,
    SAHPI_CTRL_FAN_SPEED,
    SAHPI_CTRL_DRY_CONTACT_CLOSURE,
    SAHPI_CTRL_POWER_SUPPLY_INHIBIT,
    SAHPI_CTRL_AUDIBLE,
    SAHPI_CTRL_FRONT_PANEL_LOCKOUT,
    SAHPI_CTRL_POWER_INTERLOCK,
    SAHPI_CTRL_POWER_STATE,
    SAHPI_CTRL_LCD_DISPLAY,
    SAHPI_CTRL_OEM
} SaHpiCtrlOutputTypeT;

/*
** Specific Record Types
** These types represent the specific types of control resource data records.
*/
typedef struct {
    SaHpiCtrlStateDigitalT Default;
} SaHpiCtrlRecDigitalT;

typedef struct {
    SaHpiCtrlStateDiscreteT Default;
} SaHpiCtrlRecDiscreteT;

typedef struct {
    SaHpiCtrlStateAnalogT  Min;    /* Minimum Value */
    SaHpiCtrlStateAnalogT  Max;    /* Maximum Value */
    SaHpiCtrlStateAnalogT  Default;
} SaHpiCtrlRecAnalogT;

typedef struct {
   SaHpiCtrlStateStreamT  Default;
} SaHpiCtrlRecStreamT;

typedef struct {
    SaHpiUint8T             MaxChars; /* Maximum chars per line.
                                         If the control DataType is
                                         SAHPI_TL_TYPE_UNICODE, there will
                                         be two bytes required for each
                                         character.  This field reports the
                                         number of characters per line- not the
                                         number of bytes.  MaxChars must not be
                                         larger than the number of characters
                                         that can be placed in a single
                                         SaHpiTextBufferT structure. */
    SaHpiUint8T             MaxLines; /* Maximum # of lines */
    SaHpiLanguageT          Language; /* Language Code */
    SaHpiTextTypeT          DataType; /* Permitted Data */
    SaHpiCtrlStateTextT     Default;
} SaHpiCtrlRecTextT;

#define SAHPI_CTRL_OEM_CONFIG_LENGTH 10
typedef struct {
    SaHpiManufacturerIdT    MId;
    SaHpiUint8T             ConfigData[SAHPI_CTRL_OEM_CONFIG_LENGTH];
    SaHpiCtrlStateOemT      Default;
} SaHpiCtrlRecOemT;

typedef union {
    SaHpiCtrlRecDigitalT  Digital;
    SaHpiCtrlRecDiscreteT Discrete;
    SaHpiCtrlRecAnalogT   Analog;
    SaHpiCtrlRecStreamT   Stream;
    SaHpiCtrlRecTextT     Text;
    SaHpiCtrlRecOemT      Oem;
```

```
} SaHpiCtrlRecUnionT;

/*
** Default Control Mode Structure
** This structure tells an HPI User if the control comes up in Auto mode or
** in Manual mode, by default.  It also indicates if the mode can be
** changed (using saHpiControlSet()).  When ReadOnly is False, the mode
** can be changed from its default setting; otherwise attempting to
** change the mode will result in an error.
*/
typedef struct {
    SaHpiCtrlModeT          Mode; /* Auto or Manual */
    SaHpiBoolT              ReadOnly; /* Indicates if mode is read-only */
} SaHpiCtrlDefaultModeT;

/*
** Record Definition
** Definition of the control resource data record.
*/
typedef struct {
    SaHpiCtrlNumT         Num;         /* Control Number/Index */
    SaHpiCtrlOutputTypeT  OutputType;
    SaHpiCtrlTypeT        Type;        /* Type of control */
    SaHpiCtrlRecUnionT    TypeUnion;   /* Specific control record */
    SaHpiCtrlDefaultModeT DefaultMode; /*Indicates if the control comes up
                                         in Auto or Manual mode. */
    SaHpiBoolT            WriteOnly;   /* Indicates if the control is
                                          write-only. */
    SaHpiUint32T          Oem;         /* Reserved for OEM use */
} SaHpiCtrlRecT;
```

8.9 Inventory Data Repositories

```
/**************************************************************************
**************************************************************************
**********                                                      **********
**********               Inventory Data Repositories            **********
**********                                                      **********
**************************************************************************
**************************************************************************/
/*
** These structures are used to read and write inventory data to entity
** inventory data repositories within a resource.
*/
/*
** Inventory Data Repository ID
** Identifier for an inventory data repository.
*/
typedef SaHpiInstrumentIdT SaHpiIdrIdT;
#define SAHPI_DEFAULT_INVENTORY_ID (SaHpiIdrIdT)0x00000000

/* Inventory Data Area type definitions */
typedef enum {
    SAHPI_IDR_AREATYPE_INTERNAL_USE = 0xB0,
    SAHPI_IDR_AREATYPE_CHASSIS_INFO,
    SAHPI_IDR_AREATYPE_BOARD_INFO,
    SAHPI_IDR_AREATYPE_PRODUCT_INFO,
    SAHPI_IDR_AREATYPE_OEM = 0xC0,
    SAHPI_IDR_AREATYPE_UNSPECIFIED = 0xFF
} SaHpiIdrAreaTypeT;

/* Inventory Data Field type definitions */
typedef enum {
    SAHPI_IDR_FIELDTYPE_CHASSIS_TYPE,
    SAHPI_IDR_FIELDTYPE_MFG_DATETIME,
    SAHPI_IDR_FIELDTYPE_MANUFACTURER,
    SAHPI_IDR_FIELDTYPE_PRODUCT_NAME,
    SAHPI_IDR_FIELDTYPE_PRODUCT_VERSION,
    SAHPI_IDR_FIELDTYPE_SERIAL_NUMBER,
```

```
    SAHPI_IDR_FIELDTYPE_PART_NUMBER,
    SAHPI_IDR_FIELDTYPE_FILE_ID,
    SAHPI_IDR_FIELDTYPE_ASSET_TAG,
    SAHPI_IDR_FIELDTYPE_CUSTOM,
    SAHPI_IDR_FIELDTYPE_UNSPECIFIED = 0xFF
} SaHpiIdrFieldTypeT;

/* Inventory Data Field structure definition */
typedef struct {
    SaHpiEntryIdT        AreaId;    /* AreaId for the IDA to which */
                                    /* the Field belongs */
    SaHpiEntryIdT        FieldId;   /* Field Identifier */
    SaHpiIdrFieldTypeT   Type;      /* Field Type */
    SaHpiBoolT           ReadOnly;  /* Describes if a field is read-only. */
                                    /* All fields in a read-only area are */
                                    /* flagged as read-only as well.*/
    SaHpiTextBufferT     Field;     /* Field Data */
} SaHpiIdrFieldT;

/* Inventory Data Area header structure definition */
typedef struct {
    SaHpiEntryIdT      AreaId;      /* Area Identifier */
    SaHpiIdrAreaTypeT  Type;        /* Type of area */
    SaHpiBoolT         ReadOnly;    /* Describes if an area is read-only. */
                                    /* All area headers in a read-only IDR */
                                    /* are flagged as read-only as well.*/
    SaHpiUint32T       NumFields;   /* Number of Fields contained in Area */
} SaHpiIdrAreaHeaderT;

/* Inventory Data Repository Information structure definition */
typedef struct {
    SaHpiIdrIdT        IdrId;       /* Repository Identifier */
    SaHpiUint32T       UpdateCount; /* The count is incremented any time the */
                                    /* IDR is changed.  It rolls over to zero */
                                    /* when the maximum value is reached */
    SaHpiBoolT         ReadOnly;    /* Describes if the IDR is read-only. */
                                    /* All area headers and fields in a  */
                                    /* read-only IDR are flagged as */
                                    /* read-only as well.*/
    SaHpiUint32T       NumAreas;    /* Number of Area contained in IDR */
} SaHpiIdrInfoT;
```

8.10 Inventory Data Repository Resource Data Records

```
/*****************************************************************************
*****************************************************************************
**********                                                         **********
**********       Inventory Data Repository Resource Data Records   **********
**********                                                         **********
*****************************************************************************
*****************************************************************************/

/*
** All inventory data contained in an inventory data repository
** must be represented in the RDR repository
** with an SaHpiInventoryRecT.
*/
typedef struct {
    SaHpiIdrIdT   IdrId;
    SaHpiBoolT    Persistent;  /* True indicates that updates to IDR are
                                  automatically and immediately persisted.
                                  False indicates that updates are not
                                  immediately persisted; but optionally may be
                                  persisted via saHpiParmControl() function, as
                                  defined in implementation documentation.*/
    SaHpiUint32T  Oem;
} SaHpiInventoryRecT;
```

8.11 Watchdogs

```
/*****************************************************************************
*****************************************************************************
**********                                                         **********
**********                        Watchdogs                        **********
**********                                                         **********
*****************************************************************************
*****************************************************************************/

/*
** This section defines all of the data types associated with watchdog timers.
*/

/* Watchdog Number - Identifier for a watchdog timer. */
typedef SaHpiInstrumentIdT SaHpiWatchdogNumT;
#define SAHPI_DEFAULT_WATCHDOG_NUM (SaHpiWatchdogNumT)0x00000000

/*
** Watchdog Timer Action
**
** These enumerations represent the possible actions to be taken upon watchdog
** timer timeout and the events that are generated for watchdog actions.
*/
typedef enum {
    SAHPI_WA_NO_ACTION = 0,
    SAHPI_WA_RESET,
    SAHPI_WA_POWER_DOWN,
    SAHPI_WA_POWER_CYCLE
} SaHpiWatchdogActionT;

typedef enum {
    SAHPI_WAE_NO_ACTION = 0,
    SAHPI_WAE_RESET,
    SAHPI_WAE_POWER_DOWN,
    SAHPI_WAE_POWER_CYCLE,
    SAHPI_WAE_TIMER_INT=0x08   /* Used if Timer Preinterrupt only */
} SaHpiWatchdogActionEventT;

/*
** Watchdog Pre-timer Interrupt
**
** These enumerations represent the possible types of interrupts that may be
** triggered by a watchdog pre-timer event. The actual meaning of these
** operations may differ depending on the hardware architecture.
*/
typedef enum {
    SAHPI_WPI_NONE = 0,
    SAHPI_WPI_SMI,
    SAHPI_WPI_NMI,
    SAHPI_WPI_MESSAGE_INTERRUPT,
    SAHPI_WPI_OEM = 0x0F
} SaHpiWatchdogPretimerInterruptT;

/*
** Watchdog Timer Use
**
** These enumerations represent the possible watchdog users that may have caused
** the watchdog to expire. For instance, if watchdog is being used during power
** on self test (POST), and it expires, the SAHPI_WTU_BIOS_POST expiration type
** will be set. Most specific uses for Watchdog timer by users of HPI should
** indicate SAHPI_WTU_SMS_OS  if the use is to provide an OS-healthy heartbeat,
** or SAHPI_WTU_OEM if it is used for some other purpose.
*/
typedef enum {
    SAHPI_WTU_NONE = 0,
    SAHPI_WTU_BIOS_FRB2,
    SAHPI_WTU_BIOS_POST,
    SAHPI_WTU_OS_LOAD,
```

```
    SAHPI_WTU_SMS_OS,            /* System Management System providing
                                    heartbeat for OS */
    SAHPI_WTU_OEM,
    SAHPI_WTU_UNSPECIFIED = 0x0F
} SaHpiWatchdogTimerUseT;

/*
** Timer Use Expiration Flags
** These values are used for the Watchdog Timer Use Expiration flags in the
** SaHpiWatchdogT structure.
*/
typedef SaHpiUint8T SaHpiWatchdogExpFlagsT;
#define SAHPI_WATCHDOG_EXP_BIOS_FRB2   (SaHpiWatchdogExpFlagsT)0x02
#define SAHPI_WATCHDOG_EXP_BIOS_POST   (SaHpiWatchdogExpFlagsT)0x04
#define SAHPI_WATCHDOG_EXP_OS_LOAD     (SaHpiWatchdogExpFlagsT)0x08
#define SAHPI_WATCHDOG_EXP_SMS_OS      (SaHpiWatchdogExpFlagsT)0x10
#define SAHPI_WATCHDOG_EXP_OEM         (SaHpiWatchdogExpFlagsT)0x20

/*
** Watchdog Structure
**
** This structure is used by the saHpiWatchdogTimerGet() and
** saHpiWatchdogTimerSet() functions. The use of the structure varies slightly
** by each function.
**
** For saHpiWatchdogTimerGet() :
**
**   Log –                indicates whether or not the Watchdog is configured to
**                        issue events. True=events will be generated.
**   Running –            indicates whether or not the Watchdog is currently
**                        running or stopped. True=Watchdog is running.
**   TimerUse –           indicates the current use of the timer; one of the
**                        enumerated preset uses which was included on the last
**                        saHpiWatchdogTimerSet() function call, or through some
**                        other implementation-dependent means to start the
**                        Watchdog timer.
**   TimerAction –        indicates what action will be taken when the Watchdog
**                        times out.
**   PretimerInterrupt –  indicates which action will be taken
**                        “PreTimeoutInterval” milliseconds prior to Watchdog
**                        timer expiration.
**   PreTimeoutInterval – indicates how many milliseconds prior to timer time
**                        out the PretimerInterrupt action will be taken. If
**                        “PreTimeoutInterval” = 0, the PretimerInterrupt action
**                        will occur concurrently with “TimerAction.” HPI
**                        implementations may not be able to support millisecond
**                        resolution, and because of this may have rounded the
**                        set value to whatever resolution could be supported.
**                        The HPI implementation will return this rounded value.
**   TimerUseExpFlags –   set of five bit flags which indicate that a Watchdog
**                        timer timeout has occurred while the corresponding
**                        TimerUse value was set. Once set, these flags stay
**                        set until specifically cleared with a
**                        saHpiWatchdogTimerSet() call, or by some other
**                        implementation-dependent means.
**   InitialCount –       The time, in milliseconds, before the timer will time
**                        out after the watchdog is started/restarted, or some
**                        other implementation-dependent strobe is
**                        sent to the Watchdog. HPI implementations may not be
**                        able to support millisecond resolution, and because
**                        of this may have rounded the set value to whatever
**                        resolution could be supported.  The HPI implementation
**                        will return this rounded value.
**   PresentCount –       The remaining time in milliseconds before the timer
**                        will time out unless a saHpiWatchdogTimerReset()
**                        function call is made, or some other implementation-
**                        dependent strobe is sent to the Watchdog.
**                        HPI implementations may not be able to support
**                        millisecond resolution on watchdog timers, but will
**                        return the number of clock ticks remaining times the
**                        number of milliseconds between each tick.
```

```
**
** For saHpiWatchdogTimerSet():
**
**   Log –                 indicates whether or not the Watchdog should issue
**                         events. True=event will be generated.
**   Running –             indicates whether or not the Watchdog should be
**                         stopped before updating.
**                         True =  Watchdog is not stopped. If it is
**                                 already stopped, it will remain stopped,
**                                 but if it is running, it will continue
**                                 to run, with the countown timer reset
**                                 to the new InitialCount.  Note that
**                                 there is a race condition possible
**                                 with this setting, so it should be used
**                                 with care.
**                         False = Watchdog is stopped. After
**                                 saHpiWatchdogTimerSet() is called, a
**                                 subsequent call to
**                                 saHpiWatchdogTimerReset() is required
**                                 to start the timer.
**   TimerUse –            indicates the current use of the timer. Will control
**                         which TimerUseExpFlag is set if the timer expires.
**   TimerAction –         indicates what action will be taken when the Watchdog
**                         times out.
**   PretimerInterrupt –   indicates which action will be taken
**                         “PreTimeoutInterval” milliseconds prior to Watchdog
**                         timer expiration.
**   PreTimeoutInterval –  indicates how many milliseconds prior to timer time
**                         out the PretimerInterrupt action will be taken. If
**                         “PreTimeoutInterval” = 0, the PretimerInterrupt action
**                         will occur concurrently with “TimerAction.” HPI
**                         implementations may not be able to support millisecond
**                         resolution and may have a maximum value restriction.
**                         These restrictions should be documented by the
**                         provider of the HPI interface.
**   TimerUseExpFlags –    Set of five bit flags corresponding to the five
**                         TimerUse values. For each bit set, the corresponding
**                         Timer Use Expiration Flag will be CLEARED. Generally,
**                         a program should only clear the Timer Use Expiration
**                         Flag corresponding to its own TimerUse, so that other
**                         software, which may have used the timer for another
**                         purpose in the past can still read its TimerUseExpFlag
**                         to determine whether or not the timer expired during
**                         that use.
**   InitialCount –        The time, in milliseconds, before the timer will time
**                         out after a saHpiWatchdogTimerReset() function call is
**                         made, or some other implementation-dependent strobe is
**                         sent to the Watchdog. HPI implementations may not be
**                         able to support millisecond resolution and may have a
**                         maximum value restriction. These restrictions should
**                         be documented by the provider of the HPI interface.
**   PresentCount –        Not used on saHpiWatchdogTimerSet() function. Ignored.
**
*/

typedef struct {
    SaHpiBoolT                          Log;
    SaHpiBoolT                          Running;
    SaHpiWatchdogTimerUseT              TimerUse;
    SaHpiWatchdogActionT                TimerAction;
    SaHpiWatchdogPretimerInterruptT     PretimerInterrupt;
    SaHpiUint32T                        PreTimeoutInterval;
    SaHpiWatchdogExpFlagsT              TimerUseExpFlags;
    SaHpiUint32T                        InitialCount;
    SaHpiUint32T                        PresentCount;
} SaHpiWatchdogT;
```

8.12 Watchdog Resource Data Records

```
/******************************************************************************
******************************************************************************
**********                                                          **********
**********                 Watchdog Resource Data Records           **********
**********                                                          **********
******************************************************************************
******************************************************************************/

/*
** When the “Watchdog” capability is set in a resource, a watchdog with an
** identifier of SAHPI_DEFAULT_WATCHDOG_NUM is required. All watchdogs must be
** represented in the RDR repository with an SaHpiWatchdogRecT, including the
** watchdog with an identifier of SAHPI_DEFAULT_WATCHDOG_NUM.
*/
typedef struct {
    SaHpiWatchdogNumT  WatchdogNum;
    SaHpiUint32T       Oem;
} SaHpiWatchdogRecT;
```

8.13 Hot Swap

```
/*****************************************************************************
*****************************************************************************
**********                                                         **********
**********                        Hot Swap                         **********
**********                                                         **********
*****************************************************************************
*****************************************************************************/

/* Hot Swap Indicator State */
typedef enum {
    SAHPI_HS_INDICATOR_OFF = 0,
    SAHPI_HS_INDICATOR_ON
} SaHpiHsIndicatorStateT;

/* Hot Swap Action  */
typedef enum {
    SAHPI_HS_ACTION_INSERTION = 0,
    SAHPI_HS_ACTION_EXTRACTION
} SaHpiHsActionT;

/* Hot Swap State */
typedef enum {
    SAHPI_HS_STATE_INACTIVE = 0,
    SAHPI_HS_STATE_INSERTION_PENDING,
    SAHPI_HS_STATE_ACTIVE,
    SAHPI_HS_STATE_EXTRACTION_PENDING,
    SAHPI_HS_STATE_NOT_PRESENT
} SaHpiHsStateT;
```

8.14 Events, Part 2

```
/*****************************************************************************
*****************************************************************************
**********                                                         **********
**********                    Events, Part 2                       **********
**********                                                         **********
*****************************************************************************
*****************************************************************************/

/* Event Data Structures */

typedef enum {
    SAHPI_CRITICAL = 0,
    SAHPI_MAJOR,
    SAHPI_MINOR,
    SAHPI_INFORMATIONAL,
    SAHPI_OK,
    SAHPI_DEBUG = 0xF0,
    SAHPI_ALL_SEVERITIES = 0xFF  /* Only used with DAT and Annunciator */
                                 /* functions.  This is not a valid */
                                 /* severity for events or alarms */
} SaHpiSeverityT;


typedef enum {
    SAHPI_RESE_RESOURCE_FAILURE,
    SAHPI_RESE_RESOURCE_RESTORED,
    SAHPI_RESE_RESOURCE_ADDED
} SaHpiResourceEventTypeT;


typedef struct {
    SaHpiResourceEventTypeT  ResourceEventType;
} SaHpiResourceEventT;


/*
** Domain events are used to announce the addition of domain references
```

```
** and the removal of domain references to the DRT.
*/
typedef enum {
    SAHPI_DOMAIN_REF_ADDED,
    SAHPI_DOMAIN_REF_REMOVED
} SaHpiDomainEventTypeT;

typedef struct {
    SaHpiDomainEventTypeT  Type;        /* Type of domain event */
    SaHpiDomainIdT         DomainId;    /* Domain Id of domain added
                                           to or removed from DRT. */
} SaHpiDomainEventT;


/*
** Sensor Optional Data
**
** Sensor events may contain optional data items passed and stored with the
** event. If these optional data items are present, they will be included with
** the event data returned in response to a saHpiEventGet() or
** saHpiEventLogEntryGet() function call. Also, the optional data items may be
** included with the event data passed to the saHpiEventLogEntryAdd() function.
**
** Specific implementations of HPI may have restrictions on how much data may
** be passed to saHpiEventLogEntryAdd(). These restrictions should be documented
** by the provider of the HPI interface.
*/
typedef SaHpiUint8T SaHpiSensorOptionalDataT;

#define SAHPI_SOD_TRIGGER_READING   (SaHpiSensorOptionalDataT)0x01
#define SAHPI_SOD_TRIGGER_THRESHOLD (SaHpiSensorOptionalDataT)0x02
#define SAHPI_SOD_OEM               (SaHpiSensorOptionalDataT)0x04
#define SAHPI_SOD_PREVIOUS_STATE    (SaHpiSensorOptionalDataT)0x08
#define SAHPI_SOD_CURRENT_STATE     (SaHpiSensorOptionalDataT)0x10
#define SAHPI_SOD_SENSOR_SPECIFIC   (SaHpiSensorOptionalDataT)0x20
typedef struct {
    SaHpiSensorNumT           SensorNum;
    SaHpiSensorTypeT          SensorType;
    SaHpiEventCategoryT       EventCategory;
    SaHpiBoolT                Assertion;        /* True = Event State
                                                  asserted
                                                  False = deasserted */
    SaHpiEventStateT          EventState;       /* single state being asserted
                                                  or deasserted*/
    SaHpiSensorOptionalDataT  OptionalDataPresent;
    /* the following fields are only valid if the corresponding flag is set
       in the OptionalDataPresent field */
    SaHpiSensorReadingT       TriggerReading;   /* Reading that triggered
                                                  the event */
    SaHpiSensorReadingT       TriggerThreshold; /* Value of the threshold
                                                  that was crossed.  Will not
                                                  be present if threshold is
                                                  not readable. */
    SaHpiEventStateT          PreviousState;    /* Previous set of asserted
                                                  event states.  If multiple
                                                  event states change at once,
                                                  multiple events may be
                                                  generated for each changing
                                                  event state.  This field
                                                  should indicate the status of
                                                  the sensor event states prior
                                                  to any of the simultaneous
                                                  changes.

                                                  Thus, it will be the same in
                                                  each event generated due to
                                                  multiple simultaneous event
                                                  state changes. */

    SaHpiEventStateT          CurrentState;     /* Current set of asserted
                                                  event states. */
```

```
    SaHpiUint32T             Oem;
    SaHpiUint32T             SensorSpecific;
} SaHpiSensorEventT;

typedef SaHpiUint8T SaHpiSensorEnableOptDataT;

#define SAHPI_SEOD_CURRENT_STATE        (SaHpiSensorEnableOptDataT)0x10

typedef struct {
    SaHpiSensorNumT          SensorNum;
    SaHpiSensorTypeT         SensorType;
    SaHpiEventCategoryT      EventCategory;
    SaHpiBoolT               SensorEnable;  /* current sensor enable status  */
    SaHpiBoolT               SensorEventEnable; /* current evt enable status */
    SaHpiEventStateT         AssertEventMask;   /* current assert event mask */
    SaHpiEventStateT         DeassertEventMask; /* current deassert evt mask */
    SaHpiSensorEnableOptDataT OptionalDataPresent;
    /* the following fields are only valid if the corresponding flag is set
       in the OptionalDataPresent field */
    SaHpiEventStateT         CurrentState;      /* Current set of asserted
                                                   Event states. */
} SaHpiSensorEnableChangeEventT;

typedef struct {
    SaHpiHsStateT HotSwapState;
    SaHpiHsStateT PreviousHotSwapState;
} SaHpiHotSwapEventT;

typedef struct {
    SaHpiWatchdogNumT              WatchdogNum;
    SaHpiWatchdogActionEventT      WatchdogAction;
    SaHpiWatchdogPretimerInterruptT WatchdogPreTimerAction;
    SaHpiWatchdogTimerUseT         WatchdogUse;
} SaHpiWatchdogEventT;

/*
** The following type defines the types of events that can be reported
** by the HPI software implementation.
**
** Audit events report a discrepancy in the audit process.  Audits are typically
** performed by HA software to detect problems.  Audits may look for such things
** as corrupted data stores, inconsistent RPT information, or improperly managed
** queues.
**
** Startup events report a failure to start-up properly, or inconsistencies in
** persisted data.
*/
typedef enum {
    SAHPI_HPIE_AUDIT,
    SAHPI_HPIE_STARTUP,
    SAHPI_HPIE_OTHER
} SaHpiSwEventTypeT;

typedef struct {
    SaHpiManufacturerIdT MId;
    SaHpiSwEventTypeT    Type;
    SaHpiTextBufferT     EventData;
} SaHpiHpiSwEventT;

typedef struct {
    SaHpiManufacturerIdT MId;
    SaHpiTextBufferT     OemEventData;
} SaHpiOemEventT;

/*
** User events may be used for storing custom events created by an HPI User
** when injecting events into the Event Log using saHpiEventLogEntryAdd().
*/
typedef struct {
    SaHpiTextBufferT     UserEventData;
} SaHpiUserEventT;
```

```
typedef enum {
    SAHPI_ET_RESOURCE,
    SAHPI_ET_DOMAIN,
    SAHPI_ET_SENSOR,
    SAHPI_ET_SENSOR_ENABLE_CHANGE,
    SAHPI_ET_HOTSWAP,
    SAHPI_ET_WATCHDOG,
    SAHPI_ET_HPI_SW,
    SAHPI_ET_OEM,
    SAHPI_ET_USER
} SaHpiEventTypeT;

typedef union {
    SaHpiResourceEventT           ResourceEvent;
    SaHpiDomainEventT             DomainEvent;
    SaHpiSensorEventT             SensorEvent;
    SaHpiSensorEnableChangeEventT SensorEnableChangeEvent;
    SaHpiHotSwapEventT            HotSwapEvent;
    SaHpiWatchdogEventT           WatchdogEvent;
    SaHpiHpiSwEventT              HpiSwEvent;
    SaHpiOemEventT                OemEvent;
    SaHpiUserEventT               UserEvent;
} SaHpiEventUnionT;

typedef struct {
    SaHpiResourceIdT  Source;
    SaHpiEventTypeT   EventType;
    SaHpiTimeT        Timestamp; /*Equal to SAHPI_TIME_UNSPECIFED if time is
                                   not available; Absolute time if greater
                                   than SAHPI_TIME_MAX_RELATIVE, Relative
                                   time if less than or equal to
                                   SAHPI_TIME_MAX_RELATIVE */
    SaHpiSeverityT    Severity;
    SaHpiEventUnionT  EventDataUnion;
} SaHpiEventT;

/*
** Event Queue Status
**
**        This status word is returned to HPI Users that request it
**        when saHpiEventGet() is called.
**
*/

typedef SaHpiUint32T     SaHpiEvtQueueStatusT;

#define SAHPI_EVT_QUEUE_OVERFLOW        (SaHpiEvtQueueStatusT)0x0001
```

8.15 Annunciators

```
/*****************************************************************************
*****************************************************************************
**********                                                         **********
**********                      Annunciators                       **********
**********                                                         **********
*****************************************************************************
*****************************************************************************/

/*
** Annunciator Number
** Identifier for an Annunciator management instrument.
*/
typedef SaHpiInstrumentIdT SaHpiAnnunciatorNumT;

/*
** The following data type is equivalent to the AIS data type SaNameT.
** it is defined in this header file, so that inclusion of the AIS
** header file is not required.  This data type is based on version 1.0
** of the AIS specification
```

```
*/
#define SA_HPI_MAX_NAME_LENGTH 256

typedef struct {
    SaHpiUint16T  Length;
    unsigned char Value[SA_HPI_MAX_NAME_LENGTH];
} SaHpiNameT;

/*
** Enumeration of Announcement Types
*/
typedef enum {
    SAHPI_STATUS_COND_TYPE_SENSOR,
    SAHPI_STATUS_COND_TYPE_RESOURCE,
    SAHPI_STATUS_COND_TYPE_OEM,
    SAHPI_STATUS_COND_TYPE_USER
} SaHpiStatusCondTypeT;


/* Condition structure definition */
typedef struct {

    SaHpiStatusCondTypeT Type;         /* Status Condition Type */
    SaHpiEntityPathT     Entity;       /* Entity assoc. with status condition */
    SaHpiDomainIdT       DomainId;     /* Domain associated with status.
                                          May be SAHPI_UNSPECIFIED_DOMAIND_ID
                                          meaning current domain, or domain
                                          not meaningful for status condition*/
    SaHpiResourceIdT     ResourceId;   /* Resource associated with status.
                                          May be SAHPI_UNSPECIFIED_RESOURCE_ID
                                          if Type is SAHPI_STATUS_COND_USER.
                                          Must be set to valid ResourceId in
                                          domain specified by DomainId,
                                          or in current domain, if DomainId
                                          is SAHPI_UNSPECIFIED_DOMAIN_ID */
    SaHpiSensorNumT      SensorNum;    /* Sensor associated with status
                                          Only valid if Type is
                                          SAHPI_STATUS_COND_TYPE_SENSOR */
    SaHpiEventStateT     EventState;   /* Sensor event state.
                                          Only valid if Type is
                                          SAHPI_STATUS_COND_TYPE_SENSOR. */
    SaHpiNameT           Name;         /* AIS compatible identifier associated
                                          with Status condition */
    SaHpiManufacturerIdT Mid;          /* Manufacturer Id associated with
                                          status condition, required when type
                                          is SAHPI_STATUS_COND_TYPE_OEM. */
    SaHpiTextBufferT     Data;         /* Optional Data associated with
                                          Status condition */
} SaHpiConditionT;


/* Announcement structure definition */
typedef struct {
    SaHpiEntryIdT        EntryId;      /* Announcment Entry Id */
    SaHpiTimeT           Timestamp;    /* Time when announcement added to set */
    SaHpiBoolT           AddedByUser;  /* True if added to set by HPI User,
                                          False if added automatically by
                                          HPI implementation */
    SaHpiSeverityT       Severity;     /* Severity of announcement */
    SaHpiBoolT           Acknowledged; /* Acknowledged flag */
    SaHpiConditionT      StatusCond;   /* Detailed status condition */
} SaHpiAnnouncementT;


/* Annunciator Mode – defines who may add or delete entries in set.  */

typedef enum {
    SAHPI_ANNUNCIATOR_MODE_AUTO,
    SAHPI_ANNUNCIATOR_MODE_USER,
    SAHPI_ANNUNCIATOR_MODE_SHARED
} SaHpiAnnunciatorModeT;
```

8.16 Annunciator Resource Data Records

```
/*****************************************************************************
*****************************************************************************
**********                                                          **********
**********              Annunciator Resource Data Records           **********
**********                                                          **********
*****************************************************************************
*****************************************************************************/

/*
** The following enumerated type defines the possible output types
** which can be associated with an Annunciator Management Instrument
*/
typedef enum {
    SAHPI_ANNUNCIATOR_TYPE_LED = 0,
    SAHPI_ANNUNCIATOR_TYPE_DRY_CONTACT_CLOSURE,
    SAHPI_ANNUNCIATOR_TYPE_AUDIBLE,
    SAHPI_ANNUNCIATOR_TYPE_LCD_DISPLAY,
    SAHPI_ANNUNCIATOR_TYPE_MESSAGE,
    SAHPI_ANNUNCIATOR_TYPE_COMPOSITE,
    SAHPI_ANNUNCIATOR_TYPE_OEM
} SaHpiAnnunciatorTypeT;


/*
** All annunciator management instruments
** must be represented in the RDR repository
** with an SaHpiAnnunciatorRecT.
*/
typedef struct {
    SaHpiAnnunciatorNumT      AnnunciatorNum;
    SaHpiAnnunciatorTypeT     AnnunciatorType; /* Annunciator Output Type */
    SaHpiBoolT                ModeReadOnly;    /* if True, Mode may
                                                  not be changed by HPI User */
    SaHpiUint32T              MaxConditions;   /* maximum number of conditions
                                                  that can be held in current
                                                  set.  0 means no fixed
                                                  limit.  */
    SaHpiUint32T              Oem;
} SaHpiAnnunciatorRecT;
```

8.17 Resource Data Records

```
/*****************************************************************************
*****************************************************************************
**********                                                          **********
**********                      Resource Data Record                **********
**********                                                          **********
*****************************************************************************
*****************************************************************************/

/*
** The following describes the different types of records that exist within a
** RDR repository and the RDR super-structure to all of the specific RDR types
** (sensor, inventory data, watchdog, etc.).
*/
typedef enum {
    SAHPI_NO_RECORD,
    SAHPI_CTRL_RDR,
    SAHPI_SENSOR_RDR,
    SAHPI_INVENTORY_RDR,
    SAHPI_WATCHDOG_RDR,
    SAHPI_ANNUNCIATOR_RDR
} SaHpiRdrTypeT;

typedef union {
    SaHpiCtrlRecT        CtrlRec;
```

```
    SaHpiSensorRecT      SensorRec;
    SaHpiInventoryRecT   InventoryRec;
    SaHpiWatchdogRecT    WatchdogRec;
    SaHpiAnnunciatorRecT AnnunciatorRec;
} SaHpiRdrTypeUnionT;

typedef struct {
    SaHpiEntryIdT        RecordId;
    SaHpiRdrTypeT        RdrType;
    SaHpiEntityPathT     Entity;        /* Entity to which this RDR relates. */
    SaHpiBoolT           IsFru;         /* Entity is a FRU.  This field is
                                           Only valid if the Entity path given
                                           in the “Entity” field is different
                                           from the Entity path in the RPT
                                           entry for the resource. */
    SaHpiRdrTypeUnionT   RdrTypeUnion;
    SaHpiTextBufferT     IdString;
} SaHpiRdrT;
```

8.18 Parameter Control

```
/*****************************************************************************
*****************************************************************************
**********                                                         **********
**********                    Parameter Control                    **********
**********                                                         **********
*****************************************************************************
*****************************************************************************/

typedef enum {
    SAHPI_DEFAULT_PARM = 0,
    SAHPI_SAVE_PARM,
    SAHPI_RESTORE_PARM
} SaHpiParmActionT;
```

8.19 Reset

```
/*****************************************************************************
*****************************************************************************
**********                                                         **********
**********                    Reset                                **********
**********                                                         **********
*****************************************************************************
*****************************************************************************/

typedef enum {
    SAHPI_COLD_RESET = 0,
    SAHPI_WARM_RESET,
    SAHPI_RESET_ASSERT,
    SAHPI_RESET_DEASSERT
} SaHpiResetActionT;
```

8.20 Power

```
/*****************************************************************************
*****************************************************************************
**********                                                         **********
**********                    Power                                **********
**********                                                         **********
*****************************************************************************
*****************************************************************************/

typedef enum {
    SAHPI_POWER_OFF = 0,
    SAHPI_POWER_ON,
    SAHPI_POWER_CYCLE
} SaHpiPowerStateT;
```

8.21 Resource Presence Table

```
/******************************************************************************
******************************************************************************
**********                                                          **********
**********                  Resource Presence Table                 **********
**********                                                          **********
******************************************************************************
******************************************************************************/

/*  This section defines the types associated with the RPT. */

/*
** GUID – Globally Unique Identifier
**
** The format if the ID follows that specified by the Wired for Management
** Baseline, Version 2.0 specification.  HPI uses version 1 of the GUID
** format, with a 3-bit variant field of 10x (where x indicates 'don't care')
*/
typedef SaHpiUint8T    SaHpiGuidT[16];

/*
** Resource Info Type Definitions
**
**
** SaHpiResourceInfoT contains static configuration data concerning the
** management controller associated with the resource, or the resource itself.
** Note this information is used to describe the resource; that is, the piece of
** infrastructure which manages an entity (or multiple entities) – NOT the
** entities for which the resource provides management. The purpose of the
** SaHpiResourceInfoT structure is to provide information that an HPI User may
** need in order to interact correctly with the resource (e.g., recognize a
** specific management controller which may have defined OEM fields in sensors,
** OEM controls, etc.).
**
** The GUID is used to uniquely identify a Resource. A GUID value of zero is not
** valid and indicates that the Resource does not have an associated GUID.
**
** All of the fields in the following structure may or may not be used by a
** given resource.
*/
typedef struct {
    SaHpiUint8T            ResourceRev;
    SaHpiUint8T            SpecificVer;
    SaHpiUint8T            DeviceSupport;
    SaHpiManufacturerIdT   ManufacturerId;
    SaHpiUint16T           ProductId;
    SaHpiUint8T            FirmwareMajorRev;
    SaHpiUint8T            FirmwareMinorRev;
    SaHpiUint8T            AuxFirmwareRev;
    SaHpiGuidT             Guid;
} SaHpiResourceInfoT;

/*
** Resource Capabilities
**
** This definition defines the capabilities of a given resource. One resource
** may support any number of capabilities using the bit mask. Because each entry
** in an RPT will have the SAHPI_CAPABILITY_RESOURCE bit set, zero is not a
** valid value for the capability flag, and is thus used to indicate "no RPT
** entry present" in some function calls.
**
** SAHPI_CAPABILITY_RESOURCE
** SAHPI_CAPABILITY_EVT_DEASSERTS
**   Indicates that all sensors on the resource have the property that their
**   Assertion and Deassertion event enable flags are the same. That is,
**   for all event states whose assertion triggers an event, it is
**   guaranteed that the deassertion of that event will also
**   trigger an event. Thus, an HPI User may track the state of sensors on the
**   resource by monitoring events rather than polling for state changes.
```

```
** SAHPI_CAPABILITY_AGGREGATE_STATUS
** SAHPI_CAPABILITY_CONFIGURATION
** SAHPI_CAPABILITY_MANAGED_HOTSWAP
**   Indicates that the resource supports the full managed hot swap model.
**   Since hot swap only makes sense for field-replaceable units, the
**   SAHPI_CAPABILITY_FRU capability bit must also be set for this resource.
** SAHPI_CAPABILITY_WATCHDOG
** SAHPI_CAPABILITY_CONTROL
** SAHPI_CAPABILITY_FRU
**   Indicates that the resource is a field-replaceable unit; i.e., it is
**   capable of being removed and replaced in a live system. If
**   SAHPI_CAPABILITY_MANAGED_HOTSWAP is also set, the resource supports
**   the full hot swap model.  If SAHPI_CAPABILITY_MANAGED_HOTSWAP is not
**   set, the resource supports the simplified hot swap model.
** SAHPI_CAPABILITY_ANNUNCIATOR
** SAHPI_CAPABILITY_POWER
** SAHPI_CAPABILITY_RESET
** SAHPI_CAPABILITY_INVENTORY_DATA
** SAHPI_CAPABILITY_EVENT_LOG
** SAHPI_CAPABILITY_RDR
**   Indicates that a resource data record (RDR) repository is supplied
**   by the resource. Since the existence of an RDR is mandatory for all
**   management instruments, this
**   capability must be asserted if the resource
**   contains any sensors, controls, watchdog timers, or inventory
**   data repositories.
** SAHPI_CAPABILITY_SENSOR
*/

typedef SaHpiUint32T SaHpiCapabilitiesT;
#define SAHPI_CAPABILITY_RESOURCE         (SaHpiCapabilitiesT)0X40000000
#define SAHPI_CAPABILITY_EVT_DEASSERTS    (SaHpiCapabilitiesT)0x00008000
#define SAHPI_CAPABILITY_AGGREGATE_STATUS (SaHpiCapabilitiesT)0x00002000
#define SAHPI_CAPABILITY_CONFIGURATION    (SaHpiCapabilitiesT)0x00001000
#define SAHPI_CAPABILITY_MANAGED_HOTSWAP  (SaHpiCapabilitiesT)0x00000800
#define SAHPI_CAPABILITY_WATCHDOG         (SaHpiCapabilitiesT)0x00000400
#define SAHPI_CAPABILITY_CONTROL          (SaHpiCapabilitiesT)0x00000200
#define SAHPI_CAPABILITY_FRU              (SaHpiCapabilitiesT)0x00000100
#define SAHPI_CAPABILITY_ANNUNCIATOR      (SaHpiCapabilitiesT)0x00000040
#define SAHPI_CAPABILITY_POWER            (SaHpiCapabilitiesT)0x00000020
#define SAHPI_CAPABILITY_RESET            (SaHpiCapabilitiesT)0x00000010
#define SAHPI_CAPABILITY_INVENTORY_DATA   (SaHpiCapabilitiesT)0x00000008
#define SAHPI_CAPABILITY_EVENT_LOG        (SaHpiCapabilitiesT)0x00000004
#define SAHPI_CAPABILITY_RDR              (SaHpiCapabilitiesT)0x00000002
#define SAHPI_CAPABILITY_SENSOR           (SaHpiCapabilitiesT)0x00000001

/*
** Resource Managed Hot Swap Capabilities
**
** This definition defines the managed hot swap capabilities of a given
** resource.
**
** SAHPI_HS_CAPABILITY_AUTOEXTRACT_READ_ONLY
** This capability indicates if the hot swap autoextract timer is read-only.
** SAHPI_HS_CAPABILITY_INDICATOR_SUPPORTED
** Indicates whether or not the resource has a hot swap indicator.
*/

typedef SaHpiUint32T SaHpiHsCapabilitiesT;
#define SAHPI_HS_CAPABILITY_AUTOEXTRACT_READ_ONLY \
   (SaHpiHsCapabilitiesT)0x80000000
#define SAHPI_HS_CAPABILITY_INDICATOR_SUPPORTED \
   (SaHpiHsCapabilitiesT)0X40000000
/*
** RPT Entry
**
** This structure is used to store the RPT entry information.
**
** The ResourceCapabilities field definies the capabilities of the resource.
** This field must be non-zero for all valid resources.
**
```

```
** The HotSwapCapabilities field denotes the capabilities of the resource,
** specifically related to hot swap.  This field is only valid if the
** resource supports managed hot swap, as indicated by the
** SAHPI_CAPABILITY_MANAGED_HOT_SWAP resource capability.
**
** The ResourceTag is a data field within an RPT entry available to the HPI
** User for associating application specific data with a resource.  The HPI
** User supplied data is purely informational and is not used by the HPI
** implementation, domain, or associated resource.  For example, an HPI User
** can set the resource tag to a “descriptive” value, which can be used to
** identify the resource in messages to a human operator.
*/
typedef struct {
    SaHpiEntryIdT        EntryId;
    SaHpiResourceIdT     ResourceId;
    SaHpiResourceInfoT   ResourceInfo;
    SaHpiEntityPathT     ResourceEntity;  /* If resource manages a FRU, entity
                                             path of the FRU */
                                         /* If resource manages a single
                                             entity, entity path of that
                                             entity. */
                                         /* If resource manages multiple
                                             entities, the entity path of the
                                             “primary” entity managed by the
                                             resource */
                                         /* Must be set to the same value in
                                             every domain which contains this
                                             resource */
    SaHpiCapabilitiesT   ResourceCapabilities;  /* Must be non-0. */
    SaHpiHsCapabilitiesT HotSwapCapabilities; /* Indicates the hot swap
                                                 capabilities of the resource */
    SaHpiSeverityT       ResourceSeverity; /* Indicates the criticality that
                                              should be raised when the resource
                                              is not responding   */
    SaHpiBoolT           ResourceFailed;  /* Indicates that the resource is not
                                             currently functional */
    SaHpiTextBufferT     ResourceTag;
} SaHpiRptEntryT;
```

8.22 Domains

```
/*****************************************************************************
*****************************************************************************
**********                                                         **********
**********                     Domain Information                  **********
**********                                                         **********
*****************************************************************************
*****************************************************************************/

/*  This section defines the types associated with the domain controller. */

/*
** Domain Capabilities
**
** This definition defines the capabilities of a given domain. A domain
** may support any number of capabilities using the bit mask.
**
** SAHPI_DOMAIN_CAP_AUTOINSERT_READ_ONLY
**   Indicates that the domain auto insert timeout value is read-only
**   and may not be modified using the saHpiHotSwapAutoInsertTimeoutSet()
**   function.
*/

typedef SaHpiUint32T SaHpiDomainCapabilitiesT;
#define SAHPI_DOMAIN_CAP_AUTOINSERT_READ_ONLY \
   (SaHpiDomainCapabilitiesT)0X00000001

/*
** Domain Info
**
```

```
** This structure is used to store the information regarding the domain
** including information regarding the domain reference table (DRT) and
** the resource presence table (RPT).
**
** The DomainTag field is an informational value that supplies an HPI User
** with naming information for the domain.
**
** NOTE: Regarding timestamps - If the implementation cannot supply an absolute
** timestamp, then it may supply a timestamp relative to some system-defined
** epoch, such as system boot. The value SAHPI_TIME_UNSPECIFIED indicates that
** the time of the update cannot be determined. Otherwise, If the value is less
** than or equal to SAHPI_TIME_MAX_RELATIVE, then it is relative; if it is
** greater than SAHPI_TIME_MAX_RELATIVE, then it is absolute.
**
** The GUID is used to uniquely identify a domain. A GUID value of zero is not
** valid and indicates that the domain does not have an associated GUID.
*/

typedef struct {
    SaHpiDomainIdT    DomainId;        /* Unique Domain Id associated with
                                          domain */
    SaHpiDomainCapabilitiesT DomainCapabilities;    /* Domain Capabilities */
    SaHpiBoolT        IsPeer;          /* Indicates that this domain
                                          participates in a peer
                                          relationship. */
    SaHpiTextBufferT  DomainTag;       /* Information tag associated with
                                          domain */
    SaHpiUint32T      DrtUpdateCount;  /* This count is incremented any time the
                                          table is changed. It rolls over to
                                          zero when the maximum value is
                                          reached  */
    SaHpiTimeT        DrtUpdateTimestamp; /* This timestamp is set any time the
                                             DRT table is changed. */
    SaHpiUint32T      RptUpdateCount;   /* This count is incremented any time
                                           the RPT is changed. It rolls over
                                           to zero when the maximum value is
                                           reached  */
    SaHpiTimeT        RptUpdateTimestamp; /* This timestamp is set any time the
                                             RPT table is changed. */
    SaHpiUint32T       DatUpdateCount;  /* This count is incremented any time
                                           the DAT is changed. It rolls over to
                                           zero when the maximum value is
                                           reached */
    SaHpiTimeT         DatUpdateTimestamp; /* This timestamp is set any time the
                                              DAT is changed. */
    SaHpiUint32T       ActiveAlarms;     /* Count of active alarms in the DAT */
    SaHpiUint32T       CriticalAlarms;   /* Count of active critical alarms in
                                            the DAT */
    SaHpiUint32T       MajorAlarms;      /* Count of active major alarms in the
                                            DAT */
    SaHpiUint32T       MinorAlarms;      /* Count of active minor alarms in the
                                            DAT */
    SaHpiUint32T       DatUserAlarmLimit; /* Maximum User Alarms that can be
                                             added to DAT.  0=no fixed limit */
    SaHpiBoolT         DatOverflow;       /* Set to True if there are one
                                             or more non-User Alarms that
                                             are missing from the DAT because
                                             of space limitations */
    SaHpiGuidT        Guid;              /* GUID associated with domain.*/
} SaHpiDomainInfoT;

/*
** DRT Entry
**
** This structure is used to store the DRT entry information.
**
*/
typedef struct {
    SaHpiEntryIdT        EntryId;
    SaHpiDomainIdT       DomainId;  /* The Domain ID referenced by this entry */
    SaHpiBoolT           IsPeer;    /* Indicates if this domain reference
```

```
                                        is a peer. If not, the domain reference
                                        is a tier. */
} SaHpiDrtEntryT;


/*
** DAT Entry
**
** This structure is used to store alarm informatin in the DAT
**
*/


typedef SaHpiEntryIdT SaHpiAlarmIdT;

typedef struct {
    SaHpiAlarmIdT        AlarmId;      /* Alarm Id */
    SaHpiTimeT           Timestamp;    /* Time when alarm added to DAT */
    SaHpiSeverityT       Severity;     /* Severity of alarm */
    SaHpiBoolT           Acknowledged; /* Acknowledged flag */
    SaHpiConditionT     AlarmCond;   /* Detailed alarm condition */
} SaHpiAlarmT;
```

8.23 Event Log

```
/*****************************************************************************
*****************************************************************************
**********                                                         **********
**********                      Event Log                          **********
**********                                                         **********
*****************************************************************************
*****************************************************************************/
/* This section defines the types associated with the Event Log. */
/*
** Event Log Information
**
** The Entries entry denotes the number of active entries contained in the Event
**   Log.
** The Size entry denotes the total number of entries the Event Log is able to
**   hold.
** The UserEventMaxSize entry indicates the maximum size of the text buffer
**   data field in an HPI User event that is supported by the Event Log
**   implementation.  If the implementation does not enforce a more restrictive
**   data length, it should be set to SAHPI_MAX_TEXT_BUFFER_LENGTH.
** The UpdateTimestamp entry denotes the time of the last update to the Event
**   Log.
** The CurrentTime entry denotes the Event Log's idea of the current time; i.e
**   the timestamp that would be placed on an entry if it was added now.
** The Enabled entry indicates whether the Event Log is enabled. If the Event
**   Log is "disabled" no events generated within the HPI implementation will be
**   added to the Event Log. Events may still be added to the Event Log with
**   the saHpiEventLogEntryAdd() function. When the Event Log is "enabled"
**   events may be automatically added to the Event Log as they are generated
**   in a resource or a domain, however, it is implementation-specific which
**   events are automatically added to any Event Log.
** The OverflowFlag entry indicates the Event Log has overflowed. Events have
**   been dropped or overwritten due to a table overflow.
** The OverflowAction entry indicates the behavior of the Event Log when an
**   overflow occurs.
** The OverflowResetable entry indicates if the overflow flag can be
**   cleared by an HPI User with the saHpiEventLogOverflowReset() function.
*/
typedef enum {
    SAHPI_EL_OVERFLOW_DROP,       /* New entries are dropped when Event Log is
                                     full*/
    SAHPI_EL_OVERFLOW_OVERWRITE  /* Event Log overwrites existing entries
                                     when Event Log is full */
} SaHpiEventLogOverflowActionT;

typedef struct {
    SaHpiUint32T                  Entries;
```

```
    SaHpiUint32T                  Size;
    SaHpiUint32T                  UserEventMaxSize;
    SaHpiTimeT                    UpdateTimestamp;
    SaHpiTimeT                    CurrentTime;
    SaHpiBoolT                    Enabled;
    SaHpiBoolT                    OverflowFlag;
    SaHpiBoolT                    OverflowResetable;
    SaHpiEventLogOverflowActionT  OverflowAction;
} SaHpiEventLogInfoT;
/*
** Event Log Entry
** These types define the Event Log entry.
*/
typedef SaHpiUint32T SaHpiEventLogEntryIdT;
/* Reserved values for Event Log entry IDs */
#define SAHPI_OLDEST_ENTRY    (SaHpiEventLogEntryIdT)0x00000000
#define SAHPI_NEWEST_ENTRY    (SaHpiEventLogEntryIdT)0xFFFFFFFF
#define SAHPI_NO_MORE_ENTRIES (SaHpiEventLogEntryIdT)0xFFFFFFFE

typedef struct {
    SaHpiEventLogEntryIdT EntryId;   /* Entry ID for record */
    SaHpiTimeT            Timestamp; /* Time at which the event was placed
                                        in the Event Log. If less than or equal to
                                        SAHPI_TIME_MAX_RELATIVE, then it is
                                        relative; if it is greater than SAHPI_TIME_
                                        MAX_RELATIVE, then it is absolute. */
    SaHpiEventT           Event;     /* Logged Event */
} SaHpiEventLogEntryT;
```

A. Usage Descriptions

A.1 Watchdog Timer Example Usage

The following time line shows an example of the operation of the watchdog. The case shown here illustrates a watchdog timer that supports pre-timeout, with the Message Interrupt action set; other use cases exist, but are not illustrated here:

```
T0:  saHpiWatchdogTimerSet() called.
Log = SAHPI_TRUE;
Running = SAHPI_FALSE;
TimerUse = SAHPI_WTU_SMS_OS;
TimerAction = SAHPI_WA_RESET;
PretimerInterrupt = SAHPI_WPI_MESSAGE_INTERRUPT;
PreTimeoutInterval = 3000; (3 seconds)
TimerUseExpFlags = SAHPI_WATCHDOG_EXP_SMS_OS;
InitialCount = 25000;  (25 seconds)

Watchdog timer is configured to reset resource after timeout of 25 seconds,
with a 3 second pre-timeout interrupt. Event will be generated at the pre-timer interval
prior to timeout and on timeout. SAHPI_WATCHDOG_EXP_SMS_OS Expiration flag has been cleared.
Watchdog timer is currently stopped.

T0+10 seconds:  saHpiWatchdogTimerGet() called. Values returned:
Log = SAHPI_TRUE;
Running = SAHPI_FALSE;
TimerUse = SAHPI_WTU_SMS_OS;
TimerAction = SAHPI_WATCHDOG_RESET;
PretimerInterrupt = SAHPI_WPI_MESSAGE_INTERRUPT;
PreTimeoutInterval = 3000;
TimerUseExpFlags = SAHPI_WATCHDOG_EXP_SMS_OS bit will be CLEARED. Other bits
may be set or cleared, as they were before initial saHpiWatchdogTimerSet() call.
InitialCount = 25000;
PresentCount = undefined

T0+30 seconds: saHpiWatchDogTimerReset() called.
Timer starts to run.

T0+45 seconds: saHpiWatchdogTimerGet() called. Values returned:
Log = SAHPI_TRUE;
Running = SAHPI_TRUE;
TimerUse = SAHPI_WTU_SMS_OS;
TimerAction = SAHPI_WA_RESET;
PretimerInterrupt = SAHPI_WPI_MESSAGE_INTERRUPT;
PreTimeoutInterval = 3000;
TimerUseExpFlags = SAHPI_WATCHDOG_EXP_SMS_OS bit will be CLEARED. Other bits
may be set or cleared, as they were before initial saHpiWatchdogTimerSet() call.
InitialCount = 25000;
PresentCount = 10000; (It is 15 seconds since last reset, so there are 10
Seconds before timeout)

T0+50 seconds:  saHpiWatchdogTimerReset() called.

T0+55 seconds:  saHpiWatchdogTimerGet() called. Values returned:
Log = SAHPI_TRUE;
Running = SAHPI_TRUE;
TimerUse = SAHPI_WTU_SMS_OS;
TimerAction = SAHPI_WA_RESET;
PretimerInterrupt = SAHPI_WPI_MESSAGE_INTERRUPT;
PreTimeoutInterval = 3000;
TimerUseExpFlags = SAHPI_WATCHDOG_EXP_SMS_OS bit will be CLEARED. Other bits
may be set or cleared, as they were before initial saHpiWatchdogTimerSet() call.
InitialCount = 25000;
PresentCount = 20000; (It is 5 seconds since last reset, so there are 20
seconds before timeout)
```

```
T0+72 seconds:  Three seconds prior to timeout - Watchdog timer initiates
“Message Interrupt” on resource.
Event is issued:
    Source = ResourceId of resource hosting watchdog;
    Event Type = SAHPI_ES_WATCHDOG;
    Severity = SAHPI_MAJOR;
    EventDataUnion.WatchdogEvent.WatchdogNum = Watchdog  number for watchdog that
    reached pre-timer interval;
    EventDataUnion.WatchdogEvent.WatchdogAction = SAHPI_WAE_TIMER_INT;
    EventDataUnion.WatchdogEvent.WatchdogPreTimerAction = SAHPI_WPI_MESSAGE_INTERRUPT;
    EventDataUnion.WatchdogEvent.WatchdogUse = SAHPI_WTU_SMS_OS;

T0+73.5 seconds:  saHpiWatchdogTimerGet() called. Values returned:
Log = SAHPI_TRUE;
Running = SAHPI_TRUE;
TimerUse = SAHPI_WTU_SMS_OS;
TimerAction = SAHPI_WA_RESET;
PretimerInterrupt = SAHPI_WPI_MESSAGE_INTERRUPT;
PreTimeoutInterval = 3000;
TimerUseExpFlags = SAHPI_WATCHDOG_EXP_SMS_OS bit will be CLEARED. Other bits
may be set or cleared, as they were before initial saHpiWatchdogTimerSet() call.
InitialCount = 25000;
PresentCount = 1500; (It is 23.5 seconds since last reset, so there are 1.5
seconds before timeout)

T0+74 seconds: saHpiWatchdogTimerReset() called.
Because the pre-timeout interrupt has already occurred, this function call has
no effect, and will return the error SA_ERR_HPI_INVALID_REQUEST.

T0+75 seconds: Timer expires, and:
Resource is reset;
Event is issued:
    Source = ResourceId of resource hosting watchdog;
    Event Type = SAHPI_ES_WATCHDOG;
    Severity = SAHPI_MAJOR;
    EventDataUnion.WatchdogEvent.WatchdogNum = Watchdog  number for watchdog that
    fired;
    EventDataUnion.WatchdogEvent.WatchdogAction = SAHPI_WAE_RESET;
    EventDataUnion.WatchdogEvent.WatchdogUse = SAHPI_WTU_SMS_OS;

T0+80 seconds:  saHpiWatchdogTimerGet() called. Values returned:
    Log = SAHPI_FALSE;
    Running = SAHPI_FALSE;
    TimerUse = SAHPI_WTU_SMS_OS;
    TimerAction = SAHPI_WA_RESET;
    PretimerInterrupt = SAHPI_WPI_MESSAGE_INTERRUPT;
    PreTimeoutInterval = 3000;
    TimerUseExpFlags = SAHPI_WATCHDOG_EXP_SMS_OS bit will be set. Other bits may be
    set or cleared, as they were before initial saHpiWatchdogTimerSet() call.
    InitialCount = 25000;
    PresentCount = undefined
```

A.2 Managing a Fantray FRU from an AlarmCard Resource

Generally, all sensors, controls, etc., associated with a single entity are associated with a single resource. There is one case, however, where this may not be appropriate. If a management controller in the system (e.g., an alarm card) is responsible for managing entities on other FRU's (e.g., fans on a non-intelligent removable fantray), there are two different occurrences which can impact the manageability of those entities. If a FRU containing managed entities (i.e., fantray) is removed from the system, those entities become unmanageable. But, also, if the managing FRU (i.e., alarm card) fails, then all entities it manages become unmanageable. To correctly model this, each of these “remotely managed” FRU entities needs to be associated with two different resources: one associated with its own local FRU (i.e., fantray), and one associated with the managing FRU (i.e., alarm card). An acceptable method to reflect this situation in the HPI is to create a resource for the managed entities' FRU (i.e., fantray) which contains no sensors, controls, etc., but provides only the “hot swap” capability for the FRU. Each of the sensors, controls, etc., for the FRU would be included in the RDR table for the resource associated with the managing FRU (i.e., alarm card). When a management instrument (sensor, control, etc.) located in one resource is associated with a different FRU in this way, the *IsFru* flag is set in the Resource Data Record for that management instrument to indicate that fact.

As a further example, it is permissible for two alarm card FRU's to manage a single fan (or multiple fans) on a fantray FRU. In this case the two alarm cards would typically be represented via two distinct resources, while the fantray FRU would also be represented as a resource. Note that in this example if each alarm card included a "fan speed" sensor for the same fan on the fantray FRU, those would be two different sensors in the system. There is nothing that automatically tells an HPI User that those two sensors are measuring the same (physical) thing. An HPI User would have to use its own intelligence to know that it doesn't make sense for the same fan to be reporting two different speeds, for example. By the same token if each alarm card included a "fan speed" control for the same fan, it would be up to an HPI User to establish rules for access to the control (i.e., an HPI User should probably control the fan through an "active" but not a "standby" alarm card).

The following rules apply to resources:

1. The entity paths in the RPT of each resource, throughout an HPI implementation, must be unique -- thus the entity paths in the RPT's of two distinct resources cannot be identical. The RPT entity path is the unambiguous identifier of a resource wherever it appears.

2. The same resource can show up in separate domains, possibly with differing resource numbers in each of these domains. If the entity paths in RPT's of resources in separate domains are identical, than it can be concluded that these resources are really the same resource showing up in those domains.

Note: Where the same resource shows up in separate domains, all representations of that resource shall be identical in all respects except for a possibly differing resource number. It is therefore acceptable for an implementation to use one physical copy of a resource and present it in the separate domains.

3. A given resource may only show up once in any given domain. Therefore, the entity paths in RPT's of resources in the same domain can never be identical, since identical entity paths would imply that the same resource is showing up multiple times in the same domain. For example:

   Not allowed: The following implies that the same resource is showing up twice in the same domain (the domain numbers and entity paths are identical):

   Domain 0, resource 1, RPT entity path: /SBC Blade 3/chassis 2

   Domain 0, resource 5, RPT entity path: /SBC Blade 3/chassis 2

   Permitted: The following implies that the same resource is showing up in different domains with two different resource numbers (note the differing domain numbers). The representations of the resource in domains 0 and 2 are identical in all respects, except for the differing resource numbers.

   Domain 0, resource 1, RPT entity path: /SBC Blade 3/chassis 2

   Domain 2, resource 5, RPT entity path: /SBC Blade 3/chassis 2