Document Number: DSP0248

Date: 16 March 2009

Version: 1.0.0

Platform Level Data Model (PLDM) for Platform Monitoring and Control Specification

Document Type: Specification

Document Status: DMTF Standard

Document Language: E

CONTENTS

Figures

Tables

Foreword

The *Platform Level Data Model (PLDM) for Platform Monitoring and Control Specification* (DSP0248) was prepared by the Platform Management Components Intercommunications (PMCI Working Group of the DMTF.

DMTF is a not-for-profit association of industry members dedicated to promoting enterprise and systems management and interoperability.

Introduction

The *Platform Level Data Model (PLDM) Monitoring and Control Specification* defines messages and data structures for discovering, describing, initializing, and accessing sensors and effecters within the management controllers and management devices of a platform management subsystem. Additional functions related to platform monitoring and control, such as the generation and logging of platform level events, are also defined.

Platform Level Data Model (PLDM) for Platform Monitoring and Control

1 Scope

This specification defines the functions and data structures used for discovering, describing, initializing, and accessing sensors and effecters within the management controllers and management devices of a platform management subsystem using PLDM messaging. Additional functions related to platform monitoring and control, such as the generation and logging of platform level events, are also defined. This document does not specify the operation of PLDM messaging.

This specification is not a system-level requirements document. The mandatory requirements stated in this specification apply when a particular capability is implemented through PLDM messaging in a manner that is conformant with this specification. This specification does not specify whether a given system is required to implement that capability. For example, this specification does not specify whether a given system must provide sensors or effecters. However, if a system does implement sensors or effecters or other functions described in this specification, the specification defines the requirements to access and use those functions under PLDM.

Portions of this specification rely on information and definitions from other specifications, which are identified in section 2. Two of these references are particularly relevant:

- DMTF DSP0240, *Platform Level Data Model (PLDM) Base Specification*, provides definitions of common terminology, conventions, and notations used across the different PLDM specifications as well as the general operation of the PLDM messaging protocol and message format.
- DMTF DSP0249, *Platform Level Data Model (PLDM) State Sets Specification*, defines the values that are used to represent different types of states and entities within this specification.

2 Normative References

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

2.1 Approved References

DMTF DSP4004, *DMTF Release Process 1.6*

DMTF DSP0236, *MCTP Base Specification 1.0 Preliminary*

DMTF DSP0240, *Platform Level Data Model (PLDM) Base Specification*

DMTF DSP0241, *Platform Level Data Model (PLDM) Over MCTP Binding Specification*

DMTF DSP0245, *Platform Level Data Model (PLDM) IDs and Codes Specification*

2.2 References under Development

DMTF DSP0249, *Platform Level Data Model (PLDM) State Sets Specification*

2.3 Other References

IETF, RFC2781, *UTF-16, an encoding of ISO 10646*, February 2000

IETF, RFC3629, *UTF-8, a transformation format of ISO 10646*, November 2003

IETF, RFC4122, *A Universally Unique Identifier (UUID) URN Namespace*, July 2005

IETF, RFC4646, *Tags for Identifying Languages*, September 2006

ISO 8859-1, *Final Text of DIS 8859-1, 8-bit single-byte coded graphic character sets -- Part 1: Latin alphabet No.1*, February 1998

ANSI/IEEE Standard 754-1985, *Standard for Binary Floating Point Arithmetic*

3 Terms and Definitions

Refer to DSP0240 for terms and definitions that are used across the PLDM specifications. For the purposes of this document, the following additional terms and definitions apply.

3.1

contained entity

an entity that is contained within a container entity

3.2

container entity

an entity that is identified as containing or comprising one or more other entities

3.3

container ID

a numeric value that is used within Platform Descriptor Records (PDRs) to uniquely identify a container entity

3.4

containing entity

an alternative way of referring to the container entity for a given entity

3.5

entity

a particular physical or logical entity that is identified using PLDM monitoring and control data structures for the purpose of monitoring, controlling, or identifying that entity within the platform management subsystem, or for identifying the relationship of that entity to other entities that are monitored or controlled using PLDM monitoring and control

Examples of physical entities include processors, fans, power supplies, and memory chips. Examples of logical entities include a logical power supply (which may comprise multiple physical power supplies) and a logical cooling unit (which may comprise multiple fans or cooling devices).

3.6

Entity ID

a numeric value that is used to identify a particular type of entity, but without designating whether that entity is a physical or logical entity

3.7

Entity Instance Number

a numeric value that is used to differentiate among instances of the same type

For example, if two processor entities exist, one of them can be designated with instance number 1 and the other with instance number 2.

3.8

Entity Type

a numeric value that identifies both the particular type of entity and whether the entity is a physical or logical entity

The Entity ID is a sub-field of the Entity Type.

3.9

Platform Descriptor Record

PDR

A set of data that is used to provide semantic information about sensors, effecters, monitored or controller entities, and functions and services within a PLDM implementation

PDRs are mostly used to support PLDM monitoring and control and platform events. This information also describes the relationships (associations) between sensor and control functions, the physical or logical entities that are being monitored or controlled, and the semantic information associated with those elements.

4 Symbols and Abbreviated Terms

Refer to DSP0240 for symbols and abbreviated terms that are used across the PLDM specifications. For the purposes of this document, the following additional symbols and abbreviated terms apply.

4.1

CIM

Common Information Model

4.2

EID

Endpoint ID

4.3

IANA

Internet Assigned Numbers Authority

4.4

MAP

Manageability Access Point

4.5

MCTP

Management Component Transport Protocol

4.6

PDR

Platform Descriptor Record

4.7

PLDM

Platform Level Data Model

4.8

TID

Terminus ID

5 Conventions

Refer to DSP0240 for conventions, notations, and data types that are used across the PLDM specifications. The following data types are also defined for use in this specification:

Table 1 – PLDM Monitoring and Control Data Types

Data Type	Interpretation
strASCII	A null (0x00) terminated 8-bit per character string. Unless otherwise specified, characters are encoded using the 8-bit ISO8859-1 "ASCII + Latin1" character set encoding. All strASCII strings shall have a single null (0x00) character as the last character in the string. Unless otherwise specified, strASCII strings are limited to a maximum of 256 bytes including null terminator.
strUTF-8	A null (0x00) terminated, UTF-8 encoded string per RFC3629. UTF-8 defines a variable length for Unicode encoded characters where each individual character may require one to four bytes. All strUTF-8 strings shall have a single null character as the last character in the string with encoding of the null character per RFC3629 Unless otherwise specified, strUTF-8 strings are limited to a maximum of 256 bytes including null terminator character.
strUTF-16	A null (0x0000) terminated, UTF-16 encoded string with Byte Order Mark (BOM) per RFC2781. All strUTF-16 strings shall have a single null (0x0000) character as the last character in the string. An empty string shall be represented using two bytes set to 0x0000, representing a single null (0x0000) character. Otherwise, the first two bytes shall be the BOM. Unless otherwise specified, strUTF-16 strings are limited to a maximum of 256 bytes including the BOM and null terminator.
strUTF-16LE	A null (0x0000) terminated, UTF-16, "little endian" encoded string per RFC2781. All strUTF-16LE strings shall have a single null (0x0000) character as the last character in the string. Unless otherwise specified, strUTF16LE strings are limited to a maximum of 256 bytes including the null terminator.
strUTF-16BE	A null (0x0000) terminated, UTF-16, "big-endian" encoded string per RFC2781. All strUTF-16BE strings shall have a single null character as the last character in the string. Unless otherwise specified, strUTF16BE strings are limited to a maximum of 256 bytes including the null terminator.

6 PLDM for Platform Monitoring and Control Version

The version of this *Platform Level Data Model (PLDM) for Platform Monitoring and Control Specification* shall be 1.0.0 (major version number 1, minor version number 0, update version number 0, and no alpha version).

For the GetPLDMVersion command described in DSP0240, the version of this specification is reported using the encoding as 0xF1F0F000.

7 PLDM for Platform Monitoring and Control Overview

This specification describes the operation and format of request messages (also referred to as commands) and response messages for accessing the monitoring and control functions within the management controllers and management devices of a platform management subsystem. These messages are designed to be delivered using PLDM messaging.

The basic format that is used for sending PLDM messages is defined in DSP0240. The format that is used for carrying PLDM messages over a particular transport or medium is given in companion documents to the base specification. For example, DSP0241 defines how PLDM messages are formatted and sent using MCTP as the transport. The *Platform Level Data Model (PLDM) for Platform Monitoring and Control Specification* defines messages that support the following items:

- sensors and effecters

  This specification defines a model for sensors and effecters through which monitoring and control are achieved, and the commands that are used for sensor and effecter initialization, configuration, and access. Sensors and effecters are classified according to the general type of data that they use:

  – Numeric sensors provide a number that is representative of a monitored value that can be expressed using units such as degrees Celsius, volts, and amps.

  – State sensors are used for accessing a number from an enumeration that represents the state of a monitored entity. Different states are enumerated in predefined sets called state sets. Example state sets can include states for Availability (enabled, disabled, shut down, and so on), Door State (open, closed), Presence (present, not present) and so on. The values for State Sets are defined in DSP0249.

  – Numeric effecters are used for setting a number that configures or controls the operation of a controlled entity. Like numeric sensors, numeric effecters also use units such as degrees Celsius, volts, and amps.

  – State effecters are used for setting a number that configures or controls a state that is associated with a controlled entity. State effecters draw upon the same state set definitions as state sensors.

- Platform Descriptor Records (PDRs)

  PDRs are data structures that can provide semantic information for sensors and effecters, their relationship to the entities that are being monitored or controlled, and associations that exist between entities within the platform. The PDRs also include information that describes the presence and location of different PLDM termini. This information can be used to discover the population of sensors and effecters and how to access them by using PLDM messaging. The information also facilitates building Common Information Model objects and associations for the sensors, effecters, and platform entities. PDRs can also hold information that is used to initialize sensors and effecters. PDRs are collected into a logical storage area called a PDR Repository. A central PDR Repository called the Primary PDR Repository can be used to hold an aggregation of all PDR information within the PLDM subsystem.

- platform events

  This specification defines messages that are asynchronously sent upon particular state changes that occur within sensors, effecters, or the PLDM platform management subsystem. The messages are delivered to a central function called the PLDM Event Receiver.

- platform event logging

  The specification includes the definition of a central, non-volatile storage function called the PLDM Event Log that can be used to log PLDM Event Messages. The specification also defines messages for accessing and maintaining the PLDM Event Log.

- support functions

  This specification also includes the definition of support functions as required to support the initialization of sensors and effecters, and the maintenance of PDRs in the Primary PDR Repository. The main support functions are the Discovery Agent and the Initialization Agent.

  – The Discovery Agent function is responsible for keeping the Primary PDR information up to date if entities are added, relocated, or removed from the PLDM platform management subsystem. The Discovery Agent function is also responsible for setting the Event Receiver location into PLDM termini that support PLDM monitoring and control messages.

  – The Initialization Agent function is responsible for initializing sensors and effecters that may require initialization or re-initialization upon state changes to the PLDM terminus or the managed system, such as system hard resets, the terminus coming online for PLDM communication, and so on.

- OEM/vendor-specific functions

  This specification includes provisions for supporting OEM or vendor-specific functions and semantic information. This includes the ability to define OEM units for numeric sensors or effecters, OEM state sets, and OEM entity types. An OEM PDR type is also available as an opaque storage mechanism for holding OEM-defined data in PDR Repositories.

8 PDR Architecture

This section provides an overview of when and how PDRs are used within a platform management subsystem that uses the PLDM Platform Monitoring and Control commands.

8.1 General

PLDM generally separates the access of functions such as sensors and effecters from the semantic information or description of those functions. For example, PLDM commands such as GetNumericSensorReading return binary values for a sensor, but the meaning of those values, such as whether they represent a temperature or voltage, is described separately. The description or semantic information for sensors, effecters, and other elements of the PLDM platform management subsystem is provided through Platform Descriptor Records, or PDRs.

This separation provides several benefits:

- Overhead for simple Intelligent Management Devices is reduced. In many implementations, a primary management controller may access one or two simpler controllers that act as Intelligent Management Devices (sometimes also called "satellite controllers"). Those controllers generally are very cost sensitive and limited in resources such as RAM, non-volatile storage capabilities, data transfer performance, and so on. The amount of data that needs to be stored and transferred to provide the semantic information for a sensor is typically an order of magnitude or more greater than the amount of data that needs to be transferred to get the state or reading information from a sensor.
- PDRs provide information that associates sensors, effecters, and the entities that are being monitored or controlled within the overall context of the PLDM platform management subsystem. This eliminates the need for devices that implement sensors and effecters to understand their position and use in the overall system. Providing this association and context information for sensors and effecters enables the automatic instantiation of CIM objects and CIM associations.
- The impact of extensions to descriptions is reduced. The definitions of the semantic information (PDRs) can be extended and modified without affecting the commands that are used to access sensors and effecters.

8.2 Primary PDR Repository and Device PDR Repositories

The PDRs for a PLDM subsystem are collected into a single, central PDR Repository called the Primary PDR Repository. A central repository provides a single place from which PDR information can be retrieved and simplifies the inter-association of PDR semantic information for the different elements and monitored or controlled entities within the subsystem.

Individual devices, such as hot-plug devices, can hold their own Device PDRs that describe their local semantics. Typically, this information has only local context. That is, the information covers only the elements on the add-in card and has no information about the positioning of the card and its capabilities relative to the overall subsystem. Thus, additional steps are typically taken to integrate Device PDR information into the overall context of the PLDM subsystem.

8.3 Use of PDRs

Whether PDRs are used is based on the needs and goals of the PLDM subsystem implementation. This section describes three different applications of PLDM and their level of PDR support.

8.3.1 PLDM for Access Only

Figure 1 shows an implementation that does not use PDRs. PLDM is used only as a mechanism for accessing monitoring and control functions; it is not used for providing semantic information about those functions.

In this example, Device A provides a DMTF Manageability Access Point (MAP) function that makes platform information available over a network using CIM as the data model and WS-MAN as the transport protocol for CIM. In this example, PLDM is used only for accessing the functions in Devices B and C, and for Devices B and C to send PLDM Event Messages to Device A.

All the semantic or descriptive information that is needed to map the sensors and effecters to CIM objects and properties is handled by proprietary mechanisms. Typically a vendor-specific configuration utility is used by the system integrator to configure or customize a set of proprietary configuration information that provides whatever contextual or semantic information is required for the particular platform implementation. Since the mechanisms for recording semantic information are proprietary, most of the PLDM-to-CIM mapping function is also proprietary. A standard approach for the PLDM-to-CIM mapping function cannot be specified when proprietary mechanisms are used for the semantic information.

Thus, in this example PLDM does not offer much to assist or direct the way sensor and effecter functions of external management devices would be mapped into the instantiation of CIM objects. The implementation only uses PLDM to provide a common mechanism for accessing the functions in the external Intelligent Management Devices. This enables the implementation to be designed with "Device Driver" and PLDM Event Handling code that can be reused if it is necessary to change the design to support different external Intelligent Management Devices.

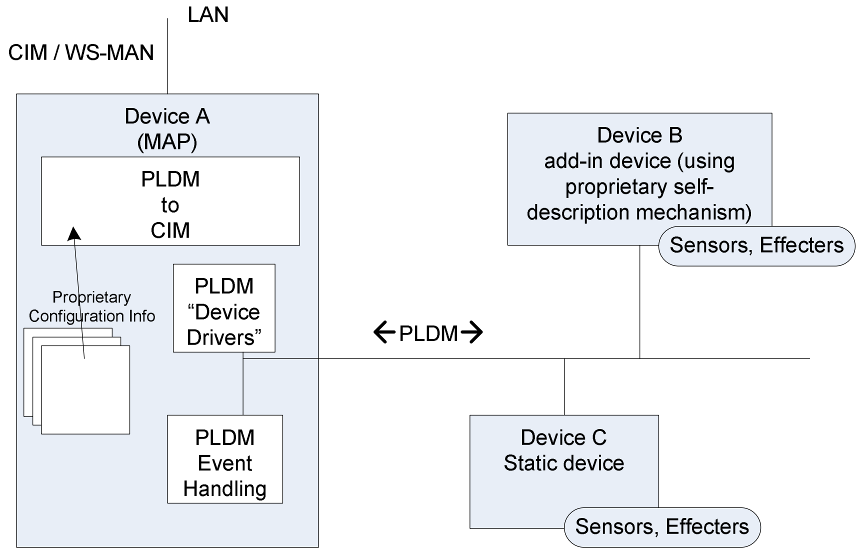


Figure 1 – PLDM Used for Access Only

8.3.2 PLDM with PDRs for Add-in Devices

Figure 2 illustrates how PDRs can be used with add-in cards. The vendor of an add-in card knows the relationships and semantics of the monitoring and control (sensor and effecter) capabilities on their card. However, the vendor of the card typically will not know the relationship that card will have relative to a particular overall system. For example, the vendor would not know a-priori what the system name was, or how many processors the system has, or which slot the card will be plugged into. Thus, in this example, the add-in card exports PDRs that describe the relationships relative to the add-in card. The MAP takes this information and integrates it into the semantic view of the overall system. The PDR information could be converted and linked into a proprietary internal database, as shown in Figure 2. The PDRs thus provide a common way for add-in cards to describe themselves to the MAP.

The internal database for the MAP could be implemented as a PDR Repository instead of a proprietary database. This would potentially simplify the PLDM-to-CIM mapping process, enabling the integrated data to be accessed as PDRs using PDR Repository access commands and enabling software or other parties to see the integrated view of the platform at the PLDM level. Also, because the PLDM-to-CIM mapping is defined using PDRs, the PDR format may also be useful in developing a consistent PLDM-to-CIM mapping in the MAP.

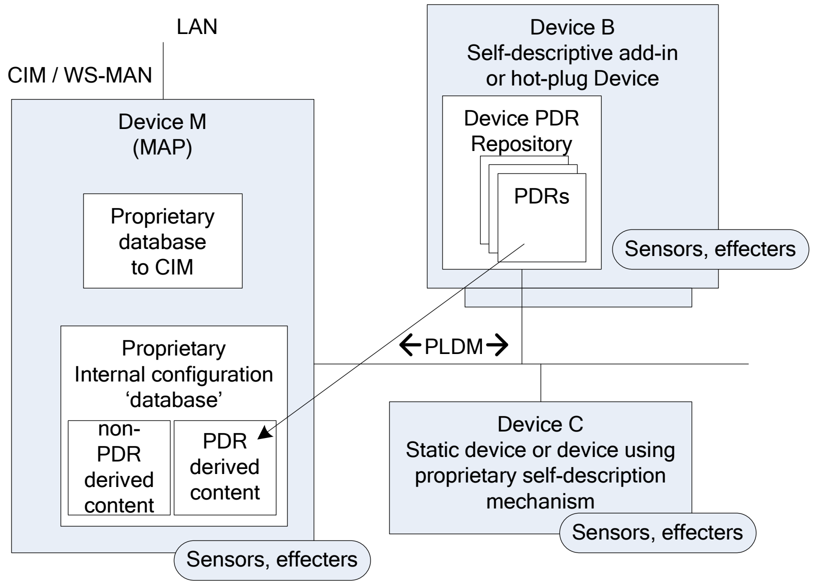


Figure 2 – PLDM with Device PDRs

8.3.3 PLDM with Primary PDR Repository

Figure 3 shows an example of using PDRs to describe an entire PLDM platform management subsystem to an add-in card, Device M, that provides a MAP function. In this example, PDRs are collected into a central PDR Repository called the Primary PDR Repository that is provided by Device A.

The PDRs in the Primary PDR Repository represent the entire PLDM subsystem behind Device A. Thus, the MAP of Device M needs to connect only to Device A to discover and get semantic information about the monitoring and control functions for that entire subsystem. This approach can enable Device M to automatically adapt itself to the management capabilities offered by different systems.

Such an implementation enables the MAP to come from one party while the platform management subsystem comes from another without the need to explicitly configure the MAP with the semantic information for the subsystem. For example, the platform management subsystem represented through Device A could be built into a motherboard and the MAP of Device M provided on a PCIe add-in card from a third party. The MAP on the add-in card can use the Primary PDR Repository to automatically discover the capabilities and semantic information of the platform management subsystem and use that information to instantiate CIM objects and data structures for the subsystem.

Device A maintains the Primary PDR Repository that includes information about static sensors and effecters (such as those within Device C and within Device A itself) and integrates that information into the overall view of the platform management subsystem held in the Primary PDR Repository. This involves discovering and extracting PDRs from "Self-descriptive" devices such as Device B, and synthesizing additional PDRs, such as association and Terminus Locator PDRs, in order to integrate the PDRs into the repository and create a coherent view of the overall subsystem.

Because Device M is an add-in card, it could also have its own sensors and effecters and associated PDRs that Device A would integrate into the Primary PDR Repository in the same manner that it integrates PDR information from Device B.

Another advantage of implementing a Primary PDR Repository is that any party with access to Device A can get the full set of semantic information for the subsystem. This is useful when more than one party might need to access that information—for example, if support was necessary for multiple add-in cards that provided MAP functions for different media (such as one card that provided MAP functions over cabled Ethernet and another that provided MAP access using a wireless network connection).

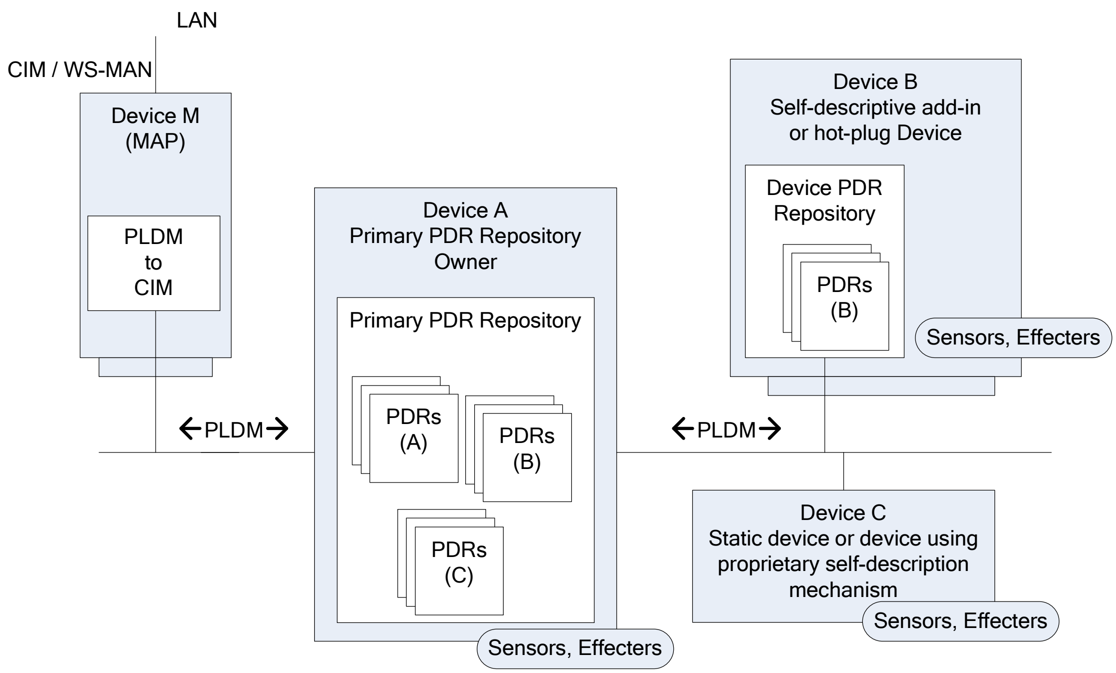


Figure 3 – PLDM with PDRs for Subsystem

9 Entities

Within the context of this specification, the term entity is used either to refer to a physical or logical entity that is monitored or controlled, or to describe the topology or structure of the system that is being monitored or controlled.

Examples of typical physical entities include processors, fans, memory devices, and power supplies. Examples of logical entities include logical power supplies that are formed from multiple physical power supplies (as in the case of a redundant power supply subsystem) and a logical cooling unit formed from multiple physical fans.

9.1 Entity Identification Information

Individual entities are identified within PLDM PDRs using three fields: Entity Type, Entity Instance Number, and Container ID. Together, these fields are referred to as the Entity Identification Information. Figure 4 presents an overview of the meaning of the individual fields. The fields are discussed in more detail in the next sections.

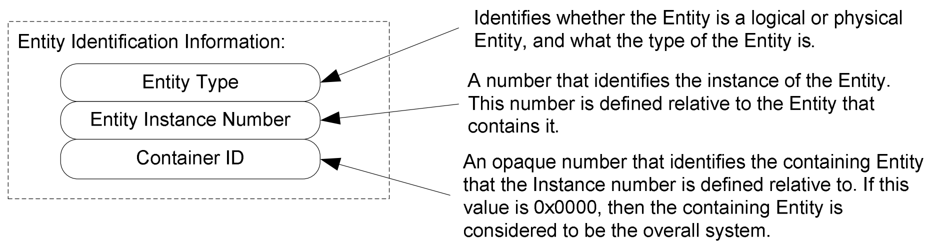


Figure 4 – Entity Identification Information

The combination of Entity Type, Entity Instance Number, and Container ID must be unique for each individual entity referenced in the PDRs. These three fields are always used together in the PDRs and in the same order. The combination of the three fields is represented in the PDRs using three uint16 values in the format shown in Figure 5.

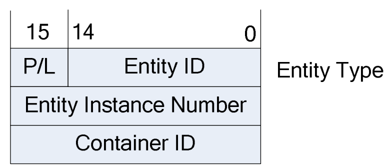


Figure 5 – Entity Identification Information Format

Table 2 describes the parts of the Entity Identification Information format.

Table 2 – Parts of the Entity Identification Information Format

Part	Description
Entity Type	Combination of the P/L bit and the Entity ID value
P/L	Physical/Logical bit (0b = physical, 1b = logical)
Entity ID	15-bit Entity ID value from DSP0249 that identifies the general type of the entity
Entity Instance Number	16-bit number that differentiates among instances of entities that have the same Entity Type and Container ID values
Container ID	A 16-bit number that identifies the containing entity that the Entity Instance Number is defined relative to. If this value is 0x0000, the containing entity is considered to be the overall system.

9.2 Entity Type and Entity IDs

The Entity Type field is a concatenation of the physical/logical designation for the entity and the value from the Entity ID enumeration that identifies the general type or category of the entity, such as whether the entity is a power supply, fan, processor, and so on. The Entity Type field indicates whether the entity is a physical fan, logical power supply, and so on.

The different general types of entities within PLDM are identified using an enumeration value referred to as an "Entity ID." The different types of standardized entities and their corresponding Entity ID values are specified in DSP0249.

Physical and logical entities that have the same Entity ID are considered to be different Entity Types.

9.2.1 Vendor-Specific (OEM) Entity IDs

The Entity ID values include a special range of values for identifying vendor- or OEM-specific entities. In order to be interpreted, these values must be accompanied by a OEM EntityID PDR that identifies which vendor defined the entity and, optionally, a string or strings that provide the name for the entity. Refer to 28.19 for additional information about how OEM Entity IDs are used.

9.2.2 Logical and Physical Entities

A physical entity is defined as an entity that is formed of one or more physically identifiable components. For example, a physical Power Supply could be one or more integrated circuits and associated components that together form a power supply.

A logical entity is defined as an entity that is formed when the entity or grouping of entities lacks a physical definition or a readily identifiable physical boundary or grouping that would be associated with the type of entity being represented. For example, a logical cooling device could be used to represent a combination of physical fans that forms a redundant fan subsystem, or a logical power supply could be used to represent the combination or grouping of power supplies that forms a redundant power supply subsystem.

The choice of when to use a logical or physical designation for a particular type of entity can be subtle. Consider the following questions:

- Is the entity or grouping of entities separately replaceable or identifiable as a single physical unit or as a set of physical units?
- Would the physical grouping be something that a user would typically think of as a separate physical unit that can be represented by a single type of entity?

For example, consider a system with a motherboard that directly supports connectors for a redundant fan configuration. The fans would typically be individually replaceable, and the motherboard would be individually replaceable, but the "redundant fan subsystem" would not be. A user would not typically consider the combination of a motherboard and fans to be the definition of a physical redundant fan subsystem because the motherboard provides many other functions beyond those that are part of the implementation of a redundant fan subsystem. The redundant fan subsystem does not have a distinct physical boundary that would let it be replaced independently from other subsystems.

9.3 Entity Instance Numbers

A given platform often has more than one occurrence of a particular type of entity. The Entity Instance Number, in combination with the Container ID, differentiates one instance of a particular type of entity from another within the PDRs.

Entity Instance Numbers are defined in a numeric space that is associated with a particular containing entity. For example, the Entity Instance Numbers for processors contained on an add-in card are defined relative to that add-in card, whereas the Entity Instance Numbers for processors on the motherboard are defined relative to the motherboard.

The Entity Instance Number is a value that could be used when instantiating CIM objects or presenting PLDM data as part of the "name" of the managed object. For example, if a processor entity has an Entity Instance Number of "1", the expectation is that the entity would be presented as “Processor 1”.

The assignment of Entity Instance Number values under a given Container ID is left up to the implementation. However, it is typical that Entity Instance Number values are allocated sequentially starting from 0 or 1 for a given Entity Type under the Container ID.

9.4 Container ID

The value in this field identifies a "containing Entity" that in turn defines the numeric space under which Entity Instance Numbers are allocated. For example, if an add-in card has two processors on it and a motherboard has two processors on it, it would be common to refer to the processors on the add-in card as "Processor 1" and "Processor 2" and to the processors on the motherboard also as "Processor 1" and "Processor 2".

The Container ID field provides a mechanism that locates a particular containing entity, such as "motherboard 1" or "add-in card 1". This enables the Entity Instance Numbers to be allocated relative to each particular containing Entity. The Container ID field, therefore, effectively provides a value that indicates that the "Processor 1" entity on the motherboard is a different entity than the "Processor 1" entity on the add-in card.

In most cases, the Container ID field value points to a particular PDR that describes a "containment association" that identifies a container entity (such as motherboard 1) and one or more contained entities (such as processor 1 and processor 2). An exception occurs when an entity instance is defined only relative to the overall system, in which case the Container ID holds a special value that indicates that the "system" is the container entity.

9.5 Use of Container ID in PDRs

With the exception of the entity that represents an overall system, all entities are contained within at least one other physical or logical entity. Each entity is thus part of a containment hierarchy that starts with the overall system as the topmost entity. A strict hierarchy is formed when each entity is only allowed to identify a single containing entity using the Container ID value. With this restriction, an entity's position in the hierarchy can be uniquely identified, and when combined with the entity type and instance information provides the unique Entity Identification Information for the entity. Thus, although a given entity may be identified as being contained within more than one container entity, only one Container ID value shall be used for the Entity Identification Information for an entity.

The Container ID points to a particular type of PDR called an Entity Association PDR that holds the information that identifies and associates a containing entity with one or more contained entities. Association PDRs are described in clause 10.

The overall system is considered to be the top of the hierarchy of containment and thus does not appear as a contained entity in any Entity Association PDR. In this case, there is no explicit Entity Association PDR for the overall system. A special value (0x0000) is used for the Container ID to indicate when the overall system is the container entity.

In some cases, a particular entity may be part of more than one containment hierarchy. For example, a physical fan could be part of a logical cooling unit *and* a physical chassis. When both physical and logical containers exist for a given entity, the physical container relationship should be used for identifying the entity.

10 PLDM Associations

Different mechanisms are used to associate different elements of PLDM with one another. This section describes the different association mechanisms and how they're used.

10.1 Association Examples

Following are some examples of associations that are covered by PDRs:

- Sensor/Effecter Semantic Information to Sensor/Effecter Access associations:
  Sensor and effecter PDRs describe the characteristics of a particular sensor or effecter. These records include information that can be used to identify which PLDM terminus provides the interface to the sensor, and the parameters that are used to access that sensor. These records provide a way to form an association between the semantic information for a sensor/effecter (provided by other information in the PDRs) and the access of the sensor (provided by PLDM commands for sensor or effecter access).
- Sensor/Effecter to Entity associations:
  A sensor or effecter monitors or controls some physical or logical entity. The PDRs provide a mechanism for associating a sensor or effecter with the entity.
- Entity to Entity associations:
  Entities have relationships with other entities, such as physical and logical containment. For example, a redundant power supply subsystem may be represented as a logical power supply that is made up of multiple physical power supplies.
- PLDM Event to PDR associations:
  PLDM Event Messages identify the terminus that was the source of the message, and the sensor within the terminus that was the source of the event, but semantic information and the context for the sensor are not carried in the event information. The PDRs include information that associates the information in an event message with the semantic information that enables interpretation of the event and its context.

Two general mechanisms are used for specifying associations for PLDM: Internal Associations and External Associations.

10.2 Internal and External Associations

The term "Internal Association" is used when a particular type of association is formed solely by using fields within the PDRs that directly associate PDRs with one another. For example, a value called the Terminus Handle is used in all PDRs that are associated with a particular terminus. The Terminus Handle is a form of Internal Association, where the association is "PDRs that belong to a given terminus." Internal Associations effectively associate records by defining and using a common field as a key.

Therefore, Internal Associations require a common field to be defined among the elements that are associated with each other. The Internal Association mechanism is efficient, but not readily extensible, because a new type of association would typically require new fields to be defined and added to the PDRs that are to be associated with one another, along with specifications that document how the field is used to form links to other records. Because the fields that support Internal Associations must be pre-defined as part of the PDR, internal associations are generally used only for the most fundamental and common types of associations. For other types of associations, a more generalized mechanism called "External Associations" is provided.

External Associations are formed by using a separate data structure (PDR) to associate different elements with one another. This is accomplished among the PDRs by using another PDR that is referred to as an “association PDR.” The advantage of using External Associations is that they enable associations between PDRs or entities without requiring the definition of common fields among them. Thus, new types of associations can be defined without requiring changes to existing PDR definitions. The disadvantage is that External Associations require the use of at least one additional PDR to form the association.

10.3 Sensor/Effecter to Entity Associations

Each sensor or effecter that is described using PDRs has a corresponding Sensor or Effecter PDR that provides semantic information for individual sensors or effecters, such as information that identifies which terminus the sensor or effecter is associated with, the type of parameter that the sensor or effecter is monitoring or controlling, and so on. Included in this information is Entity Identification Information for the entity that is associated with the sensor or effecter. (The terms Sensor PDRs and Effecter PDRs are used as shorthand to refer to a general class of PDRs. The actual PDRs define separate PDRs for numeric sensors, state sensors, numeric effecters, state effecters, and so on.)

Figure 6 shows a subset of the fields in the Sensor PDR for a PLDM Numeric Sensor. The Entity Identification Information is represented by the fields highlighted with dashed lines. Note that from this point in the document onward figures and tables will use field names as they are given in the definition of the PDRs, for example "entityInstanceNumber" instead of "entity instance number".

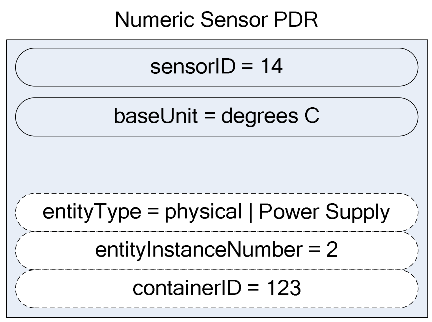


Figure 6 – Entity Identification Information in a Sensor PDR

Table 3 describes the meaning of the fields shown in Figure 6.

Table 3 – Field and Value Descriptions for Entity Identification Information in a Sensor PDR

Field and Value	Description
sensorID = 14	All sensors and effecters within a given terminus have unique sensorID or effecterID numbers. This field holds a value that is used in commands such as GetSensorReading to access the particular sensor or effecter within the terminus. The sensorID number is used only for accessing the sensor. The example shows that the value 14 would be used in commands to access this particular sensor.
baseUnit = degrees C	The baseUnit field identifies the measurement unit for the parameter being monitored by the sensor. The measurement unit is simplified for this example. The actual PDR contains additional fields that contribute to the definition of the measurement unit for a numeric sensor. Refer to the field's description in Table 66 for more information.
entityType = physical | Power Supply	This field represents the concatenation of the physical/logical bit and the Entity ID for "power supply" from the Entity IDs table (see 9.2).
entityInstanceNumber = 2	The entityInstanceNumber differentiates instances of entities that have the same Entity Type and Container ID values. Because the entityInstanceNumber is defined relative to a containing entity, a system can have a processor on the motherboard identified as "processor 1" and a processor on an add-on card also identified as "processor 1". The two occurrences of "processor 1" are recognized as being unique and separate entities because they have different container entities. In this example, the entityInstanceNumber 2 indicates that

Field and Value	Description
	this numeric sensor is monitoring physical Power Supply 2, which is contained within the container entity identified by containerID 123.
containerID = 123	This field is used to identify or locate the containing entity that defines the numeric space for the entityInstanceNumber. In this example, the number 123 would be used to locate an Entity Association PDR that identifies the containing entity (see 9.4 for more information). Association PDRs are described in detail in section 11.

The details included in Table 3 provide a significant amount of the information that is typically used for identifying a sensor or effecter and its use within a management subsystem. For example, a string that contains the following identification information for the sensor could be derived from the Numeric Sensor PDR without referring to any additional PDRs:

"Entity(123) physical power supply 2 degrees C 1"

The information is based on the following fields:

container ID | entityType | entityInstanceNumber | baseUnit | sensorInstanceNumber

Note that an application would typically not use just the baseUnits name "degrees C" but would augment it to make it more readable. For example:

"Entity(123) physical power supply 2 Temperature 1 (Celsius)"

To interpret Entity(123), it is necessary to interpret the Container ID. If the Container ID is for "system," the PDR may be interpreted as follows:

"System Physical Power Supply 2 Temperature 1 (Celsius)"

If the Container ID is for an entity other than system, the Container ID information can be used to locate the Entity Association PDR that identifies the containing entity for the sensor.

11 Entity Association PDRs

Entity Association PDRs associate entities with one another.

11.1 Physical to Physical Containment Associations

One of the most common associations is the "physical containment association." This association is used to indicate that a physical entity contains one or more other physical entities. For example, the association can be used to represent that a physical chassis contains multiple power supplies. Figure 7 shows an example of selected fields within an Entity Association PDR that describes a physical containment association.

The example shows a containerID field and an associationType field in the PDR. The containerID is tied to the identification information for the container entity, which in this example is "system physical chassis 1." The associationType field indicates that the association is a physical-to-physical containment association.

The record has entries for two contained power supplies, physical Power Supply 1 and physical Power Supply 2. The Entity Identification Information for both supplies refers back to the containerID 123 for the container entity, system physical chassis 1. Although this may appear redundant, it is done so that Entity Identification Information within PDRs is consistently represented with the same three-field format, and because in some types of associations the contained entity references the ID for a container entity that is identified in a different PDR.

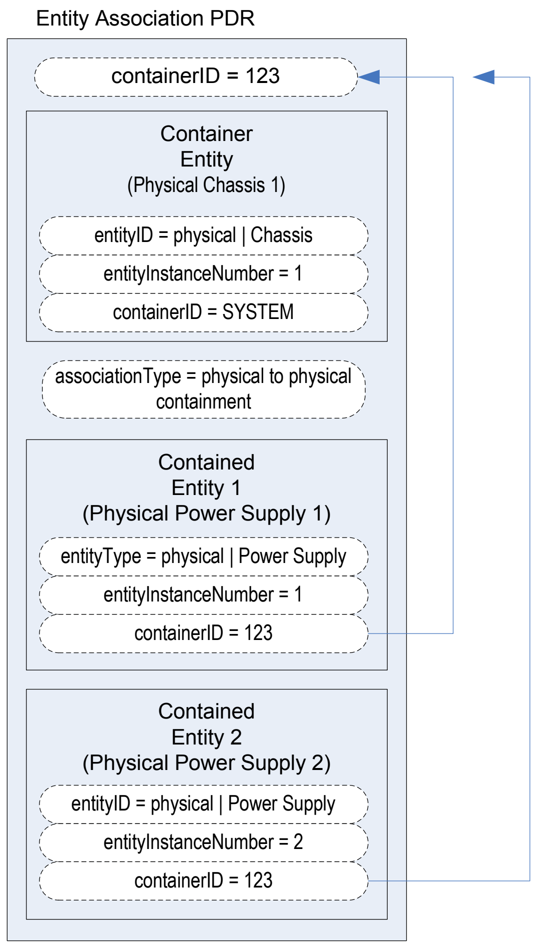


Figure 7 – Physical Containment Entity Association PDR

Although the definition and use of the first containerID field might be confusing at first, think of the value as a single, unique number that identifies a container entity within the PLDM PDRs. The value thus represents the combination of the EntityType, entityInstanceNumber, and containerID values for the container entity. For example, referring to Figure 7, containerID 123 represents physical Chassis 1 (where instance number 1 is defined relative to SYSTEM).

Figure 8 provides an illustration of how the containerID value links entities in a containment hierarchy.

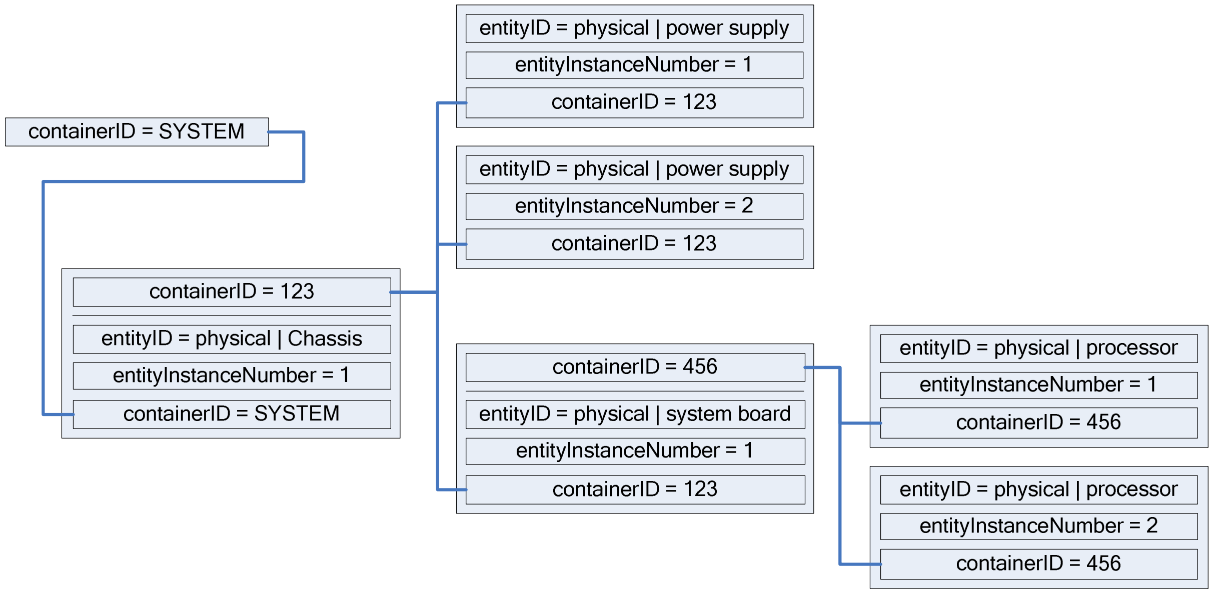


Figure 8 – containerID Relationships

11.2 Entity Identification Relationships between PDRs

Figure 9 shows the kinds of association relationships that emerge when the PDRs are used in combination. The Numeric Sensor PDR in this example has Entity Identification Information that corresponds to "Power Supply 2." The containerID information in that Numeric Sensor PDR corresponds to the containerID that is linked to Physical Chassis 1 through the Entity Association PDR. Note that Physical Chassis 1 is identified as being contained only by the overall system. Hence, its containerID is SYSTEM.

Putting this information together yields a view of the system that is represented by the block diagram shown in Figure 9, which shows that the system contains a physical chassis that in turn contains two physical power supplies, and that each physical power supply has a temperature sensor associated with it. The two temperature sensors are both referred to as "Temperature 1" because their sensorInstanceNumber is defined relative to the power supply that is being monitored.

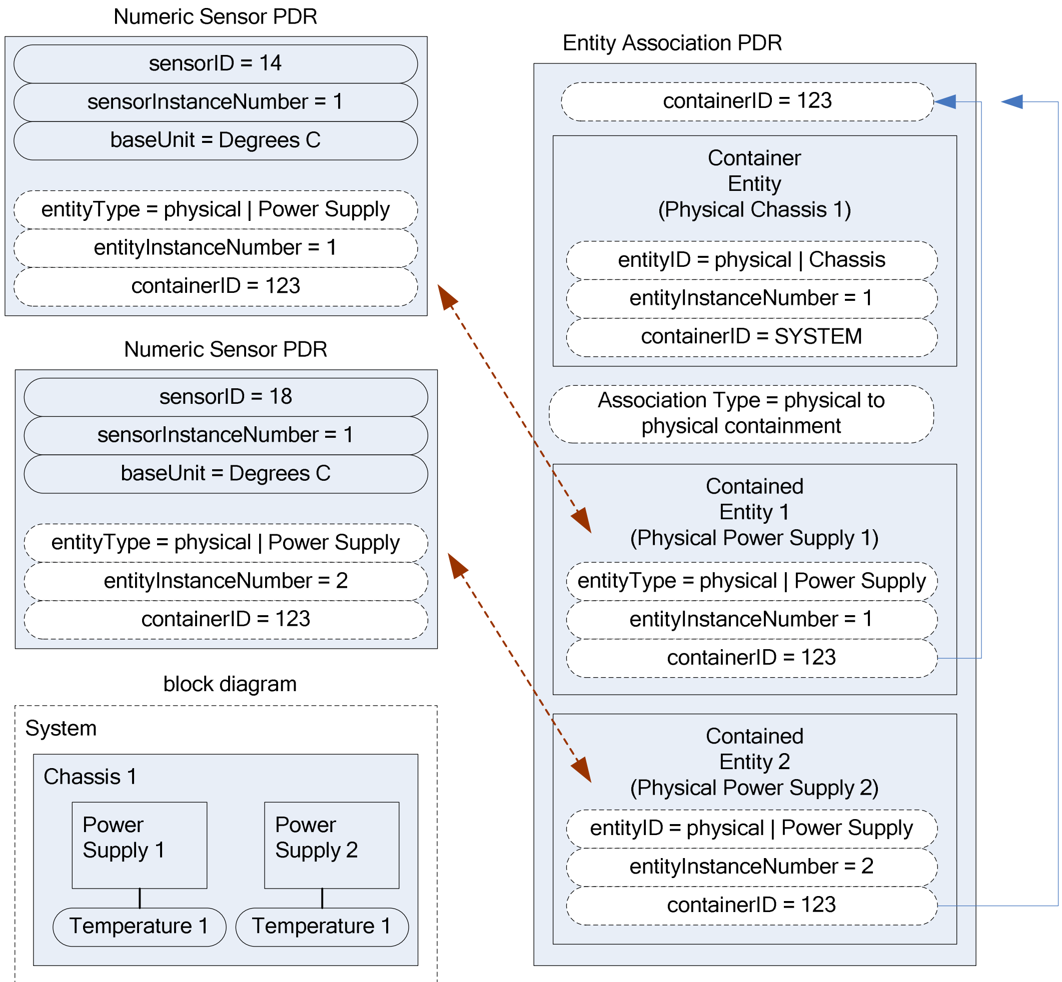


Figure 9 – Entity Identification Relationship between PDRs

The Entity Identification Information can thus be used for different types of associations within the PDRs. In this example, it is used in the Numeric Sensor PDR to identify the monitored entity in a sensor-to-entity association, and it is used within an Entity Association PDR to identify a containment association between the power supplies and the chassis.

11.3 Linked Entity Association PDRs

Certain types of PDRs can be linked together using an Internal Association to form the equivalent of a single joint PDR. In Figure 10, the two Entity Association PDRs on the right are implicitly linked together by sharing the same containerID value. (Note that in Figure 10, the linked PDRs are also required to have the same container entity information and associationType values.)

The two PDRs on the right and the large single PDR on the left represent exactly the same association relationship: the container entity "physical chassis 1" contains two physical power supplies, "power supply 1" and "power supply 2", and two physical fans, "fan 1" and "fan 2".

It is a choice of the implementation whether a single PDR or multiple PDRs are used to represent a containment association. Some implementations might want to use multiple records to make it easier to develop and maintain the records. For example, if a new physical entity is added for the chassis, it might be more convenient to create a new PDR and link it into the existing containment PDRs for a chassis rather than extending an existing containment PDR.

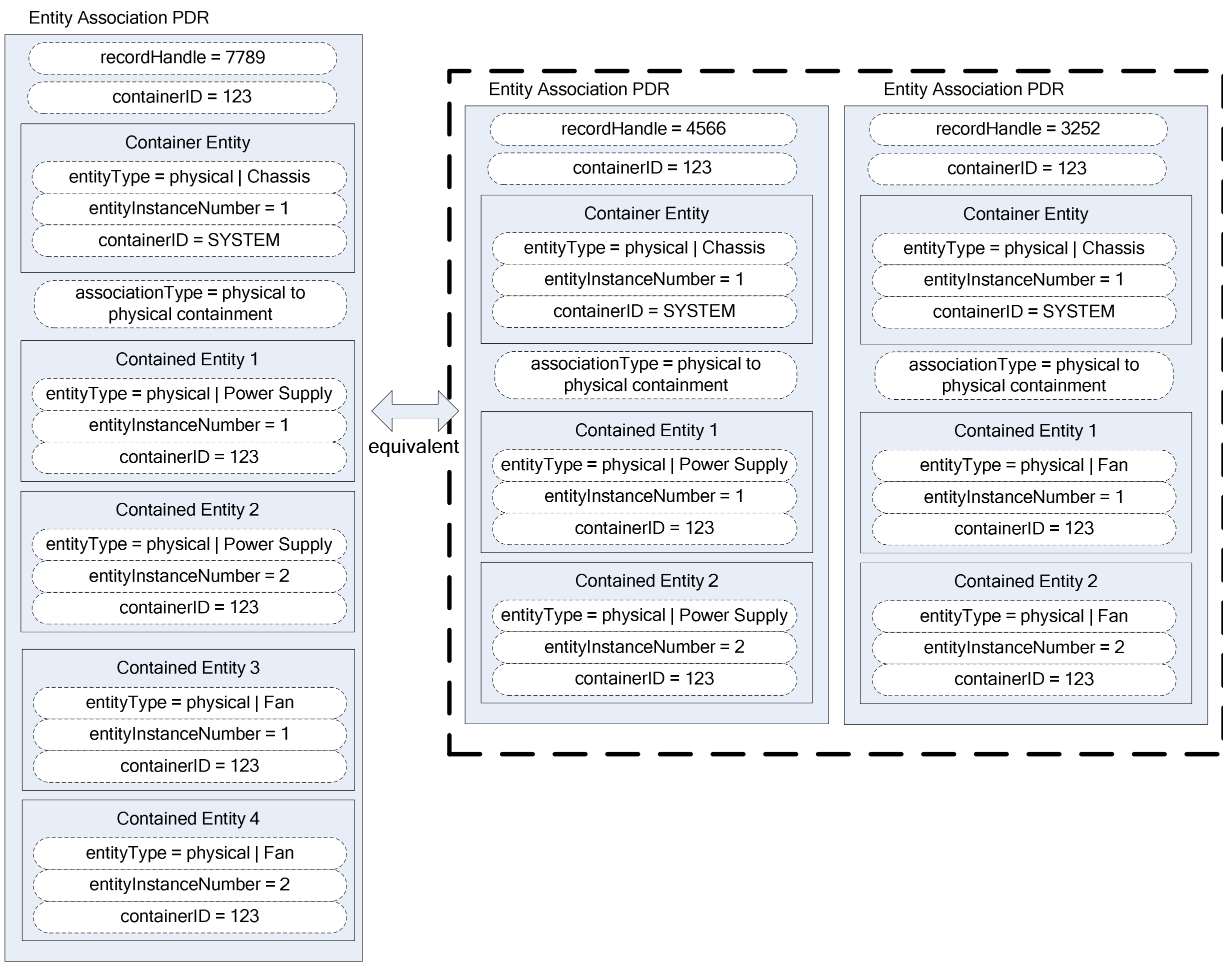


Figure 10 – Linked Entity Association PDRs

11.4 Logical Containment Associations

Entity Association PDRs can also be used to represent the relationship between logical entities and other entities. A logical containment association identifies which physical and logical entities are contained in a given logical container entity. A logical containment association can also consist of a physical container entity that contains logical entities.

This type of association is typically used to group items that have a common parameter that is monitored or controlled. For example, power supplies might be grouped into a logical power supply because they form a redundant power supply subsystem.

The example PDR in Figure 11 shows a logical power supply 1 that contains physical power supply 1 and a physical power supply 2. In this example, the containerIDs in the enclosed Entity Identification Information do not reference the containerID of this overall PDR, but instead reference a container entity from a different PDR. This follows from the previous example where containerID 123 corresponds to physical chassis 1. The explanation for this is provided in 11.5.

A logical containment association can have logical entities, physical entities, or both as contained entities. The container entity must always be defined as a logical entity.

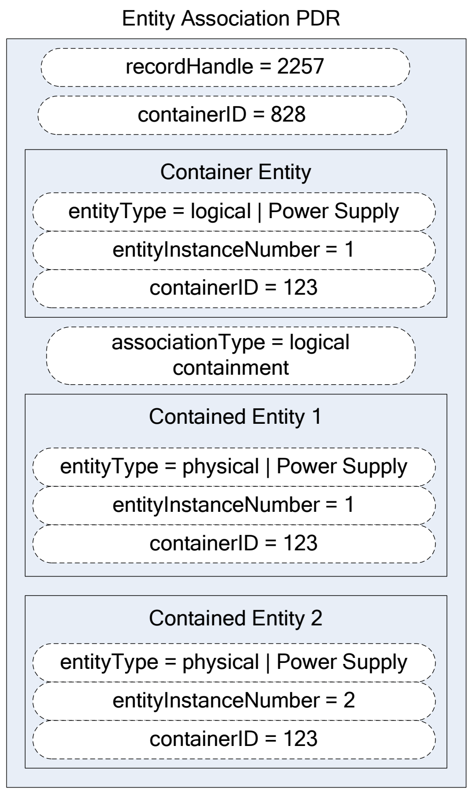


Figure 11 – Logical Containment PDR

11.5 Sensor/Effecter Associations with Logical Entities

Sensors and effecters can be associated with logical entities in the same way that they can be associated with physical entities. Figure 12 shows a state sensor that provides redundancy status and that has a sensor-to-entity association to logical power supply 1. Note that containerID 123 follows from the previous example where containerID 123 corresponds to physical chassis 1.

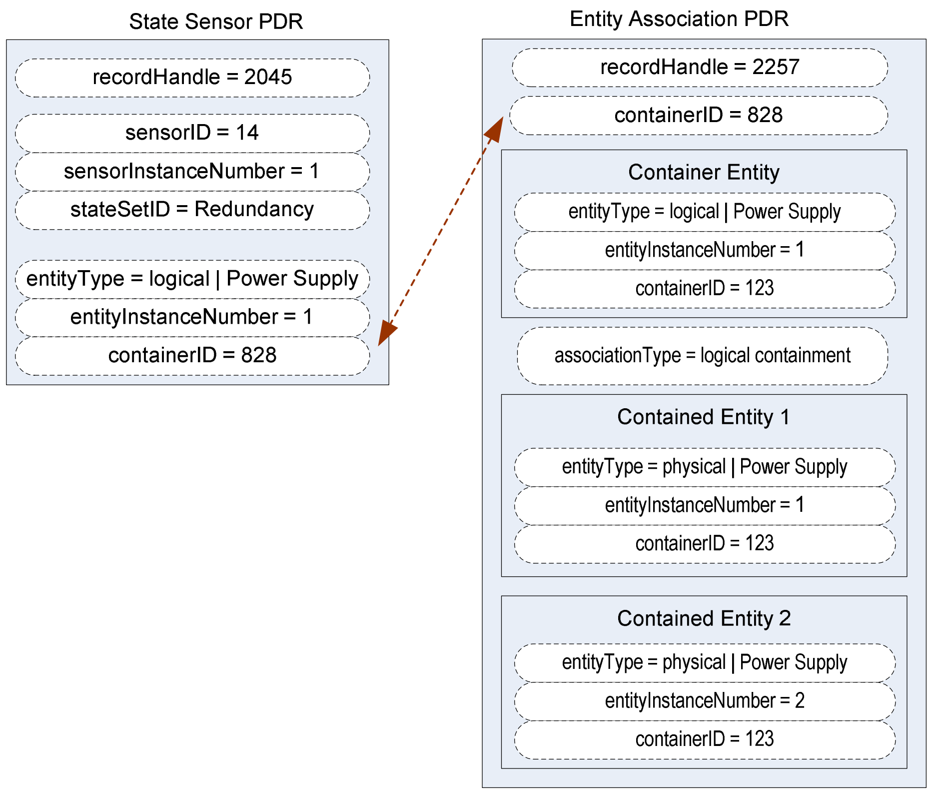


Figure 12 – Sensor/Effecter to Logical Entity Association

11.6 Merged Entity Associations

Figure 13 presents a merged example that illustrates the different aspects and types of entity associations that were introduced in previous sections 11.1 through 11.5. The PDRs in the top portion of Figure 13 represent sensors and physical-to-physical containment associations. The lower half of Figure 13 has PDRs that are related to the sensor and containment associations that define a logical power supply. Together, these PDRs model a system that is represented in the block diagram shown in Figure 14.

The Entity Association PDR that defines the contained entities for logical power supply 1 uses 123 as the containerID in the Entity Identification Information for the contained physical power supplies rather than 828, the containerID for the logical association, for the following reasons:

- An entity that is contained in both physical and logical containment associations should use the containerID that corresponds to a physical containment association.
- The Entity Identification Information values for a given entity must be the same for all references to the entity within the PDRs. A given entity cannot be identified using different container IDs in different associations.

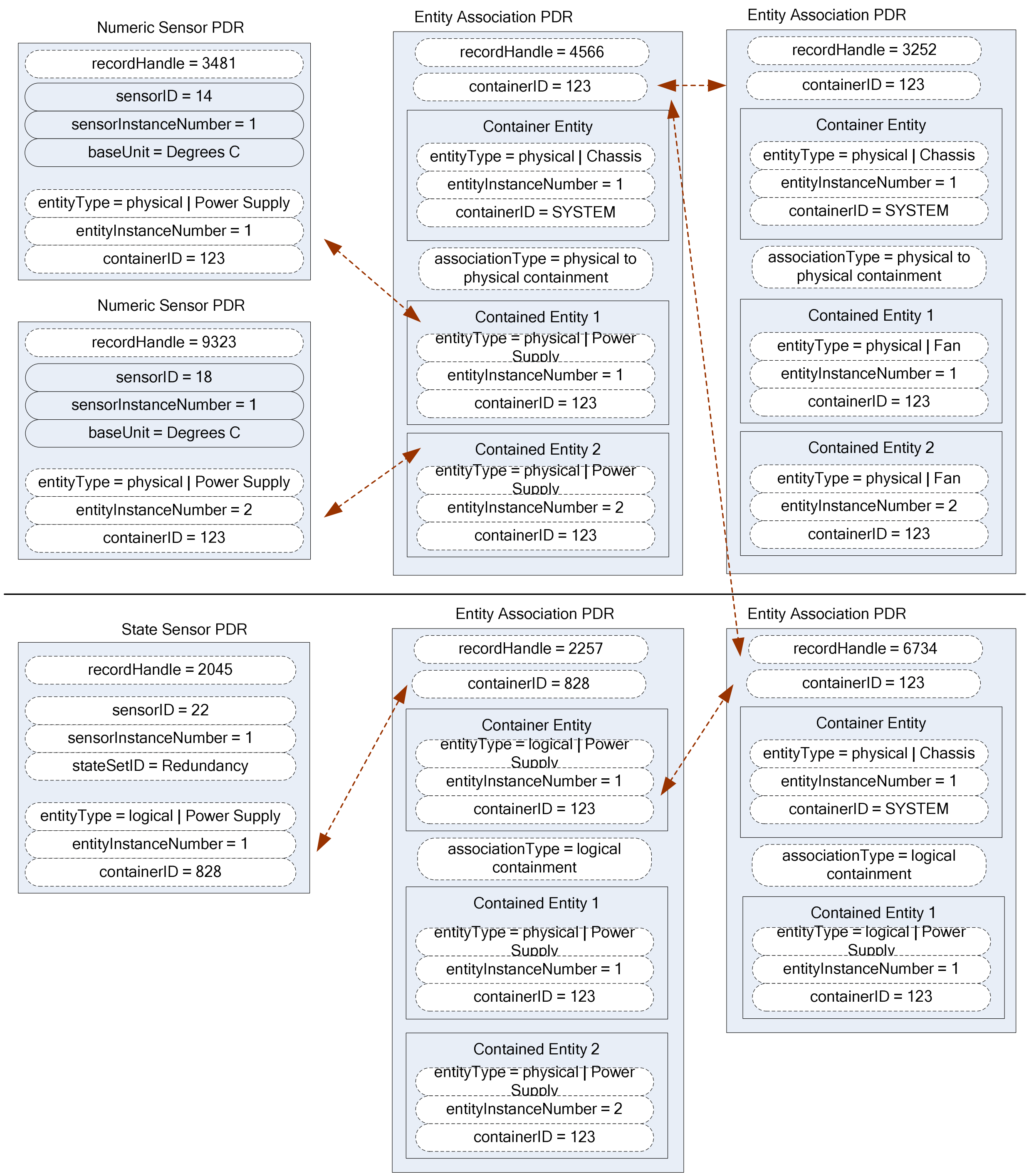


Figure 13 – Merged Entity Association PDR Example

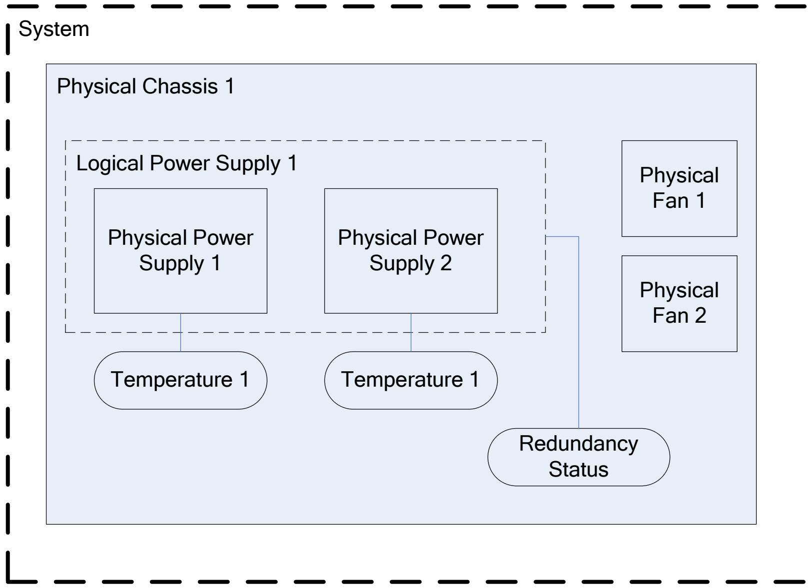


Figure 14 – Block Diagram for Merged Entity Association PDR Example

11.7 Separation of Logical and Physical Associations

Logical associations may be thought of as something that is layered on top of the physical association hierarchy. The previous example identifies container entity 123 (which corresponds to Physical Chassis 1) as the container entity for both physical and logical association PDRs. The types of associations are handled through separate PDRs, which separates the types of associations and helps avoid confusion when a given entity is part of more than one association.

Figure 14 highlights this by showing the physical-to-physical association PDRs in the upper part of the figure and the logical containment PDRs in the lower part.

11.8 Designing Association PDRs for Monitoring and Control

Following is one method for creating or designing PDRs for a simple system:

1) Identify the physical entities and assign them Entity Identification Information values:
   a) Identify the topmost physical container entities and give them the containerID for "system".
   b) Assign each remaining physical entity a different containerID value using whatever approach works best for the implementation. (For example, containerID values could be assigned sequentially starting from 1, or 1000 if it necessary to have a value that is more readily distinguishable as a being a containerID.)
2) Create Entity Association PDRs for the physical-to-physical containment associations.
3) Create the Sensor PDR, Effecter PDR, or other PDRs that are associated with the physical entities, and set the Entity Identification Information based on the containment PDRs that were created earlier.

4) Create the PDRs for any logical entities and set the containerID value for the containing entity to the containerID for the appropriate physical container entities.

5) Create the Sensor PDR, Effecter PDR, or other PDRs that reference those logical entities.

11.9 Terminus Associations

Many PDRs that are related to monitoring and control include a value called the PLDM Terminus Handle. This is an opaque value that is used solely within the PDRs in a given repository as a means of identifying the records that are associated with a particular terminus. The Terminus ID (TID) is a value that is used with PLDM messaging as a way to identify a particular terminus. A PDR called the PLDM Terminus Locator PDR is used to bind the PLDM Terminus Handle and the TID for a given terminus.

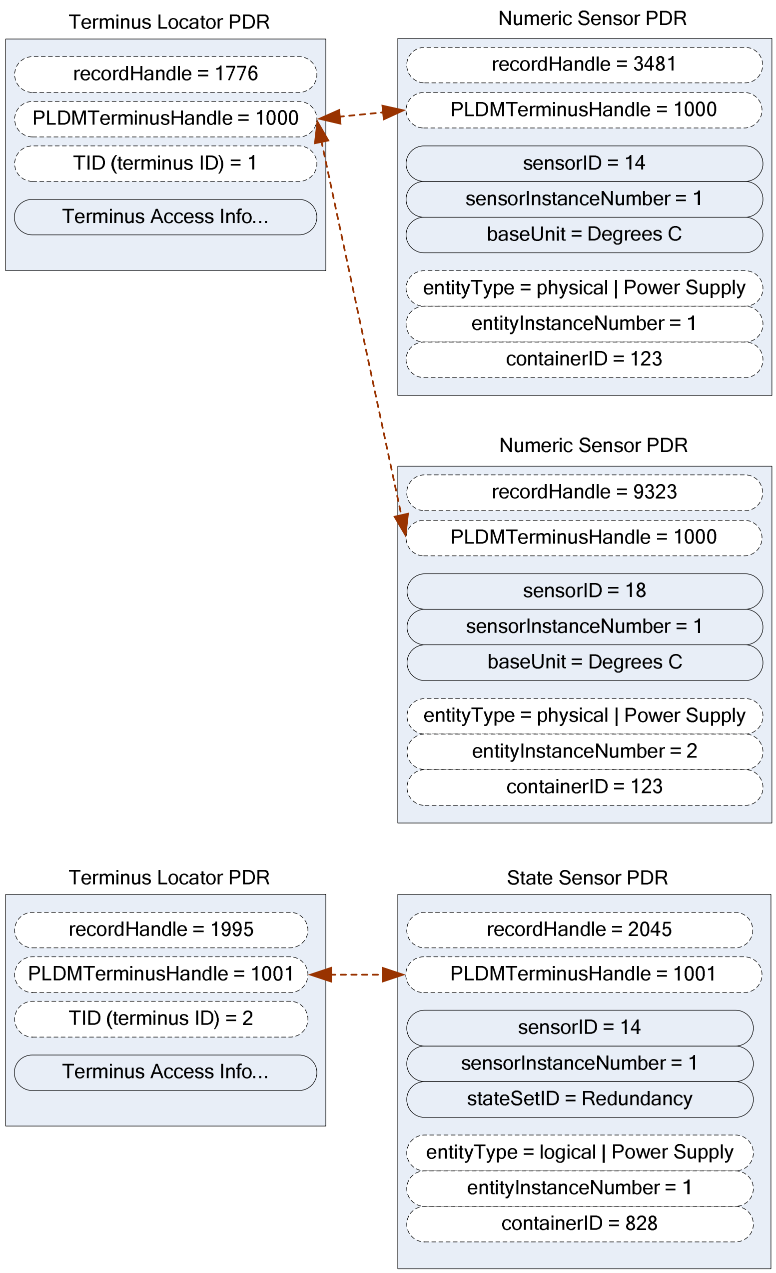
An overview of PLDM Terminus Handles and TIDs is given in 12.1. Figure 15 provides an illustration of the relationship of the PLDM Terminus Handle and TID and how they are used within the PDRs.

The association of entities with sensors and effecters is independent of the terminus that provides access to the sensor or effecter. Sensors and effecters are associated with the entity that is being monitored or controlled rather than the entity that is providing the PLDM terminus that is used to access the sensor or effecter. For example, if a system board entity has a voltage sensor and a temperature sensor, the voltage sensor could be provided through one terminus and the temperature sensor through a different terminus. Both sensors would be associated with the same system board entity, however.

Because Entity Association PDRs may have content in them that has associations with more than one terminus, the PLDM Terminus Handle is used to identify which terminus *provided* the PDR rather than which terminus *is associated with* the PDR. For example, this information can be used to identify when PDR information has been provided by an add-in card so that the PDRs can be updated if the add-in card is removed. In many applications, such as mapping PLDM to CIM, the PLDM Terminus Handle information in an Entity Association PDR can be ignored.

Figure 15 also shows how the PLDMTerminusHandle field is used to identify which sensor PDRs are accessed through a particular terminus. The example shows two different termini providing sensors for the system. The terminus with TID 1 is bound to PLDMTerminusHandle 1000 using the Terminus Locator PDR with recordHandle 1776; the terminus with TID 2 is bound to PLDM Terminus Handle 1001 using the Terminus Locator PDR with recordHandle 1995.

PLDMTerminusHandle 1000 is associated with the PDRs for two numeric temperature sensors that are then associated with physical power supplies 1 and 2. PLDMTerminusHandle 1001 is associated with a single redundancy state sensor that is associated with logical power supply 1. Figure 16 shows a block diagram of these relationships. Note that while this example shows different termini monitoring different entities, different termini can also provide sensors that monitor a common entity. For example, one terminus could provide voltage sensors for a processor while another terminus could provide a temperature sensor for the same processor.



988

989 **Figure 15 – TID and PLDM Terminus Handle Associations**

990 Figure 16 shows a block diagram representation of a hypothetical system that is consistent with the
991 terminus-to-sensor associations shown in Figure 15.

992 The example contains three management controllers. Management Controller 3 implements a PLDM
993 terminus that includes a PLDM State Sensor that provides the redundancy status of logical power supply
994 1. Management Controller 2 implements a PLDM terminus that supports PLDM access to temperature
995 sensors for physical power supplies 1 and 2. Management Controller 2 also holds the Primary PDR
996 Repository for the system. Management Controller 1 represents a management controller or some other
997 party that is accessing the PLDM subsystem. Management Controller 1 gets its view of the PLDM

subsystem by accessing the PDRs in the Primary PDR Repository provided by Management Controller 2. Although this example shows one terminus per management controller, more than one terminus can be implemented in a management controller.

The PLDM Messaging cloud represents PLDM messaging connectivity between these three controllers. In an actual implementation, this connectivity would be accomplished using a transport protocol and physical medium that supports PLDM messaging, such as MCTP over SMBus/I^2C.

The example PDRs in Figure 15 are a subset of the PDRs that would be needed to represent the system shown in Figure 16. For example, in addition to the Terminus Locator and Sensor PDRs, Entity Association PDRs would identify that physical chassis 1 contains physical power supplies 1 and 2, logical power supply 1, and a physical system board 1; that system board 1 contains Management Controllers 1, 2, and 3; and so on.

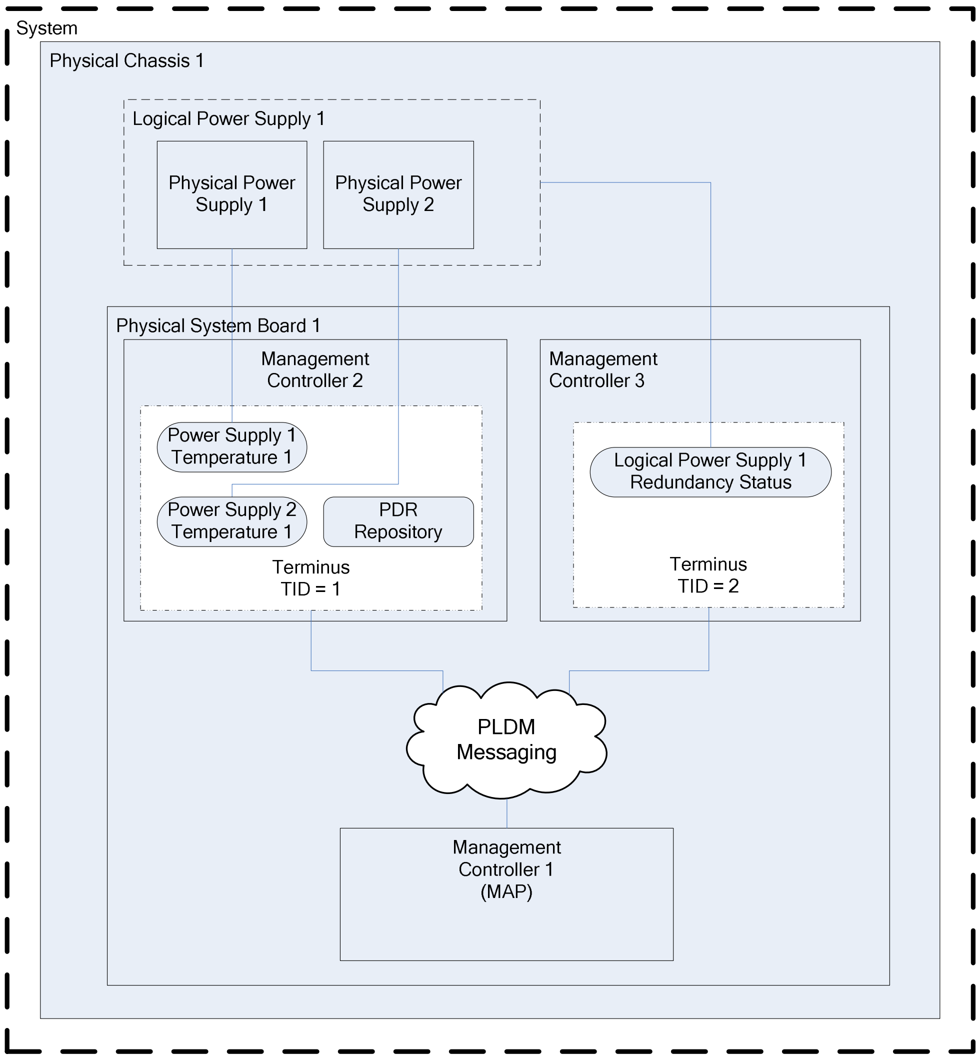


Figure 16 – Block Diagram of Terminus to Sensor Associations

11.10 Interrupt Associations

Platform interrupts represent logical or physical signals that may be monitored or controlled by PLDM, such as NMIs, IRQs, software interrupts, and so on. PLDM State Sensors and PLDM State Effecters can be used to monitor or control platform interrupts.

11.10.1 Interrupt Association PDR

PLDM includes a type of Association PDR called an Interrupt Association PDR that can be used to identify the relationship between one or more interrupt source entities and the target entity for a platform interrupt. The Interrupt Association PDR also identifies which sensor or effecter is associated with the source entity. (Because a given target may receive interrupts from multiple sources, the sensor or effecter is typically associated with the source entity rather than the target entity.)

Two kinds of interrupts can be monitored by a state sensor:

- **Received** interrupt associations identify when an interrupt target entity has received an interrupt from an interrupt source entity.
- **Requested** interrupt associations identify when an interrupt source has issued an interrupt request to an interrupt target entity.

Received interrupts and requested interrupts have different state sets. Thus, received and requested interrupts are differentiated by the state set that is used with the sensor. Effecters will typically use only the state sets for requested interrupts.

11.10.2 Interrupt Association Example

This section presents an example of using an Interrupt Association PDR. In this example, processor 1 is the interrupt target entity that is associated with PCIe Bus 1 and Management Controller 2 as potential interrupt source entities. Management Controller 1 provides the implementation of two sensors that report whether interrupts have been received from those sources.

For this example, assume that each state sensor detected that an interrupt occurred and subsequently generated an event message on that state change. The event message itself indicates only that "Sensor 14 in TID 2 has entered state x". The PDRs are used to interpret this information as follows:

1) The TID that is received in the event message is used to locate the PLDM Terminus Locator record for the terminus. From this, the PLDMTerminusHandle is obtained.
2) The PLDMTerminusHandle and sensorID value are used to locate the State Sensor PDR for the sensor that triggered the event message. This PDR indicates that the stateSetID equals the "Interrupt" state set. The state set definition indicates that the value "x" means "received interrupt detected".
3) The Entity Identification Information in the State Sensor PDR indicates that the interrupt is associated with Management Controller 1, which implies that Management Controller 1 is the source entity for the interrupt.
4) At this point, the combination of the information in the event message and the state sensor PDR yields the following interpretation of the event message:
   – "Sensor 14 in TID 2 has detected that an interrupt has been received from Management Controller 1".
5) This information does not identify the target of the interrupt, however. To identify the target, the PLDMTerminusHandle and sensorID are used to locate the Interrupt Association PDR that identifies the target.

The format of the Interrupt Association PDR in Figure 17 is similar to that of the containment association PDRs shown earlier. The main difference is that sensorID information is provided in conjunction with the

Entity Identification Information for the interrupt source entities. This additional information is required because a given source entity may be the source of more than one interrupt. The sensorID information provides the mechanism for differentiating different interrupts from the same interrupt source entity.

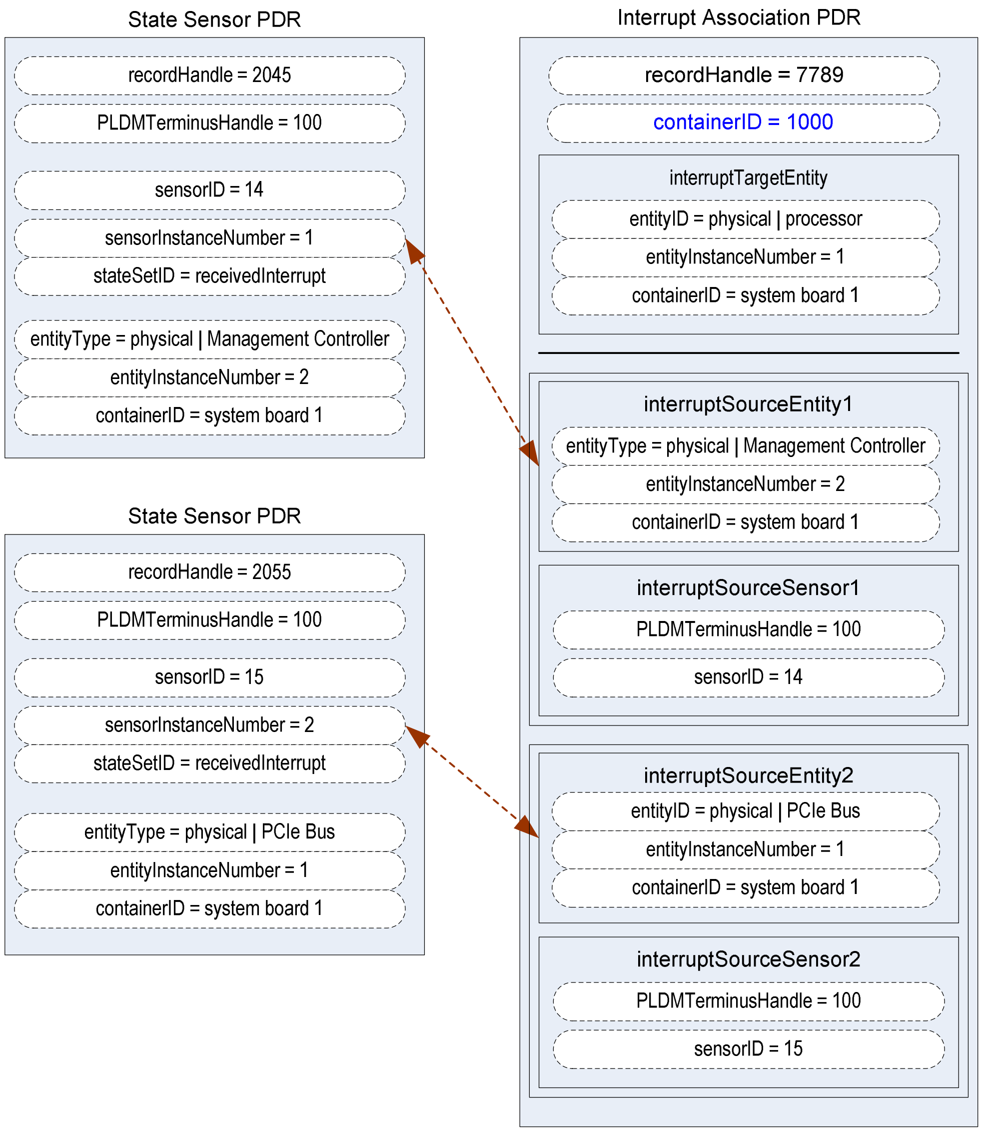


Figure 17 – Received Interrupt Association Example

12 PLDM Terminus

A PLDM terminus is the point of communication termination for PLDM messages and the PLDM functions associated with those messages. A terminus must be uniquely identifiable so that PLDM PDRs can associate semantic information with it. Additionally, a terminus must be identifiable when it generates asynchronous messages, such as event messages. This identification is accomplished through a value called the Terminus ID (TID).

12.1 TIDs, PLDM Terminus Handles, and Terminus Locator PDRs

The TID is primarily used in PLDM messages to identify which terminus generated an asynchronous message, such as an event message. The PLDM Terminus Handle is a value that is used within a PDR Repository to identify PDRs that are associated with a particular terminus. Thus, the PLDM Terminus Handle is defined only within the scope of a particular PDR Repository. A PDR called the Terminus Locator PDR is used to associate a TID with a Terminus Handle. The Terminus Locator PDR also includes information that describes how the terminus is accessed using PLDM messaging.

12.2 Requirements for Unique TIDs

The assignment of unique TIDs to termini is required in the following situations:

- Unique TIDs are required for implementations that use PDRs for describing sensors, effecters, and associations within and among termini.
- Unique TIDs are required when an implementation exposes a PLDM Event Log in order to discriminate events from different termini when reading the log.

12.3 Terminus Messaging Requirements

PLDM termini that meet this specification must implement PLDM Request (command) and Response messages per DSP0240. Additionally, a Management Controller that implements the Event Receiver function must be able to accept and process at least one Event Message request while it is processing other (non-Event Message) requests. Similarly, a device the generates Event Messages must be able to accept an incoming request while it is waiting for the response for the event message.

It is recommended that a terminus can accept and track requests from multiple requesters if the terminus is used in an implementation where it is likely to receive simultaneous requests from multiple parties.

12.4 Terminus Locator PDRs

The Terminus Locator PDR forms the association between a TID and PLDM Terminus Handle for a terminus. The Terminus Locator PDR thus binds a given terminus and the semantic information that is provided through the PDRs for the terminus. Figure 18 illustrates the relationship between a TID and PLDM Terminus Handle.

The Terminus Locator PDR also provides additional information about a terminus, such as how it can be accessed through PLDM messages (hence the name "Terminus Locator"), and whether the terminus and set of PDRs associated with that terminus should be considered present.

If the terminus has a UID or UUID, the Terminus Locator PDR may also hold a copy of the UID/UUID value. This value provides an additional mechanism to help verify that the PDRs associated with the terminus are correct for the particular terminus instance.

The relationship between the PDRs and PLDM Messaging to and from a a given terminus is identified using the following data in the Terminus Locator PDR. (This information is expressed using multiple fields within the actual record format.)

- The PLDM Terminus Handle is used to identify PDRs that are associated to a particular terminus. It is used only within the scope of a particular PDR Repository.
- The TID identifies a terminus for PLDM messaging, particularly for identifying messages that come from a given terminus. A PLDM Terminus Locator PDR associates the TID with the PLDM Terminus Handle that is used for accessing the PDRs that are associated with the terminus.
- The Terminus Access Info consists of a list of protocols and additional information, such as addressing, which enables a party to send PLDM messages to the terminus.

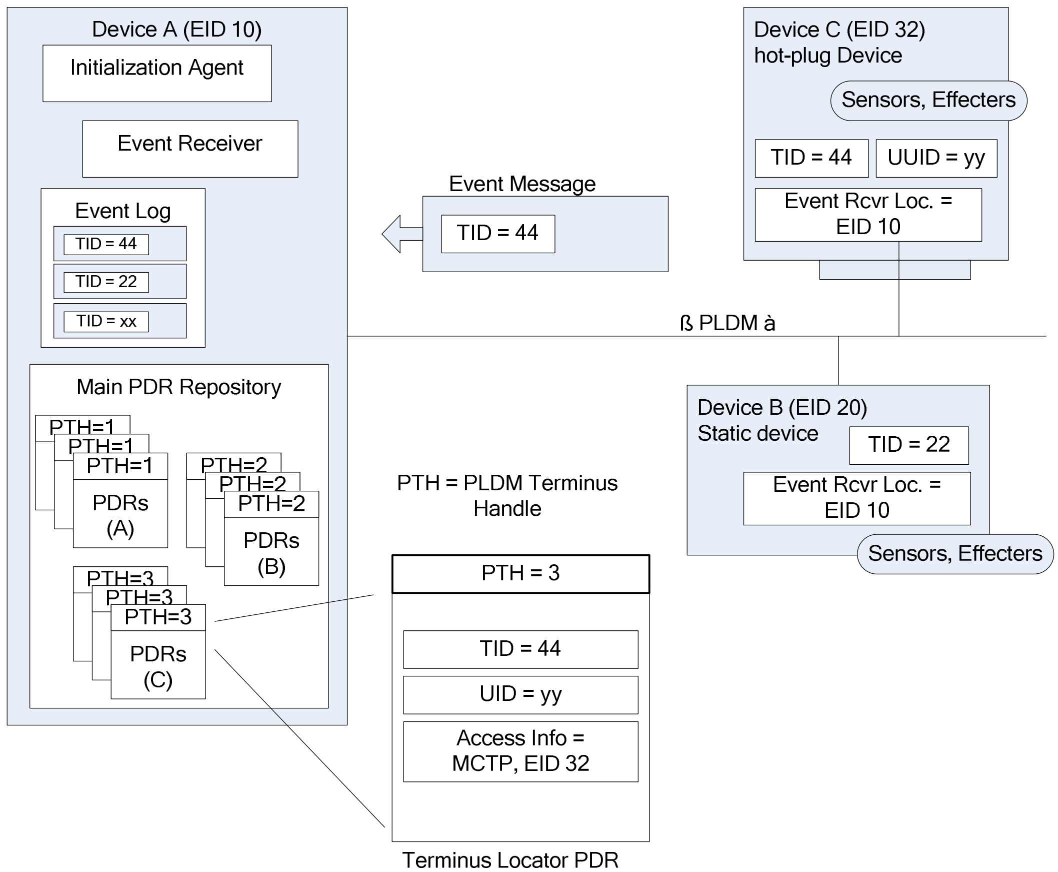


Figure 18 – Example of TID and PLDM Terminus Handle Relationships

12.5 Enumerating Termini

A party that accesses the Primary PDR Repository can use the PDRs to enumerate the termini by listing and examining the Terminus Locator PDRs.

12.5.1 General

To support alternative platform configurations and hot-plug devices, the PDR Repository may have PDRs in it for termini that might not be present. This enables the PDR Repository to hold a superset of information for the possible termini that might be installed in the system. This helps enable implementations that support different configurations of termini using a preconfigured, static set of PDRs.

To support this, the Terminus Locator PDR contains a field that indicates whether the record itself is valid. A terminus may also have a state sensor associated with it that reports whether the terminus is present and available for use (described in 12.5.3).

The following rules apply to using Terminus Locator PDRs for enumerating termini. When it is stated that a terminus should be ignored, it is not an error condition. It means that the status of the terminus is unknown and from a PLDM point-of-view should be treated as if it did not exist at all.

- A terminus must have a Terminus Locator PDR that is marked as valid in order to be considered present. Only one Terminus Locator PDR is allowed to be valid at a time for a given

PLDM Terminus Handle within a PDR Repository. It is an error condition if multiple Terminus Locator PDRs exist and are simultaneously marked as valid for a given PLDM Terminus Handle.

- If the terminus has a sensor associated with it that reports Terminus State, the sensor must indicate that the terminus is present. Otherwise, the terminus and its associated PDRs should be ignored.
- If the terminus has a sensor associated with it that reports Terminus State and the Terminus State information cannot be accessed because the operationalState of the sensor is not “enabled”, the terminus and its associated PDRs should be ignored.

12.5.2 Unlisted or Absent Termini

PDRs for a particular terminus should be ignored under the following conditions:

- The PDR does not have an associated Terminus Locator PDR.
- The PDR is related to a terminus that has an associated Terminus Locator PDR that is marked invalid or is not present based on a presence sensor.

References to termini (for example, PLDM Terminus Handles) should be ignored under the following conditions:

- The reference does not have an associated Terminus Locator PDR.
- The reference is associated with a Terminus Locator PDR that is marked invalid or is not present based on a presence sensor.

These conditions do not apply to OEM or vendor-defined PDRs.

12.5.3 Terminus Presence Using Terminus State Sensors

In some implementations, termini may need to be added or removed as devices are added to or removed from the platform or as platform configurations are changed. This can be handled by updating the validity field in the Terminus Locator PDRs or by updating the PDRs to add or remove Terminus Locator PDRs. Correspondingly, other PDRs that are associated with the terminus may also be updated, added, or removed. Updating PDRs may not be warranted in some implementations, such as when the implementation would have otherwise been able to use a static configuration of PDRs.

A more dynamic way of indicating terminus presence is to associate a terminus with a "Terminus State Sensor". A Terminus State Sensor is a type of PLDM Composite State Sensor that is associated with a logical entity of type "PLDM Terminus" using a sensor to entity association. The sensor returns state set enumerations for "Presence status" and "Operational status". A Terminus State Sensor may be implemented as a sensor at the terminus itself, or it may be implemented as a sensor under another terminus.

13 PLDM Events

PLDM events are primarily related to changes of PLDM sensor states or states that are related to the operation of PLDM or the PLDM subsystem itself.

NOTE: PLDM events are not the same as CIM indications. There will typically not be a one-to-one correspondence between PLDM events and CIM indications. In some cases, a PLDM event may trigger a MAP to generate indications or entries in a CIM record log, while in other cases a PLDM event may be used solely to update CIM properties to eliminate or reduce polling by the MAP, or to report information about the internal health or operation of the PLDM subsystem that is not exposed through CIM.

13.1 PLDM Event Messages

PLDM Event Messages are PLDM monitoring and control messages that are used by a PLDM terminus to asynchronously report PLDM events to a central party called the PLDM Event Receiver.

13.2 PLDM Event Receiver

The destination for event messages within PLDM is called the Event Receiver. The Event Receiver function is implemented by a PLDM terminus within the platform management subsystem. Multiple termini can send Event Messages to the Event Receiver function. The SetEventReceiver command is used to give the location of the Event Receiver function to termini that generate event messages.

A PLDM subsystem implementation can have only one PLDM Event Receiver function enabled at a given time. It is expected that typical implementations will always assign the same Event Receiver location. However, the location of the Event Receiver function is allowed to be changed during PLDM subsystem operation. For example, some implementations may do this to support a failover of the Event Receiver function, or to migrate it to a management controller that is hot plugged into the system, and so forth.

13.3 PLDM Event Logging

PLDM Event Logging defines an interface through which event messages that have been received at the Event Receiver can be saved in an area of storage called the PLDM Event Log for later retrieval. Event logging includes mechanisms for storing and time-stamping event records, determining characteristics of the log (such as its capacity), and reading and clearing the contents of the log.

Additionally, "virtual" PLDM Event Messages may be internally generated within the terminus that is providing the PLDM Event Log function and directly logged without appearing as PLDM Event Messages on any external interface.

A PLDM subsystem shall contain only one PLDM Event Log function.

Additional information about event logging is provided in section 23.

13.4 PLDM Event Log Clearing Policies

The PLDM Event Log can use different policies for automatically clearing entries from the log (Table 4). The active policy is configured through the SetPLDMEventLogPolicy command. Refer to the specification of this command for policy support requirements.

Table 4 – PLDM Event Log Clearing Policies

Policy	Description
Fill and Stop	The PLDM Event Log stops accepting new entries after it has become full. The log does not automatically clear. It must be cleared using the ClearPLDMEventLog command. This policy does not utlize any parameters.
FIFO	When the log is full, the oldest *N* entries are automatically deleted when the next entry is received. This policy uses a single parameter, *N*. *N* may be a fixed or configurable parameter, depending on the implementation. An implementation can also express N as a percentage of the log (NPercentage) instead of as an integral number of entries.
Clear on Age	When the log has filled past a threshold number of entries, *M*, the age of the first *N* entries is checked to see if they have been in the log for more than a given age interval. If the *N*th entry is older than the age interval, the first *N* entries are automatically cleared from the log. If the log is less than *M* entries full, entries are retained indefinitely, regardless of their age.

Policy	Description
	This policy uses three parameters: Age, N, and M.The Age interval, the number of automatically cleared entries, *N*, and the threshold value, M, may be fixed or configurable parameters, depending on the implementation. The policy may also be implemented with *N* and *M* given as percentages of the log (MPercentage and NPercentage) instead of an integral number of entries.

13.5 Oldest and Newest Log Entries

Unless otherwise specified, when the terms *old*, *older*, *oldest*, *new*, *newer*, and *newest* are used to refer to PLDM Event Log entries, the terms refer to the time that the event was entered into the log rather than the time stamp of the entry. This is because the setting of the log time stamp clock might be changed during system operation, making it possible for temporally newer log entries to have time stamps that refer to an older time than temporally older entries.

13.6 Event Receiver Location

The information that is used by a given terminus to send messages to the Event Receiver function (such as addressing) is referred to as the Event Receiver Location information. Event Receiver Location information is transport dependent; for example, for MCTP the information would consist of the EID (MCTP Endpoint ID) of the Event Receiver. Additionally, the Event Receiver Location information may vary on a per-terminus basis, depending on the requirements of the transport and medium. The PLDM Transport binding specifications define how the Event Receiver Location is set for a particular transport and medium.

PLDM supports a SetEventReceiver command that enables the Event Receiver Location information to be delivered to termini that generate event messages. This approach provides the following characteristics:

- It eliminates the need to specify a well known address for the Event Receiver function for each different medium and transport.
- It supports assigning the Event Receiver function to a different location, which could be used to
  - support failover of the Event Receiver function to another device
  - enable the Event Receiver function to be handled by an alternative device that gets added into the system
  - support a situation in which the Event Receiver function is on a medium where its address changes during PLDM operation
- It provides a mechanism that helps synchronize the generation of event messages with the availability of the Event Receiver function.

13.7 PLDM Event Log Entry Formats

Table 5 shows the general format that is used for all PLDM Event Log entries.

Table 5 – PLDM Event Log Entry Format

Byte	Type	Field
0	enum8	**entryType** value: { PLDMPlatformEvent, OEMTimestampedEntry, OEMEntry }

1	uint8	**entryDataLength** The size in bytes of the entryData field.
variable	–	**entryData** Data for the entry, dependent on the entryType. If entryType = PLDMPlatformEvent, the entryData format is given in Table 6. If entryType = OEMTimestampedEntry, the entryData format is given in Table 7. If entryType = OEMEntry, the entryData format is given in Table 8.

13.8 PLDM Platform Event Entry Data Format

Table 6 specifies the format used for the entryData field in PLDM Event Log entries that use the PLDMPlatformEvent value for the entryType field.

Table 6 – Platform Event Entry Data Format

Byte	Type	Field
0	sint8	**entryTimestampUTCOffset** The UTC offset for the log entry time stamp in increments of 1/2 hour special value: 0xFF = unspecified
1:5	uint40	**entryTimestampSeconds** This value corresponds to a 40-bit unsigned integer that represents the number of seconds since midnight UTC of January 1, 1970 (not counting leap seconds).
6	uint8	**entryTimestamp100s** This value provides a number of 1/100ths of a second added to entryTimestampSeconds. value: 0 to 99 special value: 0xFF = unspecified. Use this value if the implementation timestamps entries to no finer than a one second resolution.
variable	–	**eventData** The eventData format is the same as the format for the request parameters of the PlatformEventMessage command (see Table 13).

13.9 OEM Timestamped Event Entry Data Format

Table 7 specifies the format used for the entryData field in PLDM Event Log entries that use the OEMTimestampedEntry value for the entryType field.

Table 7 – OEM Timestamped Event Entry Data Format

Byte	Type	Field
0:3	uint32	**vendorIANA** The IANA Enterprise Number for the vendor that is defining the OEMData. The list of Enterprise Numbers can be found at www.iana.org/protocols/. special value: 0 = unspecified.

4	sint8	**entryTimestampUTCOffset** The UTC offset for the log entry time stamp in increments of 1/2 hour special value: 0xFF = unspecified
5	uint40	**entryTimestampSeconds** This value corresponds to a 40-bit unsigned integer that represents the number of seconds since midnight UTC of January 1, 1970 (not counting leap seconds).
10	uint8	**entryTimestamp100s** This value provides a number of 1/100ths of a second added to entryTimestampSeconds. value: 0 to 99 special value: 0xFF = unspecified. This value is used if the implementation timestamps entries to no finer than a one second resolution.
variable	0 to 32 bytes	**OEMData** 0 to 32 bytes of OEM-specific data that is specified by the vendor identified by vendorIANA

13.10 OEM Event Entry Data Format

Table 8 specifies the format used for the entryData field in PLDM Event Log entries that use the OEMEntry value for the entryType field. The format is similar to the OEM Timestamped Event Entry Data format (shown in Table 7), except that it does not include PLDM-defined time stamp fields.

Table 8 – OEM Event Entry Data Format

Byte	Type	Field
0:3	uint32	**vendorIANA** The IANA Enterprise Number for the vendor that is defining the OEMData special value: 0 = unspecified
variable	0 to 32 bytes	**OEMData** 0 to 32 bytes of OEM-specific data that is specified by the vendor identified by vendorIANA

14 Discovery Agent

The Discovery Agent function is responsible for discovering termini, assigning them unique TID values, and assigning them the address of the Event Receiver function.

If the implementation is maintaining a Primary PDR Repository, the Discovery Agent may also be required to automatically create or update PDRs to support devices such as hot-plug devices that may be dynamically added or removed from the system. This includes the following actions:

- creating records such as Terminus Locator PDRs
- extracting Device PDR information and merging it into the Primary PDR Repository
- updating associating records to link Device PDR information into the overall context of the platform management subsystem

Any OEM PDRs in the Device PDR information that are identified to be copied to the Primary PDR Repository are also added to the Primary PDR Repository by the Discovery Agent.

14.1 Assignment of TIDs and Event Receiver Location

Following are the support requirements for assignment of TIDs and the launching of the Initialization Agent by a Discovery Agent within a PLDM implementation:

- All termini must support the SetTID command.
- All termini that generate PLDM Event Messages shall support the SetEventReceiver command. Termini that do not generate PLDM Event Messages are not required to support the SetEventReceiver command.
- The Discovery Agent function is responsible for discovering termini and assigning them unique TID values. (A default TID setting may be pre-configured for a PLDM terminus if the terminus is statically configured into the platform. This setting must be able to be overridden using the SetTID command.)
- The Initialization Agent function is responsible for initializing PLDM sensors and effecters and setting Event Receiver location information into the termini. (A default Event Receiver setting may be pre-configured for a PLDM terminus if the terminus is statically configured into the platform. This setting must be able to be overridden using the SetEventReceiver command.)The Initialization Agent function is described in more detail in section 15.
- When PDRs are used, the Initialization Agent is also responsible for maintaining corresponding Terminus Locator PDR information.
- A terminus must have its Event Receiver information set before it can begin to issue PLDM Event Messages.
- A terminus that has standby power should retain its TID and Event Receiver settings. When the terminus comes back online, it can use that information for event messaging without requiring Event Receiver re-initialization.
- A terminus should retain its TID and Event Receiver settings during a given PLDM subsystem operation.
- Termini that are to be rediscovered (that is, termini that are not statically configured into the system and may lose PLDM communication temporarily, which might occur in different platform power states) must have a separate unique and persistent ID that can be associated with the terminus. For example, if a terminus is hot-plug, it should have a universally unique ID (UUID).
- TIDs are not required to persist or remain constant across PLDM subsystem restarts, unless the system is using PDRs or exposes a PLDM Event Log. In such cases, TIDs must be persistently stored by the termini or reassigned to the same value by the Discovery Agent function.
- A MAP or other entity that is accessing a PLDM subsystem should not cache TIDs because TIDs might change if the PLDM subsystem is reset or reinitialized.
- Termini on hot-plug cards must have a UUID or be associated with a terminus on the same card that has a UUID.
- Implementations that do not use PDRs can assign TIDs in any manner, including not assigning them at all. In this case, the implementation must define its own mechanisms for identifying and tracking termini and event messages from termini.

14.2 UUIDs for Devices in Hot-Plug or Add-in Card Applications

If the device is intended to be used on an add-in or hot-plug card, it may be required to support a universally unique ID (UUID) depending on higher-level system requirements or initiatives. In general, add-in cards that plug into standardized I/O connections and are used in multiple vendor systems, such as PCIe add-in cards, are required to use UUIDs so that multiple instances of the same card can be detected.

14.3 UID Implementation

If a terminus is required to have a unique ID (UID), how the UID is implemented depends on the component and how the device manufacturer intends the device to be used in a system. For example, it is the device manufacturer's choice whether the entire UID must be configured by the system integrator after purchasing the device, or a number of pre-configured UIDs in the device are selectable by a pin or non-volatile configuration selection, or the UID is permanently embedded in the device. Typically, each device will have fuses, PROM, EPROM/EEPROM, or some other non-volatile mechanism for holding the unique ID that is configured either during device manufacture or when the device is integrated into a system.

14.4 More Than One Terminus in a Device

The Terminus Locator PDR contains a containerEntity field that can be used to identify the entity that contains the terminus. This field provides the mechanism to identify when multiple termini are within the same device or are located within the same entity.

14.5 Examples of PDR and UUID Use with Add-in Cards

Figure 19 and Figure 20 present examples of how Device PDRs, UUIDs, and Terminus Locator PDRs work together to identify PLDM termini on add-in cards, such as hot-plug add-in cards, that may be dynamically inserted or removed during PLDM subsystem operation. Both examples illustrate MCTP-based implementations. However, the approach may be extrapolated to other transport types.

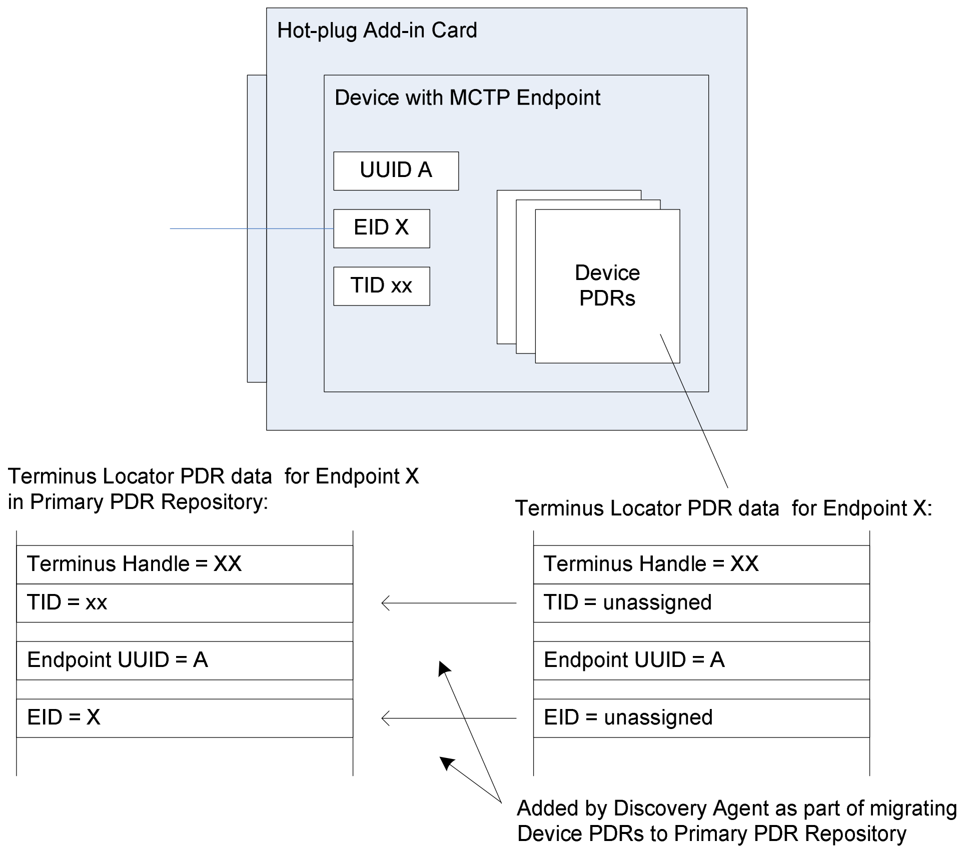


Figure 19 – Hot-Plug Add-in Card with Single PLDM Terminus

Figure 19 shows an add-in card that has a single PLDM terminus that is accessed through a single MCTP endpoint. The terminus is persistently and uniquely identified within the PLDM subsystem by a UUID that is associated with the endpoint and the terminus. This UUID is recorded in a partially filled-in Terminus Locator PDR that is part of the Device PDRs that are provided by the add-in card. The UUID can also be read by issuing a GetTerminusUID command to the terminus. The Device PDRs also report the presence of and semantic information about sensors, effecters, and other functions on the add-in card.

The Terminus Locator PDR from the Device PDRs returns "unassigned" values for the Endpoint ID (EID) and Terminus ID (TID) fields because those values are unavailable before the card has been discovered and initialized by MCTP and the PLDM Discovery Agent within the PLDM subsystem. It also eliminates the need for the terminus to update those Device PDRs whenever TID or EID values are assigned or changed. The Discovery Agent sets the TID for the terminus and adds the EID and TID values to the Terminus Locator Record PDRs when they are integrated into the Primary PDR Repository. The Discovery Agent then synthesizes other PDRs as necessary to link the add-in card into the overall semantic information of the PLDM subsystem. For example, the Discovery Agent may create association PDRs that associate the add-in card with a particular bus and connector within the system.

The Discovery Agent is also responsible for keeping those records up-to-date if EID assignments change during PLDM subsystem operation and for deleting or invalidating the PDRs that are associated with the card and its termini if it detects that the card has been removed.

Figure 20 shows an add-in card that has several MCTP endpoints, each with its own PLDM terminus. One terminus is within an MCTP Bridge device that provides the Device PDRs for all the termini on the card. Additionally, the MCTP Bridge provides a UUID that identifies the overall card for MCTP. All MCTP endpoints are defined relative to MCTP Bridge function based on the position of their routing information in the routing table.

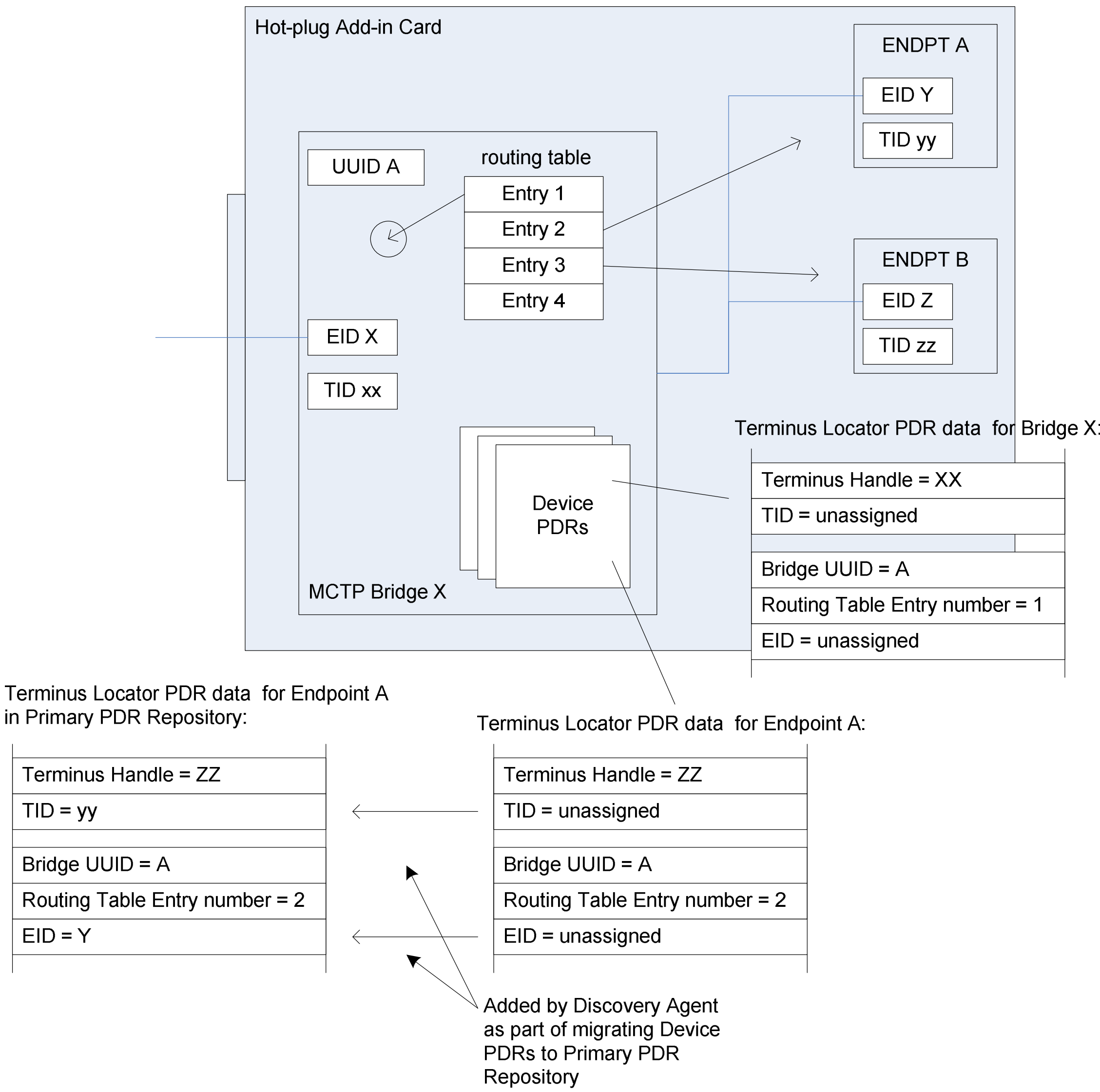


Figure 20 – Hot-Plug Add-in Card with Multiple PLDM Termini

In Figure 20, the MCTP Bridge itself is associated with the first routing table entry, Endpoint A is associated with the second entry, and Endpoint B is associated with the third entry. The Device PDRs hold Terminus Locator PDRs for each terminus that is on the add-in card. These PDRs uniquely identify each terminus using two pieces of information: the UUID of the MCTP Bridge and the position of a routing table entry that is associated with the terminus. The routing table entry positions must not change during PLDM subsystem operation. This approach eliminates the need for Endpoints A and B to have their own support for UUIDs.

15 Initialization Agent

This section describes the role and operation of the Initialization Agent function in a PLDM subsystem that uses PDRs.

15.1 General

PLDM sensors are not required to completely self-initialize and enable themselves upon PLDM subsystem startup or upon power state changes of the device that is hosting the sensor. Thus, low-cost devices are not required to have non-volatile configuration resources. Additionally, the mechanism provides options for overriding default configurations of sensors and event generation.

The Initialization Agent is a function that initializes message generation and sensor configuration as described by Sensor Initialization PDRs. The Initialization Agent function normally runs whenever the platform management subsystem is first powered up, upon system Hard and Soft Resets, and on certain other transitions. Fields in the Sensor Initialization PDRs indicate the system transitions on which a given sensor is initialized.

The Initialization Agent is also responsible for setting the Event Receiver Location information and enabling event message generation.

The Sensor Initialization PDRs hold information that describes the default threshold values, states, and event generation settings for sensors that are initialized by the Initialization Agent function. Sensor Initialization PDRs are required only for sensors that are initialized by the Initialization Agent. Sensors that are self-initializing or are initialized through some mechanism that is outside the PLDM specifications do not need Sensor Initialization PDRs.

The Initialization Agent function thus eliminates the need for all sensors to retain their own non-volatile storage for their default settings, and also provides a mechanism to retrigger any events that may have been transmitted before the Event Receiver function was ready to accept them.

Only one Initialization Agent function is supported within a given PLDM subsystem. The Initialization Agent shall be implemented behind the same terminus that provides the Primary PDR Repository for the PLDM subsystem.

15.2 PLDM and Power State Interaction

The Initialization Agent may need to re-initialize certain sensors or termini as the result of a change of system power state. An implementation should avoid requiring the Initialization Agent to execute because of low-latency power state transitions, such as transitions between ACPI S0 and S1, or S1 and S2 states. The implementation should instead ensure that termini retain their settings across low-latency power state transitions.

The Sensor Initialization PDRs include a field that tells the Initialization Agent upon which system transitions a given sensor should be initialized.

15.3 RunInitAgent Command

PLDM does not specify a particular mechanism for an implementation to use to detect when to run the Initialization Agent function. For example, it does not specify how a management controller would detect a system hard reset or power-up transition. In some implementations, it will be useful to have another management controller, system firmware, or another entity decide that the Initialization Agent should run. For example, system firmware may decide that the Initialization Agent should be run after a BIOS update. To enable this, PLDM defines a RunInitAgent command that can be used to launch the Initialization Agent “on demand.” The command includes a parameter that can select a subset of Sensor Initialization PDRs to be used.

15.4 Recommended Initialization Agent Steps

The following presents an outline of the steps for an Initialization Agent in a system implementation that includes Initialization PDRs.

1) Stop the Event Receiver function from accepting events received from any interface but the system (host) interface.
2) Scan the PDR Repository for Terminus Locator PDRs. Collect a list of valid termini that require initialization. (A field in the Terminus Locator PDR indicates whether any sensors/effecters in the terminus require initialization, and, if so, whether event messaging should be enabled after the controller has been initialized.)
3) For each terminus in the list, perform the following actions:
   - Turn off Event Generation by using the SetEventReceiver command. If a terminus does not respond to the SetEventReceiver command, take that terminus off the list.
   - Use the GetTID command to determine whether the terminus has a TID. If so, leave that value unchanged unless it is already assigned to another terminus. If not, use the SetTID command to assign a TID to the terminus.
   - Scan the PDR Repository for Initialization PDRs (for example, numeric sensor initialization PDRs or state sensor initialization PDRs) that are associated for the terminus. For each PDR that is found, perform the following actions:
     - Set the sensor type, sensor thresholds, and hysteresis as directed by the PDR using the SetSensorThresholds and SetSensorHysteresis commands.
     - Use the appropriate enabling command (for example, SetNumericSensor Enables if the sensor is a numeric sensor) to enable scanning and event generation per the PDR.
4) Enable the Event Receiver function to accept event messages.
5) For each terminus with a Terminus Locator PDR, enable event message generation using the SetEventReceiver command or leave it disabled (A field in the Management Controller Device Locator record indicates whether event messaging should be enabled after the controller has been initialized.)

16 Terminus and Event Commands

This section describes the commands that are used by PLDM termini that implement PLDM monitoring and control as defined in this specification. The command numbers for the PLDM messages are given in section 30.

If a PLDM terminus is implemented to provide access to any of the capabilities of this specification, the Mandatory/Conditional (M/C) requirements shown in Table 9 apply.

Table 9 – Terminus Commands

Command	M/C	Reference
SetTID (see DSP0240)	M	See 16.1.
GetTID (see DSP0240)	M	See 16.2.
GetTerminusUID	C [1]	See 16.3.
SetEventReceiver	C [2][3]	See 16.4.
GetEventReceiver	C [2]	See 16.5.
PlatformEventMessage	C [2][4]	See 16.6.

[1] See section 16.3.

[2] Mandatory for termini that generate PLDM Event Messages.

[3] Sending the SetEventReceiver command is Mandatory for termini that implement the Initialization Agent function.

[4] Accepting the PlatformEventMessage is Mandatory for termini that implement the Event Receiver function.

16.1 SetTID Command

The SetTID command is used to set the TID for a PLDM terminus. This command is typically used by the PLDM Discovery Agent function. This command is defined in DSP0240.

16.2 GetTID Command

The GetTID command is used to retrieve the present TID setting for a PLDM terminus. This command is defined in DSP0240.

16.3 GetTerminusUID Command

The GetTerminusUID command is used to obtain a unique ID for the terminus when it is necessary to differentiate between different instances of identical devices that hold the terminus (such as two otherwise identical add-in cards), or when it is necessary to track a particular terminus that may be "relocated," such as a terminus on an add-in card that is moved from one slot to another.

The GetTerminusUID command shall be supported by a terminus when the terminus is on a hot-pluggable or other add-in card where the platform management subsystem implementation is expected to discover and automatically adopt PLDM capabilities in the terminus (such as sensors) without requiring separate configuration steps to be taken outside of PLDM. See 14.3 and 14.2 for more information.

If more than one terminus is on the same card, only the terminus that provides PDRs for the add-in card is required to support the GetTerminusUID command. Table 10 describes the format of the command.

Table 10 – GetTerminusUID Command Format

Type	**Request Data**
–	none
Type	**Response Data**
enum8	**completionCode** value: { PLDM_BASE_CODES }
UUID	**UUIDValue**

16.4 SetEventReceiver Command

The SetEventReceiver command is used to set the address of the Event Receiver into a terminus that generates event messages. It is also used to globally enable or disable whether event messages are generated from the terminus. Table 11 describes the format of the command.

Table 11 – SetEventReceiver Command Format

Type	Request Data

enum8	**eventMessageGlobalEnable**
	This value is used to enable or disable event message generation from the terminus.
	value: {
	disable, // Disable all event message generation from the terminus. The transportProtocolType and // eventReceiverAddressInfo fields must be populated in the request, but shall be ignored // by the receiver of this command.
	enable, // Enable event message generation from the terminus. This setting is combined with the // enable and disable settings for individual sensors, effecters, and so on. For example, both // this global enable and the individual enable for a sensor must be set to “enable” for event // messages to be generated for the sensor.
	// Globally enabling event generation causes all sensors and effecters within the terminus to // reassess their event status. The sensors and effecters will generate event messages if // their present state does not match their default initialization state.
	}

enum8	**transportProtocolType** This value is provided in the request to help the responder verify that the content of the eventReceiverAddressInfo field used in this request is correct for the messaging protocol supported by the terminus. This value is defined in DSP0245. The content of the eventReceiverAddressInfo field used in this command depends on the transportProtocolType and in some cases also the medium that the terminus is using. The command shall be rejected and an INVALID_PROTOCOL_TYPE completionCode returned if the transportProtocolType is incorrect.
varies	**eventReceiverAddressInfo** This value is a medium and protocol-specific address that the responder should use when transmitting event messages using the indicated protocol. The format and specification of this field depends on the transportProtocolType. The bytes in this field may contain additional information, such as protocol version, medium type, transport binding type, and so on. The format of this field is defined in the PLDM-to-Transport binding specification identified by the transportProtocolType field. If the transportProtocolType value from DSP0245 is "Vendor-specific", the overall eventReceiverAddressInfo format is vendor-specific. However, the first field of the eventReceiverAddressInfo must be a uint32 that holds a value corresponding to the IANA Enterprise Number of the vendor or organization that has specified the format.
Type	**Response Data**
enum8	**completionCode** value: { PLDM_BASE_CODES, INVALID_PROTOCOL_TYPE=0x80 }

16.5 GetEventReceiver Command

The GetEventReceiver command is used to verify the values that were set into an Event Generator using the SetEventReceiver command. Table 12 describes the format of the command.

Table 12 – GetEventReceiver Command Format

Type	Request Data
–	**none**
Type	**Response Data**
enum8	**completionCode** value: { PLDM_BASE_CODES }
enum8	**transportProtocolType** This value indicates the transportProtocolType that the terminus uses for its eventReceiverAddress and the format of the eventReceiverAddress field. This value is defined in DSP0245.
varies	**eventReceiverAddress** This value is a medium and protocol-specific address that the responder should use when transmitting event messages using the indicated protocol. The format and specification of this field depends on the protocolType. The bytes in this field may contain additional information, such as protocol version, medium type, transport binding type, and so on. The format of this field is defined in the PLDM-to-Transport binding specification identified by the transportProtocolType field. If the transportProtocolType value from DSP0245 is "Vendor-specific", the overall eventReceiverAddress format is vendor-specific. However, the first field of the eventReceiverAddress must be a uint32 that holds a value corresponding to the IANA Enterprise Number of the vendor or organization that has specified the format. The value in the eventReceiverAddress field is unspecified if the eventReceiverAddress has not yet been initialized. Otherwise, the field returns the last value that was set using the SetEventReceiver command.

16.6 PlatformEventMessage Command

PLDM Event Messages are sent as PLDM request messages to the Event Receiver using the PlatformEventMessage command. Because PLDM requests have associated responses, this approach provides a positive acknowledgement that the event message was received. Table 13 describes the format of the command.

Table 13 – PlatformEventMessage Command Format

Type	Request Data
uint8	**formatVersion** Version of the event format (the format and definition of the following bytes): 0x01 for this format.
uint8	**TID** Terminus ID for the terminus that originated the event message

enum8	**eventClass**
	value: {
	sensorEvent, // Events that are issued for events that are related to PLDM numeric and
	// state sensors. See Table 14 for the eventData format for this eventClass.
	effecterEvent, // See Table 15 for the eventData format for this eventClass.
	}

var	**eventData** Event data based on the eventClass
Type	**Response Data**
–	**completionCode** value: { PLDM_BASE_CODES, UNSUPPORTED_EVENT_FORMAT_VERSION = 0x81 }

enum8	**status**
	value: {
	noLogging, // The event message has been accepted. The implementation does
	// not provide a PLDM Event Log at the Event Receiver.
	loggingDisabled, // The event message was accepted but will not be logged because
	// logging is disabled.
	logFull, // The event message was accepted but will not be logged because
	// the log is full.
	acceptedForLogging , // The event message has been accepted and queued up for
	// logging. Note that under some conditions the message may not be
	// logged if the log becomes full or is disabled before the queued
	// message is processed.
	logged // The event message was accepted. The implementation has
	// confirmed that the event has been logged prior to sending the
	// response.
	loggingRejected // The implementation has accepted the event message but has
	// rejected logging it based on filtering of the event message content.
	}

16.7 eventData Format for sensorEvent

Table 14 defines the format of the eventData field in PLDM Event Messages for the sensorEvent class. This field includes event data for PLDM state sensor and numeric sensor events, and for events related to changes of the sensor's operational state.

Table 14 – sensorEvent Class eventData Format

Type	Request Data
uint16	**sensorID** The sensorID is the value that is used in PDRs and PLDM sensor access commands to identify and access a particular sensor within a terminus.

enum8	**sensorEventClass**
	value: {
	sensorOpState, // Events from a PLDM state or numeric sensor that are related to // changes of the sensor's operational state
	stateSensorState, // Events from a PLDM state sensor that are related to a change // in the present state from the set of states that the sensor is // monitoring
	numericSensorState // Events from a PLDM numeric sensor that are related to a change // in the present state from the set of states that the sensor is // monitoring. Also returns the reading value that triggered the event.
	}

For sensorEventClass = stateSensorState	
uint8	**sensorOffset** Identifies which state sensor within a composite state sensor the event is being returned for 0x00 = first state sensor, 0x01 = second state sensor, and so on
enum8	**presentState** The event state value from the state change that triggered the event message
enum8	**previousState** The event status value for the state from which the present state was entered special value: This value shall be set to the same value as presentState if the previousState is unknown, which may be the case for events that are generated on the first status assessment that occurs after a sensor has been initialized.
For sensorEventClass = numericSensorState	
enum8	**presentState** The eventState value from the state change that triggered the event message
enum8	**previousState** The eventState value for the state from which the present state was entered special value: This value shall be set to the same value as presentState if the previousState is unknown (which may be the case for events that are generated on the first status assessment that occurs after a sensor has been initialized).

Type	Request Data
enum8	**sensorDataSize** The bit width and format of reading and threshold values that the sensor returns value: { uint8, sint8, uint16, sint16, uint32, sint32 }
uint8 | sint8 | uint 16 | sint16 | sint32 | uint32	**presentReading** The present value indicated by the sensor. The sensorDataSize field returns an enumeration that indicates the number of bits used to return the value. NOTE: An implementation may either periodically sample the value and return the most recently collected sample, or sample the value at the time that the presentReading is requested. The presentReading value is not required to return a correct value and must be ignored while the presentState value of the sensor is Unspecified.
For sensorEventClass = sensorOpState	
enum8	**presentOpState** The sensorOperationalState value from the state change that triggered the event message
enum8	**previousOpState** The sensorOperationalState value for the state from which the present state was entered special value: This value shall be set to the same value as presentOpState if the previousOpState is unknown, which may be the case for events that are generated on the first status assessment that occurs after a sensor has been initialized.

1469 ## 16.8 eventData Format for effecterEvent

1470 Table 15 defines the format of the eventData field in PLDM Event Messages for the effecterEvent class.
1471 This field supports events for changes of the effecter's operational state.

1472 **Table 15 – effecterEvent Class eventData Format**

Type	Request Data
uint16	**effecterID** The effecterID is the value that is used in PDRs and PLDM effecter access commands to identify and access a particular effecter within a terminus.

enum8	**effecterEventClass**
	value: {
	effecterOpState // Events from a PLDM state or numeric effecter that are related to
	// changes of the effecter's operational state
	}

For effecterEventClass = effecterOpState	
enum8	**presentOpState** The effecterOperationalState value from the state change that triggered the event message.
enum8	**previousOpState** The effecterOperationalState value for the state from which the present state was entered. special value: This value shall be set to the same value as presentOpState if the previousOpState is unknown, which may be the case for events that are generated on the first status assessment that occurs after an effecter has been initialized.

17 PLDM Numeric Sensors

This section provides information the describes the characteristics and operation of PLDM Numeric Sensors.

17.1 Sensor Readings, Data Sizes

PLDM Numeric Sensors can return a present reading value. The value is returned as a binary integer. The size of this integer and whether it is signed can vary on a per-sensor basis. The PLDM GetSensorReading command includes a parameter in its response that indicates the format used for returning the reading. The same format is used for any thresholds and hysteresis values that are used for request or response parameters. Additionally, the data size is supported in PDR information for the sensor.

17.2 Units and Reading Conversion

The sensor commands do not intrinsically identify what type of unit, such as volts, amps, or RPM, is used for the sensor's present reading value. Additionally, the value may require scaling to convert the value to normalized units, such as millivolts (mV), nanoseconds, and so on.

For example, microcontrollers commonly incorporate an 8-bit analog-to-digital (A/D) converter. If the converter is monitoring a signal where the 0x00 value of the conversion corresponds to 0 volts and a 0xFF reading corresponds to 4.00 volts, each count of the converter corresponds to a value of 4.0/255 ~= 15.686274 mV per count. Converting a particular reading from counts into volts requires multiplying the reading by a conversion factor. A reasonable guideline is that the conversion factor should be accurate to at least 4 times the resolution of the converter. In this case, the resolution of the converter is 1 part in 255, which would require the accuracy of the conversion factor to be to better than 1 part in 1020, which rounds up to four significant digits, or 15.69 mV per count.

To avoid the need for a floating point format for sensor readings and the need for multibyte multiplications and divisions in simple devices, PLDM readings are returned as “raw” integers that are converted to normalized units by the consumer of the reading data by using a specified conversion formula and sensor-specific conversion factors. The consumer of the PLDM sensor reading data will be a device serving a role such as a MAP that has more resources for doing mathematical operations. This approach avoids burdening simple devices with the conversion task.

The conversion formula is specified in 27.7. The conversion factors must be provided by the vendor or designer of the particular sensor implementation. The PDR for a numeric sensor supports returning conversion factors and the type of units (volts, amps, and so on) used for a particular numeric sensor.

17.3 Reading-Only or Threshold-Based Numeric Sensors

A particular instance of a PLDM Numeric Sensor can return just a numeric reading or a numeric reading *and* a threshold-based status. These sensors are referred to as "reading-only" or "threshold-based" numeric sensors.

17.4 Readable and Settable Thresholds

A given instance of a PLDM Numeric Sensor may have thresholds that are readable through the GetSensorThresholds command or that are settable through the SetSensorThresholds command. The PDR information can indicate whether a particular numeric sensor uses thresholds and, if so, which thresholds are supported and whether they are settable.

17.5 Thresholds, Present Status, and Event Status

PLDM Numeric Sensors that are threshold-based have associated thresholds against which the reading is compared.

17.5.1 Threshold Severity Levels

Each threshold is associated with a severity that is related to how far the threshold is from the normal range of the sensor. Unless otherwise specified, the severity level is generally based on the view that a sensor is monitoring parameters that are associated with a physical entity. Table 16 describes the threshold severity levels.

Table 16 – Threshold Severity Levels

Severity Level	Description
warning	The reading is outside of normal expected operating range but the monitored entity is expected to continue to operate normally. The warning may be an indication of a condition that is expected to become critical or fatal with time unless steps are taken to counter the condition that is causing the warning. As such, warning thresholds are usually implemented when some automated or remote action can be taken as a result of seeing the warning. For example, an application might use a warning related to an over-temperature condition to take actions to increase the system cooling or decrease its load. A warning related to increasing levels of correctable errors in a memory device might trigger an action to schedule a service call to replace the memory device before it fails.
critical	The reading is outside of supported operating range. Monitored entities might operate abnormally, have transient failures, or propagate errors to other entities under this condition. Prolonged operation under this condition might result in degraded lifetime for the monitored entity. The monitored entity will usually return to normal operation if the condition returns to a warning or normal level.
fatal	The reading is outside of rated operating range. Monitored entities might experience permanent failures or cause permanent failures to other entities under this condition. Remedial actions might require replacement of the monitored entity or other components.

17.5.2 Upper and Lower Thresholds

A given threshold for a PLDM Numeric Sensor can either be an upper or a lower threshold. Upper thresholds are for tracking events that become more severe as the reading becomes more positive numerically. Lower thresholds are for events that become more severe as the reading becomes more negative numerically.

PLDM has three upper thresholds: upper warning, upper critical, and upper fatal. Similarly, PLDM has three lower thresholds: lower warning, lower critical, and lower fatal. By convention, these thresholds occur in the following order: lower fatal, lower critical, lower warning, upper warning, upper critical, and upper fatal. Lower fatal corresponds to the most negative threshold value, and upper fatal corresponds to the most positive threshold value. This order is illustrated in Figure 21 on page 63.

A sensor is not required to implement all thresholds. For example, a sensor that monitors for an over-voltage condition may implement only an upper critical threshold. A sensor that is monitoring a low-RPM condition may implement only lower warning and lower critical thresholds. A temperature sensor may implement both upper and lower thresholds so that it can track both over-temperature and under-temperature conditions.

17.5.3 Present Status

A PLDM Numeric Sensor that uses thresholds returns a presentState value that is based on a simple numeric comparison of the present reading against the sensor to the thresholds and returns the threshold range with which the reading is associated. The presentState value is updated solely based on a numeric comparison of the present reading to the thresholds. For upper thresholds, the present status is based on whether the present reading is greater than or equal to the threshold value. For lower thresholds the status is based on whether the present reading is less than or equal to the threshold value. For example, if the present value is greater than or equal to the value for upper critical threshold but is less than the value for upper fatal threshold, the status will be UpperCritical.

17.5.4 Event Status

The eventStatus field of a PLDM Numeric Sensor is updated based on transitions between the different monitored states of the sensor. Unlike presentState, this status includes the effect of the hysteresis setting. If the hysteresis value for the sensor is equal to one count of the reading, the eventState and presentState values will be the same. Otherwise, the eventStatus setting may vary from the presentStatus due to the effect of hysteresis. See 17.9 for more information about hysteresis and its relationship to eventStatus.

The eventState behavior is also affected by whether the sensor implementation is manual- or auto-rearm (see 17.6).

17.6 Manual Re-arm and Auto Re-arm Sensors

The event state tracking for a sensor can be either auto re-arm or manual re-arm. An auto re-arm sensor updates its eventState automatically whenever the sensor detects that a state transition has occurred.

A manual re-arm sensor retains the most severe event state transition that it has detected since the time the sensor was initialized or since the last time the event status was explicitly cleared (using the rearm operation in the GetSensorReading command). If a new state is assessed that has the same criticality as the previous state, the most recently assessed value shall be returned. For example, if the previous status was upperCritical and the present status is lowerCritical, then upperCritical shall be returned.

Thus, auto re-arm sensors automatically update their status on *any* detected state transition, while manual re-arm sensors automatically update their event status only on detecting a worsening (increasing severity) transition (or upon a transition to a different state of equivalent severity as the previous state).

Re-arming of numeric sensors is done through the GetSensorReading command. Re-arming causes the sensor to internally enter its “initializing” operating state until it next updates its presentStatus and eventStatus. (This update may happen so quickly that the temporary entry into the initializing state is never reflected in the sensorOperationalState parameter of the GetSensorReading command.)

17.7 Update / Polling Intervals and Status Updates

A sensor may periodically collect internal readings and status (that is, it may poll for updates) and respond to a GetSensorReading request with the last collected values, or it may collect the values "on demand" upon receiving the request.

An updateInterval value in the PDR for the sensor provides a way for the requester to determine the maximum time from when a sensor was re-armed or accessed to when the subsequent event status or reading update should have occurred.

For a sensor that polls for updates, the updateInterval corresponds to the nominal polling interval, ±50%. (The ±50% variation is to accommodate manufacturing variations between devices implementing sensors and variations in firmware-based polling intervals.) There is no requirement for a sensor's polling interval to be synchronized (restarted) when a re-arm occurs. A sensor is also allowed to take as long as two polling intervals before updating its state following a re-arm (one interval to recognize the re-arm, and one interval to collect and apply the updated state).

For a sensor that updates "on demand," the updateInterval indicates the maximum time, ±50%, from receiving a GetSensorReading command to when a reading and status update should occur. If the sensor can update itself within the PLDM Request-to-response time (refer to DSP0240), either an updateInterval value of 0 or the actual update interval may be used in the PDR.

If the updateInterval for a given sensor is longer than the PLDM Request-to-response time, the updateInterval must be specified and the sensorOperationalStatus must be returned as "initializing" while the sensor is performing its initial state assessment after being enabled or re-armed.

Because a sensor is allowed to take up to two polling intervals to update after a re-arm, and because the variation is allowed to be ±50%, it may take as long as three nominal polling intervals (two nominal intervals times 1.5) plus a PLDM Request-to-response time before the effect of a re-arm is realized.

17.8 Event Message Generation

A PLDM Numeric Sensor that supports and is enabled to generate event messages shall generate them whenever an Event State (eventStatus) change is detected. To detect changes in the Event State, the sensor implementation must do periodic polling or incorporate some other asynchronous mechanism, such as the occurrence of an interrupt, which causes the sensor to obtain a new reading, the eventStatus to update and an event message to be generated.

17.9 Threshold Values and Hysteresis

Threshold settings for PLDM Numeric Sensors are required to be ordered from numerically most negative to most positive in the following order: lower fatal, lower critical, lower warning, upper warning, upper critical, upper fatal. The hysteresis value is always subtracted from the "upper" thresholds and added to the "lower" thresholds.

Thus, hysteresis is always applied on the transition from a more severe state to a less severe state. For example, assume that a sensor has a hysteresis value of 2, has an upper critical threshold set to 80, and is presently in the "upper warning" state. The sensor will transition to the "upper critical" state when it detects that the reading value reaches a value that is greater than or equal to the threshold setting of 80. The sensor is now in the "upper critical" state. To return to the "upper warning" state, the reading has to drop to 78 (80 minus the hysteresis value of 2).

Figure 21 helps further describe and illustrate the relationships between thresholds, hysteresis, eventStatus, and presentStatus for numeric sensors.

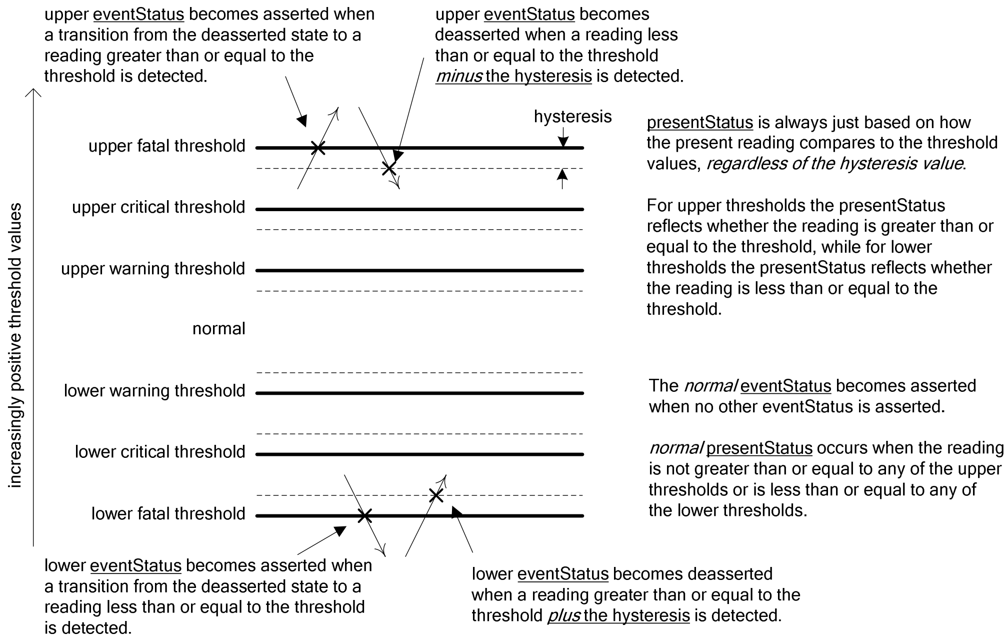


Figure 21 – Numeric Sensor Threshold and Hysteresis Relationships

18 PLDM Numeric Sensor Commands

This section describes the commands for accessing PLDM Numeric Sensors per this specification. The command numbers for the PLDM messages are given in section 30.

If PLDM numeric sensors are implemented, the Mandatory/Optional/Conditional (M/O/C) requirements shown in Table 17 apply.

Table 17 – Numeric Sensor Commands

Command	M/O/C	Reference
SetNumericSensorEnable	M	See 18.1.
GetSensorReading	M	See 18.2.
GetSensorThresholds	O, C [1]	See 18.3.
SetSensorThresholds	O	See 18.4.
RestoreSensorThresholds	O	See 18.5.
GetSensorHysteresis	O, C [2]	See 18.6.
SetSensorHysteresis	O	See 18.7.
InitNumericSensor	C [3]	See 18.8.

[1] The GetSensorThresholds command is required if the SetSensorThresholds command is implemented. Otherwise, the command is optional.

[2] The GetSensorHysteresis command is required if the SetSensorHysteresis command is implemented. Otherwise, the command is optional.

[3] The InitNumericSensor command is required if the sensor requires initialization following any one of the conditions identified in the initConditions field of the PLDM Numeric Sensor Initialization PDR.

18.1 SetNumericSensorEnable Command

The SetNumericSensorEnable command is used to set the operating state of the sensor itself and whether the sensor generates event messages. Changing this state affects only the operation of the sensor; it has no effect on the operational state of the entity or parameter that is being monitored. Event message generation is optional for a sensor. Table 18 describes the format of the command.

Table 18 – SetNumericSensorEnable Command Format

Type	Request Data
uint16	**sensorID** A handle that is used to identify and access the sensor special values: 0x0000, 0xFFFF = reserved
enum8	**sensorOperationalState** The desired state of the sensor This enumeration is a subset of the operational state values that are returned by the GetSensorReading command. Refer to the GetSensorReading command for the definition of the values in this enumeration. value: { enabled, disabled, unavailable }

Type	Request Data
enum8	**sensorEventMessageEnable** This value is used to enable or disable event message generation from the sensor. value: { noChange, disableEvents, enableEvents, enableOpEventsOnly, enableStateEventsOnly} noChange means do not alter the present setting. Use noChange when the sensor does not support event message generation.
Type	**Response Data**
enum8	**completionCode** value: { PLDM_BASE_CODES, INVALID_SENSOR_ID = 0x80, INVALID_SENSOR_OPERATIONAL_STATE = 0x81, EVENT_GENERATION_NOT_SUPPORTED = 0x82 //an attempt was made to enable or disable event generation for a sensor that does not support event message generation. }

18.2 GetSensorReading Command

The GetSensorReading command is used to get the present reading and threshold event status values from a numeric sensor, as well as the operating state of the sensor itself. Table 19 describes the format of the command.

Table 19 – GetSensorReading Command Format

Type	Request Data
uint16	**sensorID** A handle that is used to identify and access the sensor special values: 0x0000, 0xFFFF reserved
bool8	**rearmEventStatus** true = manually re-arm EventStatus after responding to this request Re-arming causes the sensor to enter the “initializing” state until it updates its presentStatus and eventStatus. Sensor implementations shall either update that status immediately upon responding to this command or wait for the conclusion of their polling interval before updating the eventStatus. If event messages are enabled, the status update shall also cause the sensor to issue a corresponding assertion event message based on the eventStatus that it assesses. This includes generating an event message for the "normal" state. false = no manual re-arm

Type	Response Data
enum8	**completionCode** value: { PLDM_BASE_CODES, INVALID_SENSOR_ID = 0x80, REARM_UNAVAILABLE_IN_PRESENT_STATE = 0x81 }
enum8	**sensorDataSize** The bit width and format of reading and threshold values that the sensor returns value: { uint8, sint8, uint16, sint16, uint32, sint32 }
enum8	**sensorOperationalState** The state of the sensor itself value: { enabled, disabled, unavailable, statusUnknown, failed, initializing, shuttingDown, inTest } enabled — Enabled and operating. The sensor is able to return valid presentState, previousState, presentReading, and eventState values. This state can be set through the SetNumericSensorEnable command. The unavailable operational state indicates a condition in which the sensor is unable to assess one of the other state values. This typically transient condition may occur when a sensor is being initialized or has been re-armed. For the following states, the presentState, eventState, and eventDeassertionStatus values shall be set to "Unknown". Other actions related to monitoring by the sensor may also cease in this state. For example, a sensor device that polls to collect monitored values may stop polling. Unless otherwise specified, the following states are not settable through PLDM commands. disabled — The sensor is disabled from returning presentReading and event status values. This state is settable through the SetNumericSensorEnable command. unavailable — The sensor should be ignored due to the configuration of the platform or monitored entity. For example, the sensor is for monitoring a processor temperature, but the processor is not installed. This state is settable through the SetNumericSensorEnable command. statusUnknown — The sensor cannot presently return valid state or reading information for the monitored entity. failed — The sensor has failed. The sensor implementation has determined that it can not return correct values for one or more of its presentState or eventState values. initializing — The sensor is in the process of transitioning to the operating state because the sensor is initializing (starting) or re-initializing. The presentState and eventStatevalues shall be ignored while the sensor is in this state. shuttingDown — The sensor is transitioning to the disabled, failed, or unavailable states. inTest — The sensor is presently undergoing testing. NOTE: The operation of sensor testing and the mechanisms for sensor testing are outside the scope of this specification.
bool8	**sensorEventMessageEnable** value: { noEventGeneration, eventsDisabled, eventsEnabled, opEventsOnlyEnabled, stateEventsOnlyEnabled }

Type	Response Data
enum8	**presentState** The most recently assessed state value monitored by the sensor. If the sensorOperationalState is set to enabled the sensor must return a value other than Unknown for the presentState. If the sensorOperationalState is not set to enabled the sensor shall return Unknown for the presentState. Parties that are using this command should also ignore the presentState value except when sensorOperationalState is set to enabled. Refer to 17.4 for important information about how presentState and eventState are generated. value: { Unknown, Normal, Warning, Critical, Fatal, LowerWarning, LowerCritical, LowerFatal, UpperWarning, UpperCritical, UpperFatal }
enum8	**previousState** The most recently assessed state value monitored by the sensor. If the sensorOperationalState is set to enabled the sensor may temporarily return Unknown for the previousState if the sensor has not yet assessed a previousState value (as may happen immediately after the sensor has become enabled). Otherwise, the sensor must return a value other than Unknown. If the sensorOperationalState is not set to enabled the sensor shall return Unknown for the previousState. Parties that are using this command should also ignore the previousState value except when sensorOperationalState is set to enabled. Refer to 17.4 for important information about how presentState and eventState are generated. value: { Unknown, Normal, Warning, Critical, Fatal, LowerWarning, LowerCritical, LowerFatal, UpperWarning, UpperCritical, UpperFatal }
enum8	**eventState** Indicates which threshold crossing assertion events have been detected. The sensor is required to return one of the specified values in the enumeration. However, the value is required to be valid only when the sensor is in the enabled state. If the sensorOperationalState is set to enabled the sensor may temporarily return Unknown for the eventState if the sensor has not yet assessed a eventState value (as may happen immediately after the sensor has become enabled). Otherwise, the sensor must return a value other than Unknown. The eventState value is set to Unknown when sensorOperationalState is set to any value except enabled. Parties that are using this command should ignore the eventState value under this condition. Refer to 17.4 for additional information about how presentState and eventState are generated. value: { Unknown, Normal, Warning, Critical, Fatal, LowerWarning, LowerCritical, LowerFatal, UpperWarning, UpperCritical, UpperFatal }
uint8 | sint8 | uint16 | sint16 | sint32 | uint32	**presentReading** The present value indicated by the sensor NOTE: The SensorDataSize field returns an enumeration that indicates the number of bits used to return the value. An implementation may either periodically sample the value and return the most recently collected sample, or it may sample the value at the time the presentReading is requested. The presentReading value is not required to return a correct value and must be ignored while the presentState value of the sensor is Unavailable.

18.3 GetSensorThresholds Command

The GetSensorThresholds command is used to get the present threshold settings for a PLDM Numeric Sensor. Table 20 describes the format of the command.

Table 20 – GetSensorThresholds Command Format

Type	Request Data
uint16	**sensorID** A handle that is used to identify and access the sensor special values: 0x0000, 0xFFFF = reserved
Type	**Response Data**
enum8	**completionCode** **value:** { PLDM_BASE_CODES, INVALID_SENSOR_ID = 0x80 }
enum8	**sensorDataSize** The bit width and format of reading and threshold values that the sensor returns value: { uint8, sint8, uint16, sint16, uint32, sint32 } NOTE: The sensorDataSize return value provides an enumeration that indicates the number of bits used to return the threshold values. All six threshold fields must be returned regardless of which thresholds are implemented. If a given threshold is not implemented the implementation can elect to put any value in the corresponding field (0 is recommended). The Numeric Sensor PDRs describe which thresholds are supported.
For sensorDataSize = uint8 or sint8	
uint8 | sint8	**upperThresholdWarning**
uint8 | sint8	**upperThresholdCritical**
uint8 | sint8	**upperThresholdFatal**
uint8 | sint8	**lowerThresholdWarning**
uint8 | sint8	**lowerThresholdCritical**
uint8 | sint8	**lowerThresholdFatal**
For sensorDataSize = uint16 or sint16	
uint16 | sint16	**upperThresholdWarning**
uint16 | sint16	**upperThresholdCritical**
uint16 | sint16	**upperThresholdFatal**
uint16 | sint16	**lowerThresholdWarning**
uint16 | sint16	**lowerThresholdCritical**
uint16 | sint16	**lowerThresholdFatal**
For sensorDataSize = uint32 or sint32	
uint32 | sint32	**upperThresholdWarning**
uint32 | sint32	**upperThresholdCritical**
uint32 | sint32	**upperThresholdFatal**

uint32 | sint32	**lowerThresholdWarning**
uint32 | sint32	**lowerThresholdCritical**
uint32 | sint32	**lowerThresholdFatal**

18.4 SetSensorThresholds Command

The SetSensorThresholds command is used to set the thresholds of a PLDM Numeric Sensor. Values for all threshold parameters must be provided. However, if a particular threshold is not supported by the sensor, the value passed in the corresponding parameter is ignored. To avoid unintended event transitions, it is recommended that the sensor be disabled while changing threshold settings.

Threshold values may be volatile or non-volatile. The level of volatility is reflected in the PDR for the sensor.

Table 21 describes the format of the command.

Table 21 – SetSensorThresholds Command Format

Type	Request Data
uint16	**sensorID** A handle that is used to identify and access the sensor special values: 0x0000, 0xFFFF = reserved
enum8	**sensorDataSize** The bit width and format for the thresholds that are set in the sensor value: { uint8, sint8, uint16, sint16, uint32, sint32 } NOTE: This value is used for checking purposes only. A sensor accepts only one particular data format. The sensor data size must be known a priori; it can be obtained from a PDR for the sensor or by issuing a GetSensorThresholds command. Values for all six threshold parameters must be provided regardless of which thresholds are supported. If a particular threshold is not supported by the sensor, the value passed in the corresponding parameter is ignored.
For sensorDataSize = uint8 or sint8	
uint8 | sint8	**upperThresholdWarning**
uint8 | sint8	**upperThresholdCritical**
uint8 | sint8	**upperThresholdFatal**
uint8 | sint8	**lowerThresholdWarning**
uint8 | sint8	**lowerThresholdCritical**
uint8 | sint8	**lowerThresholdFatal**
For sensorDataSize = uint16 or sint16	
uint16 | sint16	**upperThresholdWarning**
uint16 | sint16	**upperThresholdCritical**
uint16 | sint16	**upperThresholdFatal**
uint16 | sint16	**lowerThresholdWarning**
uint16 | sint16	**lowerThresholdCritical**
uint16 | sint16	**lowerThresholdFatal**

Type	Request Data
For sensorDataSize = uint32 or sint32	
uint32 | sint32	**upperThresholdWarning**
uint32 | sint32	**upperThresholdCritical**
uint32 | sint32	**upperThresholdFatal**
uint32 | sint32	**lowerThresholdWarning**
uint32 | sint32	**lowerThresholdCritical**
uint32 | sint32	**lowerThresholdFatal**
Type	**Response Data**
	completionCode value: { PLDM_BASE_CODES, INVALID_SENSOR_ID=0x80 }

18.5 RestoreSensorThresholds Command

The RestoreSensorThresholds command restores default thresholds for the device. Table 22 describes the format of the command.

Table 22 – RestoreSensorThresholds Command Format

Type	Request Data
uint16	**sensorID** A handle that is used to identify and access the sensor special values: 0x0000, 0xFFFF = reserved
Type	**Response Data**
enum8	**completionCode** value: { PLDM_BASE_CODES, INVALID_SENSOR_ID=0x80 }

18.6 GetSensorHysteresis Command

The GetSensorHysteresis command is used to read the present hysteresis setting for a PLDM Numeric Sensor. The hysteresis value uses the same units, data size, and conversion factors that are specified for the reading from the sensor. Table 23 describes the format of the command.

Table 23 – GetSensorHysteresis Command Format

Type	Request Data
uint16	**sensorID** A handle that is used to identify and access the sensor special values: 0x0000, 0xFFFF = reserved

Type	Response Data
enum8	**completionCode** value: { PLDM_BASE_CODES, INVALID_SENSOR_ID=0x80 }
enum8	**sensorDataSize** The bit width of the hysteresis value that is being returned value: { uint8, sint8, uint16, sint16, uint32, sint32 }
For sensorDataSize = uint8 or sint8	
uint8 | sint8	**hysteresis value**
For sensorDataSize = uint16 or sint16	
uint16 | sint16	**hysteresis value**
For sensorDataSize = uint32 or sint32	
uint32 | sint32	**hysteresis value**

18.7 SetSensorHysteresis Command

The SetSensorHysteresis command is used to set the present hysteresis setting for a PLDM Numeric Sensor. The hysteresis value uses the same units, data size, and conversion factors that are specified for the reading from the sensor. It is recommended that the sensor be disabled while changing the hysteresis setting. Table 24 describes the format of the command.

Table 24 – SetSensorHysteresis Command Format

Type	Request Data
uint16	**sensorID** A handle that is used to identify and access the sensor special values: 0x0000, 0xFFFF = reserved
enum8	**sensorDataSize** The bit width and format for the following hysteresis value that is being set into the sensor value: { uint8, sint8, uint16, sint16, uint32, sint32 } NOTE: This value is used for checking purposes only. A sensor accepts only one particular data format. The sensor data size must be known a priori; it can be obtained from a PDR for the sensor or by issuing a GetSensorHysteresis command.
For sensorDataSize = uint8 or sint8	
uint8 | sint8	**hysteresis value**
For sensorDataSize = uint16 or sint16	
uint16 | sint16	**hysteresis value**
For sensorDataSize = uint32 or sint32	
uint32 | sint32	**hysteresis value**
Type	**Response Data**
enum8	**completionCode** value: { PLDM_BASE_CODES, INVALID_SENSOR_ID=0x80 }

18.8 InitNumericSensor Command

The InitNumericSensor command is typically used by the Initialization Agent function (see section 15) to initialize PLDM Numeric Sensors. The command may also be used as an interface for “virtual sensors,” which do not actually poll and update their own state but instead rely on another management controller or system software to set their state.

Implementations should avoid virtual sensors that require initialization by the Initialization Agent function. Conflicts could occur if the sensor needs to be accessed by the Initialization Agent function at the same time it is being accessed as a virtual sensor. Typically, however, a virtual sensor would not require initialization by the Initialization Agent function.

Table 25 describes the format of the command.

Table 25 – InitNumericSensor Command Format

Type	Request Data
uint16	**sensorID** A handle that is used to identify and access the sensor special values: 0x0000, 0xFFFF = reserved
enum8	**sensorOperationalState** The expected operational state of the sensor. This enumeration is a subset of the operational state values that are returned by the GetSensorReading command. Refer to the GetSensorReading command for the definition of the values in this enumeration. This parameter is applied to the sensor *after* all other fields (sensorPresentState, eventMsgEnable, and numericReadingSetting) have been applied to the sensor. value: { enabled, disabled, unavailable }
enum8	**sensorPresentState** The expected present state of the numeric sensor. See the description of the presentState field in Table 19.

enum8	**eventMsgEnable**
	This value is used to enable or disable event message generation from the sensor.
	value: {
	enableEventMessages,
	disableEventMessages,
	noChange=0xFF // Do not alter the present event enable setting.
	}

bool8	**setNumericReading** value: { false, true } True directs the receiver to accept the following numericReadingSetting.
var	**numericReadingSetting** The size of this field depends on the sensor data size. This value is used as the initial value for the presentReading returned by the numeric sensor. Some sensor implementations may ignore this value if it is given.

Type	Response Data
enum8	**completionCode** value: { PLDM_BASE_CODES, INVALID_SENSOR_ID=0x80 }

19 PLDM State Sensors

PLDM State Sensors are used to return a status from one or more state sets. A state set is simply the name of an enumeration that is a collection of a set of related platform states. Common state sets are defined in DSP0249.

A PLDM State Sensor that returns values from only a single state set is referred to as a simple state sensor. A state sensor that returns values from more than one state set is referred to as a composite state sensor.

This specification also includes support for the definition of vendor-specific state sets using the OEM State Set PDR. (See 28.10 for more information.)

20 PLDM State Sensor Commands

This section describes the commands for accessing PLDM State Sensors per this specification. The command numbers for the PLDM messages are given in section 30.

If PLDM State Sensors are implemented, the Mandatory/Conditional (M/C) requirements shown in Table 26 apply.

Table 26 – State Sensor Commands

Command	M/C	Reference
SetStateSensorEnables	M	See 20.1.
GetStateSensorReadings	M	See 20.2.
InitStateSensor	C [1]	See 20.3.

[1] Required for sensors that are to be initialized through the Initialization Agent function.

20.1 SetStateSensorEnables Command

The SetStateSensorEnables command is used to set enable or disable sensor operation and event message generation for sensors within a PLDM Composite State Sensor. Event message generation is optional for a sensor. Table 27 describes the format of the command.

Table 27 – SetStateSensorEnables Command Format

Type	Request Data
uint16	**sensorID** A handle that is used to identify and access the sensor special values: 0x0000, 0xFFFF = reserved

Type	Request Data
uint8	**compositeSensorCount** The number of individual sets of sensor information that this command accesses. Up to eight sets of state sensor information (referred to as sensors 1 through 8) can be accessed through a given sensorID within a PLDM terminus. value: 0x01 to 0x08
opField xN	**opFields** Each opField is an instance of an opField structure that is used to set the present operational state setting and event message enables for a particular sensor within the state sensor. The opField structure is defined in Table 28.
Type	**Response Data**
enum8	**completionCode** value: { PLDM_BASE_CODES, INVALID_SENSOR_ID=0x80, EVENT_GENERATION_NOT_SUPPORTED = 0x82 }

Table 28 – SetStateSensorEnables opField Format

Type	Description
enum8	**sensorOperationalState** The expected state of the sensor This enumeration is a subset of the operational state values that are returned by the GetStateSensorReading command. Refer to the GetStateSensorReading command for the definition of the values in this enumeration. value: { enabled, disabled, unavailable }
enum8	**eventMessageEnable** This value is used to enable or disable event message generation from the sensor. value: { noChange, disableEvents, enableEvents, enableOpEventsOnly, enableStateEventsOnly } noChange means do not alter the present setting. Use noChange when the sensor does not support event message generation. NOTE: Event message generation is optional for a sensor.

20.2 GetStateSensorReadings Command

The GetStateSensorReadings command can return readings for multiple state sensors (a PLDM State Sensor that returns more than one set of state information is called a composite state sensor).

State information is returned as a sequence of one to N "stateField" structures. The first stateField structure is referred to as the structure for the sensor at offset 0, second is for the sensor at offset 1, and so on.

The same number of stateField structures must be returned and in the same sequence during platform management subsystem operation, regardless of the operational status of the sensors.

Table 29 describes the format of the command.

1708 **Table 29 – GetStateSensorReadings Command Format**

Type	Request Data
uint16	**sensorID** A handle that is used to identify and access the simple or composite sensor special values: 0x00, 0xFFFF = reserved
bitfield16	**sensorRearm** Each bit location in this field corresponds to a particular sensor within the state sensor, where bit [0] corresponds to the first state sensor (sensor 1) and bit [7] corresponds to the eighth sensor (sensor 8), sequentially. For each bit position [n] from n = 0 to compositeSensorCount-1, the bit setting operates as follows: 0b = do not re-arm sensor [n]+1 1b = re-arm sensor [n]+1 Bit positions that are greater than [compositeSensorCount-1], if any, shall be written as "0b".
Type	**Response Data**
	completionCode value: { PLDM_BASE_CODES, INVALID_SENSOR_ID=0x80 }
unit8	**compositeSensorCount** The number of individual sets of sensor information that this command accesses. Up to eight sets of state sensor information (referred to as sensors 1 through 8) can be accessed through a given sensorID within a PLDM terminus. value: 0x01 to 0x08
stateField xN	**stateFields** Each stateField is an instance of a stateField structure that is used to return the present operational state setting and the present status and event status for a particular set of sensor information contained within the state sensor. The stateField structure is defined in Table 30.

1709 **Table 30 – GetStateSensorReadings stateField Format**

Type	Description
enum8	**sensorOperationalState** The state of the sensor itself value: { enabled, disabled, initializing, shuttingDown, unavailable, failed, inTest }
enum8	**presentState** This field is used to return a state value from a PLDM State Set that is associated with the sensor. The value reflects the most recently assessed state.
enum8	**previousState** The eventState value for the state from which the present state was entered. special value: This value shall be set to the same value as presentState if the previousState is unknown, which might be the case for events that are generated on the first status assessment that occurs after a sensor has been initialized.
enum8	**eventState** This field is used to return a state value from a PLDM State Set that is associated with the sensor. The value reflects the most recently assessed state that caused an event to be generated.

20.3 InitStateSensor Command

The InitStateSensor command is typically used by the Initialization Agent function (see section 15) to initialize PLDM State Sensors. The command may also be used as an interface for virtual sensors, which do not actually poll and update their own state but instead rely on another management controller or system software to set their state.

Implementations should avoid virtual sensors that require initialization by the Initialization Agent function. Conflicts could occur if the sensor needs to be accessed by the Initialization Agent function at same time it is being accessed as a virtual sensor. Typically, however, a virtual sensor would not require initialization by the Initialization Agent function.

Table 31 describes the format of the command.

Table 31 – InitStateSensor Command Format

Type	Request Data
uint16	**sensorID** A handle that is used to identify and access the sensor special values: 0x0000, 0xFFFF = reserved
unit8	**compositeSensorCount** The number of individual sets of sensor information that this command accesses. Up to eight sets of state sensor information (referred to as sensors 1 through 8) can be accessed through a given sensorID within a PLDM terminus. value: 0x01 to 0x08
initField xN	Each initField is an instance of an initField structure that is used to set the present operational state setting and event message enables for a particular sensor within the state sensor. The initField structure is defined in Table 32.
Type	**Response Data**

enum8	**completionCode**
	value: { PLDM_BASE_CODES,
	INVALID_SENSOR_ID = 0x80,
	UNSUPPORTED_SENSORSTATE = 0x81 // an illegal value was submitted for sensorOperationState or sensorPresentState for one or more sensors
	}

Table 32 – InitStateSensor initField Format

Type	Description
enum8	**sensorOperationalState** The expected operational state of the sensor. This enumeration is a subset of the operational state values that are returned by the GetSensorReading command. Refer to 18.2 for the definition of the values in this enumeration. This parameter is applied to the sensor after all other fields (sensorPresentState and eventMsgEnable) have been applied to the sensor. value: { enabled, disabled, unavailable }

Type	Description
enum8	**sensorPresentState** The expected state of the sensor. The state values are based on the particular state set used for the sensor. The set of states that the sensor can be initialized with may be a subset of the states that the sensor reports while monitoring. value: { dependent on sensor State Set }
enum8	**eventMsgEnable** This value is used to enable or disable event message generation from the sensor. value: { enableEvents, disableEvents, noChange=0xFF } noChange means do not alter the present setting.

21 PLDM Effecters

PLDM effecters provide a general mechanism for controlling or configuring a state or numeric setting of an entity. PLDM effecters are similar to PLDM sensors, except that entity state and numeric setting values are written into an effecter rather than read from it.

PLDM commands are specified for writing the state or numeric setting to an effecter. Effecters are identified by and accessed using an EffecterID that is unique for each effecter within a given terminus. Corresponding PDRs provide basic semantic information for effecters, such as what type of states or numeric units the effecter accepts, what terminus and EffecterID value are used to access the effecter, which entity the effecter is associated with, and so on.

21.1 PLDM State Effecters

PLDM State Effecters provide a regular command structure for setting state information in order to change the state of an entity. Effecters use the same PLDM State Sets definitions as PLDM State Sensors, but instead of using the state set information to interpret the value that is read from a sensor, the state sets are used to define the value to write to an effecter. Like PLDM Composite State Sensors, PLDM State Effecters can be implemented and accessed as composite state effecters where a single EffecterID is used to access a set of state effecters. This enables multiple states to be set using a single command and to share a single PDR that provides the basic information for the effecters.

21.2 PLDM Numeric Effecters

PLDM Numeric Effecters provide a regular command structure for setting a numeric value for a controllable parameter of an entity. Numeric effecters use the same definition of units as the units for readings returned by numeric sensors (see 27.2). For example, a numeric effecter could be used to set a value for revolutions per second.

21.3 Effecter Semantics

An effecter has a meaning or use that is associated with what an effecter does or is used for. This will be referred to as the "effecter semantic", or just the "semantic."

Although PLDM effecters provide a straightforward mechanism for setting a state or numeric value for an entity, conveying the semantic of how that state or numeric value affects the entity, or how the setting should be used, is not always straightforward.

Suppose a numeric effecter is defined for setting a fan speed. A PDR for the numeric effecter can readily indicate that the effecter is for "Physical Fan 1", and that "Fan 1" is contained by Processor 1. The PDR can also indicate that the units for the setting are "RPM". However, this does not convey what the RPM is actually doing. For example, is the RPM a speed limit or a target speed?

Additionally, other information may be necessary for understanding how the effecter is to be used. If a fan speed needs to be set because one or more temperatures have become too high, how does the user of PLDM know which temperatures are associated with the fan, and what RPM value should be set for a particular temperature?

The information required to describe the meaning and use of an effecter can vary significantly depending on how generic or specific the use is to the platform implementation. The level of generality of effecter semantics in PLDM is categorized as shown in Table 33.

Table 33 – Categories for Effecter Semantics

Category	Description
By State Set or Units Only	The definition of the state set or numeric units, along with the Entity Association Information provided through the effecter PDRs, is sufficient to convey the semantic for the effecter. For example, the state set for System Power State when combined with "System" as the containerID identifies an effecter for overall system power control.
By Semantic ID	The state sets or units definitions and entity associations alone are not sufficient to identify the semantic of the effecter, but the effecter use can be indicated by providing a single "Semantic ID" value that identifies a predefined semantic for the effecter. For example, a Semantic ID could be defined for "System Power Down with Delay" where the definition specifies that the effecter accepts a time value that identifies a delay from 1 to 60 seconds and triggers a system power down after that delay when the effecter value gets set. This specification makes provision for DMTF PLDM defined or OEM (vendor-defined) Semantic IDs. See 21.4 for more information.
By Semantic ID plus PDRs	The effecter PDR information and the Semantic ID are not sufficient to identify the semantic of the effecter, but the semantic can be communicated when the Semantic ID is used with other PDRs. For example, an effecter could be defined for setting a "Fan speed override" where the fan speed is set to a "boost mode" if one or more temperature sensors in the system exceed their critical thresholds. One or more additional PDRs would be used to identify which temperature sensors in the particular platform would contribute to boost mode. Note that in this case the effecter itself is not implementing this policy. A third party, such as a MAP, would read the PDR information and use that information to know when it should change the effecter's setting.
External Information Required	The effecter semantic may not be described using the mechanisms offered by this specification. In some cases, use of the effecter may require access to information that is not provided through PDRs–for example, an effecter where the user (such as a MAP) requires access to SMBIOS data to understand how the effecter should be used. In other cases, the effecter semantic may have a private or proprietary where the effecter is implemented using PLDM commands and described in the PDRs only because the implementation wants to reuse the command infrastructure from this specification or take advantage of functions such as the Initialization Agent or Event Log.

The most generic and efficient use of effecters comes when they fall into the state sets or units only category and use standard state set or units definitions. The second most generic and efficient use of effecters is when they use a standard defined Semantic ID. Thus, if new standard effecter semantics need to be defined, it should be first examined whether a new state set or units definition should be added to the specifications, or whether a new Semantic ID should be added.

21.4 PLDM and OEM Effecter Semantic IDs

Effecter Semantic ID values are specified in DSP0249. A range of values is reserved for definition by the DMTF PLDM specifications and another range of values is available for OEM (vendor defined) effecter semantics. When the OEM range is used, the semantic is identified and optionally named using an OEM Effecter Semantic PDR. The use of the OEM Effecter Semantic PDR is similar to how OEM units, entities, and state sets are defined within the PDRs.

22 PLDM Effecter Commands

This section describes the commands for accessing PLDM effecters per this specification. The command numbers for the PLDM messages are given in section 30.

If PLDM Numeric Effecters or PLDM State Effecters are implemented, the Mandatory (M) requirements shown in Table 34 apply.

Table 34 – State and Numeric Effecter Commands

Command	M	Reference
SetNumericEffecterEnable	M [1]	See 22.1.
SetNumericEffecterValue	M [1]	See 22.2.
GetNumericEffecterValue	M [1]	See 22.3.
SetStateEffecterEnables	M [2]	See 22.4.
SetStateEffecterStates	M [2]	See 22.5.
GetStateEffecterStates	M [2]	See 22.6.

[1] Required if one of more numeric effecters are implemented

[2] Required if one or more state effecters are implemented

22.1 SetNumericEffecterEnable Command

The SetNumericEffecterEnable command is used to enable or disable effecter operation. A disabled effecter cannot have its state updated. An effecter may have a default state that it automatically returns to when it is disabled. An effecter may also be able to be returned to its default state through the SetStateNumericEffecterValue command. The PLDM Numeric Effecter PDR can describe a numeric effecter and whether it has a default state.

Table 35 describes the format of this command.

Table 35 – SetNumericEffecterEnable Command Format

Type	Request Data
uint16	**effecterID** A handle that is used to identify and access the effecter special values: 0x0000, 0xFFFF = reserved
enum8	**effecterOperationalState** The expected state of the effecter. This enumeration is a subset of the operational state values that are returned by the GetStateEffecterStates command. Refer to the GetStateEffecterStates command for the definition of the values in this enumeration. value: { enabled, disabled = 2, unavailable }
Type	**Response Data**
enum8	**completionCode** value: { PLDM_BASE_CODES, INVALID_EFFECTER_ID=0x80 }

22.2 SetNumericEffecterValue Command

The SetNumericEffecterValue command is used to set the value for a PLDM Numeric Effecter. Table 36 describes the format of this command.

Table 36 – SetNumericEffecterValue Command Format

Type	Request Data
uint16	**effecterID** A handle that is used to identify and access the effecter special values: 0x0000, 0xFFFF = reserved
enum8	**effecterDataSize** The bit width and format of the setting value for the effecter value: { uint8, sint8, uint16, sint16, uint32, sint32 } NOTE: This value does not select a data size that is to be accepted by the effecter. The value is used only to enable the responder to confirm that the effecterValue is being given in the expected format.
uint8 | sint8 | uint16 | sint16 | sint32 | uint32	**effecterValue** The setting value of numeric effecter being requested
Type	**Response Data**
enum8	**completionCode** value: { PLDM_BASE_CODES, INVALID_EFFECTER_ID=0x80, }

22.3 GetNumericEffecterValue Command

The GetNumericEffecterValue command is used to return the present numeric setting of a PLDM Numeric Effecter. Table 37 describes the format of this command.

Table 37 – GetNumericEffecterValue Command Format

Type	Request Data
uint16	**effecterID** A handle that is used to identify and access the effecter special values: 0x0000, 0xFFFF = reserved
Type	**Response Data**
enum8	**completionCode** value: { PLDM_BASE_CODES, INVALID_EFFECTER_ID=0x80 }
enum8	**effecterDataSize** The bit width and format of the setting value for the effecter value: { uint8, sint8, uint16, sint16, uint32, sint32 }
enum8	**effecterOperationalState** The state of the effecter itself value: { enabled-updatePending, enabled-noUpdatePending, disabled, unavailable, statusUnknown, failed, initializing, shuttingDown, inTest } enabled-updatePending = Enabled and operating. The effecter is able to return valid setting values. The setting of the numeric effecter is in the process of being changed to the pending value. enabled-noUpdatePending = Enabled and operating. The effecter is able to return valid setting values. The pending and presentValue fields return the present numeric setting of the effecter. The pendingValue and presentValue fields may not be valid and should be ignored when the effecter is in any of the following states. The implementation is not required to return any particular values for the pendingValue or presentValue fields in these states. disabled: The effecter is disabled from returning presentReading and event status values. This state is set through the SetNumericEffecterEnable command. unavailable: The effecter should be ignored due to configuration of the platform or monitored entity. For example, the effecter is for monitoring a processor temperature, but the processor is not installed. This state is set through the SetNumericEffecterEnable command. statusUnknown: The effecter cannot presently return valid reading information for the monitored entity. failed: The effecter has failed. The effecter implementation has determined that it cannot return correct values for its present setting. initializing: The effecter is in the process of transitioning to the operating state because the effecter has been initialized (starting) or reinitialized. The presentState and eventState values shall be ignored while the effecter is in this state. shuttingDown: The effecter is transitioning to the disabled, failed, or unavailable state. inTest: The effecter is presently undergoing testing. NOTE: The operation of effecter testing and the mechanisms for effecter testing are outside the scope of this specification.

Type	Response Data
uint8 | sint8 | uint16 | sint16 | sint32 | uint32	**pendingValue** The pending numeric value setting of the effecter. The effecterDataSize field indicates the number of bits used for this field.
uint8 | sint8 | uint16 | sint16 | sint32 | uint32	**presentValue** The present numeric value setting of the effecter. The effecterDataSize indicates the number of bits used for this field.

22.4 SetStateEffecterEnables Command

The SetStateEffecterEnables command is used to enable or disable effecter operation. A disabled effecter cannot have its state updated. An effecter may have a default state that it automatically returns to when it is disabled. An effecter may also be able to be returned to its default state through the SetStateEffecterStates command. The PLDM State Effecter PDR describes a state effecter and whether it has a default state. Table 38 describes the format of this command.

Table 38 – SetStateEffecterEnables Command Format

Type	Request Data
uint16	**effecterID** A handle that is used to identify and access the effecter special values: 0x0000, 0xFFFF = reserved
uint8	**compositeEffecterCount** The number of individual sets of state effecter information that are accessed by this command. Up to eight sets of effecter information (referred to as effecters 1 through 8) can be accessed through a given effecterID within a PLDM terminus. value: 0x01 to 0x08
opField xN	**opFields** Each opField is an instance of an opField structure that is used to set the present operational state setting and event message enables for a particular sensor within the state effecter. The opField structure is defined in Table 39.
Type	**Response Data**
enum8	**completionCode** value: { PLDM_BASE_CODES, INVALID_EFFECTER_ID=0x80 }

Table 39 – SetStateEffecterEnables opField Format

Type	Description
enum8	**effecterOperationalState** The expected state of the effecter. This enumeration is a subset of the operational state values that are returned by the GetStateEffecterStates command. Refer to the GetStateEffecterStates command for the definition of the values in this enumeration. value: { enabled, disabled=2, unavailable }
enum8	**eventMsgEnable** This value is used to enable or disable event message generation from the effecter. value: { enableEvents, disableEvents, noChange=0xFF } noChange means do not alter the present setting.

22.5 SetStateEffecterStates Command

The SetStateEffecterStates command is used to set the state of one or more effecters within a PLDM State Effecter. Table 40 describes the format of this command.

Table 40 – SetStateEffecterStates Command Format

Type	Request Data
uint16	**effecterID** A handle that is used to identify and access the effecter special values: 0x0000, 0xFFFF = reserved
unit8	**compositeEffecterCount** The number of individual sets of effecter information that are accessed by this command. Up to eight sets of state effecter information (referred to as effecters 1 through 8) can be accessed through a given effecterID within a PLDM terminus. value: 0x01 to 0x08
stateField xN	Each stateField is an instance of a stateField structure that is used to set the requested state for a particular effecter within the state effecter. The stateField structure is defined in Table 41.
Type	**Response Data**

enum8	**completionCode**
	value: { PLDM_BASE_CODES, INVALID_EFFECTER_ID=0x80,
	INVALID_STATE_VALUE=0x81,
	UNSUPPORTED_SENSORSTATE = 0x82 // An illegal value was submitted for sensorOperationState or sensorPresentState for one or more sensors.
	}

Table 41 – SetStateEffecterStates stateField Format

Type	Description

enum8	**setRequest**
	value: {
	noChange, // Do not request a change of the state of this effecter.
	requestSet // Request the effecter state to be set to the state given by the following // effecterState value.
	}

enum8	**effecterState** The expected state of the effecter. The state values come from the particular state set used for the implementation of the effecter. value: { dependent on effecter state set }

22.6 GetStateEffecterStates Command

The GetStateEffecterStates command is used to get the present status of an effecter. Table 42 describes the format of this command.

Table 42 – GetStateEffecterStates Command Format

Type	Request Data
uint16	**effecterID** A handle that is used to identify and access the simple or composite effecter special values: 0x0000, 0xFFFF = reserved
Type	**Response Data**
	completionCode value: { PLDM_BASE_CODES, INVALID_EFFECTER_ID=0x80 }
unit8	**compositeEffecterCount** The number of individual sets of effecter information that are accessed by this command. Up to eight sets of state effecter information (referred to as effecters 1 through 8) can be accessed through a given effecterID within a PLDM terminus. value: 0x01 to 0x08
stateField xN	**stateFields** Each stateField is an instance of a stateField structure that is used to return the present operational state setting and the present status for a particular effecter contained within the state effecter. The stateField structure is defined in Table 43.

Table 43 – GetStateEffecterStates stateField Format

Type	Description
enum8	**effecterOperationalState** The state of the effecter itself value: { enabled-updatePending, enabled-noUpdatePending, disabled, unavailable, failed, inTest }
enum8	**pendingState** If the value of effecterOperationalState is updatePending, this field returns the value for the requested state that is presently being processed. Otherwise, this field returns the present state of the effecter. The effecter implementation should return the "unknown" state value whenever the effecterOperationalState is anything except enabled-updatePending or enabled-noUpdatePending. Parties that are accessing this information should also ignore this field (treat it as unknown) when the effecterOperationalState is anything except enabled-updatePending or enabled-noUpdatePending. value: { dependent on effecter state set on which the effecter implementation is based }
enum8	**presentState** The present state of the effecter. The effecter implementation should return the "unknown" state value whenever the value of effecterOperationalState is anything except enabled-updatePending or enabled-noUpdatePending. Parties that are accessing this information should also ignore this field (treat it as unknown) when the effecterOperationalState is anything except enabled-updatePending or enabled-noUpdatePending. value: { dependent on the state set used for the effecter implementation }

23 PLDM Event Log Commands

This section describes the commands for accessing a PLDM Event Log per this specification. The command numbers for the PLDM messages are given in section 30.

The PLDM Event Log is typically accessed through the same PLDM terminus as the Event Receiver. However, this is not mandatory. The PDRs include information that describes which terminus is used to access the PLDM Event Log. The command numbers for PLDM commands are listed in section 29.

If a PLDM Event Log is implemented, the Mandatory/Optional/Conditional (M/O/C) requirements shown in Table 44 apply.

Table 44 – PLDM Event Log Commands

Command	M/O/C	Reference
GetPLDMEventLogInfo	M	See 23.1.
EnablePLDMEventLogging	M	See 23.2.
ClearPLDMEventLog	M	See 23.3.
GetPLDMEventLogTimestamp	M	See 23.4.
SetPLDMEventLogTimestamp	M	See 23.5.
ReadPLDMEventLog	M	See 23.6.
GetPLDMEventLogPolicyInfo	M	See 23.7.
SetPLDMEventLogPolicy	C [1]	See 23.8.
FindPLDMEventLogEntry	O	See 23.9

[1] Required if the PLDMEventLog implementation supports configurable policy parameters

23.1 GetPLDMEventLogInfo Command

The GetPLDMEventLogInfo command returns basic information about the PLDM Event Log, such as its operational status, percentage used, and time stamps for the most recent add and erase actions. Table 45 describes the format of the command.

Table 45 – GetPLDMEventLogInfo Command Format

Type	Request Data
–	none
Type	**Response Data**
enum8	**completionCode** value: { PLDM_BASE_CODES }

enum8	**logOperationalStatus**
	value: {
	loggingDisabled, // Log can be accessed, but is disabled from accepting entries.
	enabledReady, // Log can be accessed and is enabled to accept entries.
	clearInProgress, // Log is enabled but log information and entries are unable to be // accessed because the log is in the process of being cleared.
	enabledFull, // Log is enabled but cannot accept more entries because it is // full. The log shall automatically resume accepting entries once // entries are cleared. It is not necessary to explicitly re-enable // logging.
	failedLoggingDisabled, // Log has had a failure where it can no longer accept entries. // Clearing and re-enabling logging must restore the log to // normal operation. If this cannot occur, the 'failedDisabled' // logOperationalStatus value shall be returned.
	failedDisabled, // Log has had a failure where it is unable to // accept entries. Additionally, existing entries may not be able // to be accessed successfully. The log may or may not be able // to be restored to normal operation by clearing and re-enabling // the log.
	corrupted // Some or all log data has been lost due to a data corruption. // Clearing the log and re-enabling logging shall restore internal // integrity. If this cannot be done, the implementation shall // return a logOperationalStatus of failedLoggingDisabled or // failedDisabled. The log implementation shall not return records // that are known to be corrupted.
	}

enum8	**activeLogClearingPolicy** The log clearing policy that is presently in effect for this PLDM Event Log. See 13.4 for a description of the log clearing policies. value: { fillAndStop, FIFO, clearOnAge }
Type	**Response Data**
uint32	**entryCount** number of entries presently in the Event Log

uint8	**storagePercentUsed** The percentage of log storage space presently used up by entries in the log, given in increments based on the percentUsedResolution parameter from the PLDM Event Log PDR value: 0 to 100 special value: 0xFF = unspecified
uint8	**percentWear** The implementation may elect to return this value as an indication of the present level of wear on the storage medium. Values 0 to 100 indicate an estimated percentage of normal rated lifetime or storage cycles used up on the device. Values greater than 100 indicate levels that have exceeded the rated or expected lifetime. The mechanism and algorithms that are used for returning this parameter are implementation specific and outside the scope of this specification. value: 0x00 to 0x064 = wear in % special value: 0xFF = unspecified
mostRecentAddTimestamp The following three fields return the timestamp of the most recent addition or change to the log. The implementation must automatically adjust the mostRecentAddTimestamp whenever the Event Log timestamp clock is set using the SetPLDMEventLogTimestamp command. See the description of the SetPLDMEventLogTimestamp command for more information. special value: The implementation may choose to retain the mostRecentAddTimestamp value after the log has been cleared, or it may elect to set the value to the 'unspecified' value for the data type. The unspecified value shall only be used when the log is empty (cleared), or if the timestamp has been lost due to an error or firmware update condition.	
sint8	**mostRecentAddTimestampUTCOffset** The UTC offset for the log entry timestamp in increments of 1/2 hour. special value: 0xFF = unspecified
uint40	**mostRecentAddTimestampSeconds** This value corresponds to a 40-bit unsigned integer representing the number of seconds since midnight UTC of January 1, 1970 (not counting leap seconds). 0x0000000000 = unspecified.
uint8	**mostRecentAddTimestamp100s** This value provides a number of 1/100ths of a second added to **entryTimestampSeconds**. value: 0 to 99. special value: 0xFF = unspecified. This value is used if the implementation timestamps entries to no finer than a one second resolution.
mostRecentEraseTimestamp The following three fields return the most recent time that entries were deleted from the log or the log was cleared. The implementation must automatically adjust the mostRecentEraseTimestamp whenever the Event Log timestamp clock is set using the SetPLDMEventLogTimestamp command. See the description of the SetPLDMEventLogTimestamp command for more information. special value: The implementation may choose to retain the mostRecentAddTimestamp value after the log has been cleared, or it may elect to set the value to the 'unspecified' value for thedata type. The unspecified value shall only be used if the timestamp has never been initialized, or if the timestamp has been lost due to an error or firmware update condition.	
sint8	**mostRecentEraseTimestampUTCOffset** The UTC offset for the log entry timestamp in increments of 1/2 hour. special value: 0xFF = unspecified

uint40	**mostRecentEraseTimestampSeconds** This value corresponds to a 40-bit unsigned integer representing the number of seconds since midnight UTC of January 1, 1970 (not counting leap seconds). 0x0000000000 = unspecified.
uint8	**mostRecentEraseTimestamp100s** This value provides a number of 1/100ths of a second added to **entryTimestampSeconds**. value: 0 to 99. special value: 0xFF = unspecified. This value is used if the implementation timestamps entries to no finer than a one second resolution.

23.2 EnablePLDMEventLogging Command

The EnablePLDMEventLogging command is used to enable or disable the PLDM Event log from logging events. The log can be accessed and cleared while in the disabled state unless the logOperationalStatus is “failed”, in which case logging may not be able to be enabled. Table 46 describes the format of the command.

Table 46 – EnablePLDMEventLogging Command Format

Type	Request Data

enum8	**enableLogging**
	value: {
	disableLogging, // Disable accepting events into the log.
	enableLogging // Enable logging events.
	}

Type	**Response Data**
enum8	**completionCode** value: { PLDM_BASE_CODES }
enum8	**logOperationalStatus** value: { See the definition of logOperationalStatus field for the GetPLDMEventLogInfo command (Table 45). }

23.3 ClearPLDMEventLog Command

The ClearPLDMEventLog command is used to clear the contents of the PLDM Event Log. The execution of this command does not affect whether logging is enabled or disabled. Depending on the subsystem and its implementation, it is possible that events may be received or be in the process of being received during the terminus' execution of this command. If event logging is enabled, a terminus should continue to accept events while it is processing this command. It is recognized that in some implementations clearing the log device may take a significant amount of time. The number of events that an implementation may support queuing up while the log is being cleared is implementation dependent. Table 47 describes the format of this command.

Table 47 – ClearPLDMEventLog Command Format

Type	Request Data
–	none

Type	Response Data
enum8	**completionCode** value: { PLDM_BASE_CODES }
enum8	**logOperationalStatus** The status of the log following acceptance of this command. This status will typically be clearInProgress, enabledReady, or loggingDisabled, depending on the implementation. value: { See the definition of logOperationalStatus for the GetPLDMEventLogInfo command (Table 48). }

1846 ## 23.4 GetPLDMEventLogTimestamp Command

1847 The GetPLDMEventLogTimestamp command returns a snapshot of the present PLDM Event Log
1848 Timestamp time. Table 48 describes the format of this command.

1849 **Table 48 – GetPLDMEventLogTimestamp Command Format**

Type	Request Data
–	none
Type	**Response Data**
enum8	**completionCode** value: { PLDM_BASE_CODES }
sint8	**entryTimestampUTCOffset** The UTC offset for the log entry time stamp in increments of 1/2 hour special value: 0xFF = unspecified
uint40	**entryTimestampSeconds** This value corresponds to a 40-bit unsigned integer that represents the number of seconds since midnight UTC of January 1, 1970 (not counting leap seconds).
uint8	**entryTimestamp100s** This value provides a number of 1/100 of a second that is added to **entryTimestampSeconds**. value: 0 to 99 special value: 0xFF = unspecified. This value is used if the implementation timestamps entries to no finer than a one second resolution.

23.5 SetPLDMEventLogTimestamp Command

The SetPLDMEventLogTimestamp command can be used to set the PLDM Event Log Timestamp time.

Some implementations may not implement the ability to set the time stamp to 1/100 of a second resolution and will round the time up or down to match the resolution that it supports. Therefore, the time stamp value in the response may vary from what was submitted because of rounding. The returned value may also vary due to delays in command response processing within the terminus.

Implementations are required to support a 1 second or finer resolution for the time stamp. Table 49 describes the format of this command.

Table 49 – SetPLDMEventLogTimestamp Command Format

Type	Request Data
sint8	**entryTimestampUTCOffset** The UTC offset for the log entry time stamp in increments of 1/2 hour special value: 0xFF = unspecified
uint40	**entryTimestampSeconds** This value corresponds to a 40-bit unsigned integer that represents the number of seconds since midnight UTC of January 1, 1970 (not counting leap seconds).
uint8	**entryTimestamp100s** This value provides a number of 1/100 of a second that is added to **entryTimestampSeconds**. value: 0 to 99 This value is ignored if the implementation only timestamps entries to a one second resolution.

enum8	**logUpdateEvent**
	value: {
	noEvent,
	logEvent // automatically logs a time stamp change event if the new time stamp clock // value is accepted. See DSP0249 for the state set definition for time // stamp change events.
	}

Type	Response Data
enum8	**completionCode** value: { PLDM_BASE_CODES }
sint8	**entryTimestampUTCOffset** The UTC offset for the log entry time stamp in increments of 1/2 hour special value: 0xFF = unspecified
uint40	**entryTimestampSeconds** This value corresponds to a 40-bit unsigned integer that represents the number of seconds since midnight UTC of January 1, 1970 (not counting leap seconds).

uint8	**entryTimestamp100s** This value provides a number of 1/100 of a second that is added to **entryTimestampSeconds**. value: 0 to 99 special value: 0xFF = unspecified. This value is used if the implementation timestamps entries to no finer than a one second resolution.
uint8	**timestampResolution** The resolution of the time stamp that is kept by the implementation in 1/100 of a second. value: 1 to 100 (100 = 1 second resolution, 5 = .05 seconds resolution, and so on)

23.6 ReadPLDMEventLog Command

The ReadPLDMEventLog command can be used iteratively to read all or part of the entries in the PLDM Event Log. Entries are returned one at a time.The data for one or more entries may be requested. Table 50 describes the format of this command.

To use the command to start reading from the first entry in the log:

- Set entryID to 0 and transferOperationFlag to GetFirstPart.
- Issue the command to get the first portion of data for the first entry in the log.
- Take the nextEntryID and nextTransferOperationFlag data from the response and use it as the entryID and transferOperationFlag for the next request.
- Repeat this until the desired number of entries have been read or the end of the log has been reached.

The FindPLDMEventLogEntry command can be used to get the entryID for an entry that is at an offset into the log, or that has a timestamp that is older or newer than a given value. This entryID can then be used in the ReadPLDMEventLog command, along with setting transferOperationFlag = GetFirstPart, to begin reading the log starting with the found entry.

Table 50 – ReadPLDMEventLog Command Format

Type	Request Data
uint32	**entryID** A handle that identifies a particular log entry to be transferred or that is in the process of being transferred. The entryID values for the first portion of a given record are required to be unique and unchanging among all entries that are presently in the log. If the data for the entry is split across multiple responses, the entryID is also used to track which portion of the record is being returned in the response. How this is accomplished is implementation specific. For example, one possible implementation would be to use the upper bits of the entryID as an ID for the overall record, and the least significant bits of entryID to track an offset into the record. The entryID that is delivered in the response when in the middle of a multipart transfer (splitEntry = firstFragment or middleFragment) is allowed to time out. The timeout value is specified in the Event Log PDR. This provision is made to allow the responder implementation to assign a temporary ID and buffer space that can be freed up if the requester does not complete the multipart transfer of an entry. The default value for the timeout is the same value that is used for PDR Handle Timeouts, **MC1**. (See 29). If PDRs are not used, a requester should assume the default timeout value is being used unless the requester has a-priori knowledge of the implementation. value: Set to 0x00000000 and transferOperationFlag = GetFirstPart to start reading from the first (oldest) entry in the log;

Type	Request Data
enum8	**transferOperationFlag** The operation flag indicates whether this is the start of a new transfer or the continuation of a multipart transfer of an entry. GetFirstPart identifies transfer of the first entry of a multiple entry read. GetNextPart refers to a request to transfer entries that follow the first entry in a multiple entry transfer. Possible values: {GetNextPart=0x00, GetFirstPart=0x01}

Type	Response Data
enum8	**completionCode** Possible values: { PLDM_BASE_CODES, INVALID_TRANSFER_OPERATION_FLAG=0x81, INVALID_ENTRY_ID=0x82, }
uint32	**nextEntryID** An implementation-specific handle that is used by the implementation to track and identify the next portion of the transfer. This value is used as the dataTransferHandle to retrieve the next portion of eventLog data. Note that if the value for the splitEntry field (below) is firstFragment or middleFragment, the nextEntryID value is an ID that identifies the next *portion* of the record that is being transferred. If splitEntry field is full or lastFragment, the nextEntryID is the ID for the first portion of the next record in the log. special value: 0x00000000 = No next record. This value is only allowed when splitEntry = full or lastFragment. It indicates that there are no records that follow in the log. That is, the PLDMEventLogData that is being returned in the response holds the last portion of data for the last record in the log.

enum8	**splitEntry**
	value: {
	full, // All of the data for the entry is provided in the entryData field.
	firstFragment, // The eventData for the entry is split across ReadPLDMEventLogmessages.
	// The entryData field holds the first portion of the data for the entry.
	middleFragment, // The eventData for the entry is split across ReadPLDMEventLogmessages.
	// The entryData field holds a middle portion of the data for the entry.
	lastFragment // The eventData for the entry is split across ReadPLDMEventLogmessages.
	// The entryData field holds the last portion of the data for the entry.
	}

–	**PLDMEventLogData** The data or partial data for the requested PLDM Event Log entry. Entries are transferred starting from the oldest to the newest.
If splitEntry = lastFragment	

Type	Response Data
uint8	**transferCRC** A CRC-8 for the overall PLDM Event Log entry. This is provided to help verify data integrity when the entry is transferred using a multipart transfer. The CRC is calculated over the entire PLDM Event Log entry data as specified in Table 5 using the polynomial $x^8 + x^2 + x^1 + 1$ (This is the same polynomial used in the MCTP over SMBus/I^2C transport binding specification). The CRC is calculated from most-significant bit to least-signficant bit on bytes in the order that they are received. This field is only present when splitEntry = lastFragment.

1877 **Table 51 – PLDMEventLogData Format**

Type	Field
uint8	**transferredDataSize** If splitEntry = full, then dataSize = number of bytes of entryData for the entire entry. If splitEntry = firstFragment, middleFragment, or lastFragment, then dataSize = number of bytes of entryData for the portion that is being transferred.
–	**transferredEntryData** Data for all or part of an event log entry, depending on whether the entry is split across PLDM messages. See 13.7 for PLDM Event Log entry formats.

1878 ## 23.7 GetPLDMEventLogPolicyInfo Command

1879 The GetPLDMEventLogPolicyInfo command returns details about the different log clearing policies that
1880 are supported for the particular PLDM Event Log implementation. Table 52 describes the format of this
1881 command.

1882 **Table 52 – GetPLDMEventLogPolicyInfo Command Format**

Type	Request Data
enum8	**logClearingPolicy** This parameter selects the logClearingPolicy for which information is to be returned. See 13.4 for a description of the log clearing policies. The command returns the same fields regardless of whether they are used by the selected policy. Fields are filled with a special value if they are not used by the policy. The PLDM Event Log PDR indicates which policies are supported. value: { fillAndStop, FIFO, clearOnAge }
Type	**Response Data**
enum8	**completionCode** value: { PLDM_BASE_CODES }

Type	Response Data
bitfield8	**configurableParameterSupport** This information and the following fields are specific to the logClearingPolicy that was selected in the request. [7:5] – reserved [4:3] – 00b = M and MPercentage are not configurable. 01b = M is configurable 10b = MPercentage is configurable. 11b = reserved [2:1] – 00b = N and NPercentage are not configurable. 01b = N is configurable. 10b = NPercentage is configurable. 11b = reserved [0] – 1b = Age is configurable.
uint32	**NMin** The smallest number that the implementation accepts or uses as a value for N for the given logClearingPolicy (see 13.4). special value: Return 0x00000000 if the policy implementation uses NPercentage instead of N, or if the policy does not use an N value.
uint32	**NMax** The largest number that the implementation accepts or uses as a value for N for the given logClearingPolicy (see 13.4). special value: Return 0x00000000 if the policy implementation uses NPercentage instead of N, or if the policy does not use an N value.
Type	**Response Data**
uint8	**NPercentageMin** The smallest number that the implementation accepts or uses as a value for NPercentage for the given logClearingPolicy (see 13.4). value: 1 to 100; all other values = reserved special value: Return 0x00 if the policy implementation uses N instead of NPercentage, or if the policy does not use an NPercentage value.
uint8	**NPercentageMax** The largest number that the implementation accepts or uses as a value for NPercentage for the given logClearingPolicy (see 13.4). value: 1 to 100; all other values = reserved special value: Return 0x00 if the policy implementation uses N instead of NPercentage, or if the policy does not use an NPercentage value.
uint32	**MMin** The smallest number that the implementation accepts or uses as a value for M for the given logClearingPolicy (see 13.4). special value: Return 0x00000000 if the policy implementation uses MPercentage instead of M, or if the policy does not use an M value.

uint32	**MMax** The largest number that the implementation accepts or uses as a value for M for the given logClearingPolicy (see 13.4). special value: Return 0x00000000 if the policy implementation uses MPercentage instead of M, or if the policy does not use an M value.
uint8	**MPercentageMin** The smallest number that the implementation accepts or uses as a value for MPercentage for the given logClearingPolicy (see 13.4). value: 1 to 100; all other values = reserved special value: Return 0x00 if the policy implementation uses M instead of MPercentage, or if the policy does not use an MPercentage value.
uint8	**MPercentageMax** The largest number that the implementation accepts or uses as a value for MPercentage for the given logClearingPolicy (see 13.4). value: 1 to 100; all other values = reserved special value: Return 0x00 if the policy implementation uses M instead of MPercentage, or if the policy does not use an MPercentage value.
uint32	**ageMin** The smallest value that the implementation accepts or uses as a value for age in seconds for the given logClearingPolicy (see 13.4). special value: Return 0x00000000 if the policy does not use an age value.
uint32	**ageMax** The largest value that the implementation accepts or uses as a value for age in seconds for the given logClearingPolicy (see 13.4). special value: Return 0x00000000 if the policy does not use an age value.

23.8 SetPLDMEventLogPolicy Command

The SetPLDMEventLogPolicy command is used to select and configure the PLDM Event Log clearing policies. Table 53 describes the format of the command.

Table 53 – SetPLDMEventLogPolicy Command Format

Type	Request Data
enum8	**selectedLogClearingPolicy** This parameter selects the log clearing policy to be used by the PLDM Event Log. See 13.4 for a description of the log clearing policies. value: { fillAndStop, FIFO, clearOnAge }

Type	Request Data

enum8	**setOperation**
	value: {
	configureOnly, // Change the configuration of the policy identified by // selectedLogClearingPolicy by using the following configuration parameters, // but do not change which policy is selected as the active policy.
	setOnly, // Set the active policy to the policy identified by selectedLogClearingPolicy, but // do not set any of the configuration parameters. If this setOperation is used, // the following configuration parameters in the request shall be ignored by the // responder.
	configureAndSet // Set the active policy to the policy identified by selectedLogClearingPolicy and // set the configuration parameters for the selected policy using the following // configuration parameters.
	}

uint32	**N** The number of entries that will be automatically cleared for the given selectedLogClearingPolicy. See 13.4 for a description of the log clearing policies. special value: Use 0x00000000 if the policy implementation does not support a configurable N value. If the responder does not support a configurable N value, an error completionCode must be returned if this is set to a value other than 0.
uint8	**NPercentage** The percentage of the log that will be automatically cleared for the given selectedLogClearingPolicy. See 13.4 for a description of the log clearing policies. value: 1 to 100; all other values = reserved special value: Use 0x00 if the policy implementation does not support NPercentage as a configurable value. If the responder does not support a configurable NPercentage value, an error completionCode must be returned if this is set to a value other than 0.
uint32	**M** The number of entries that must be in the log before entries will be automatically cleared based on the selectedLogClearingPolicy. See 13.4 for a description of the log clearing policies. special value: Use 0x00000000 if the policy implementation does not support a configurable M value. If the responder does not support a configurable M value, an error completionCode must be returned if this is set to a value other than 0.
uint8	**MPercentage** The percentage of the log that must be filled before entries will be automatically cleared based on the selectedLogClearingPolicy. See 13.4 for a description of the log clearing policies. value: 1 to 100; all other values = reserved special value: Use 0x00 if the policy does not support MPercentage as a configurable value. If the responder does not support a configurable MPercentage value, an error completionCode must be returned if this is set to a value other than 0.
uint32	**age** This parameter sets the age interval in seconds for the given selectedLogClearingPolicy. See 13.4 for a description of the log clearing policies. special value: Use 0x00000000 if the policy implementation does not support a configurable age. If the responder does not support a configurable age, an error completionCode must be returned if this is set to a value other than 0.

Type	Request Data
Type	**Response Data**
enum8	**completionCode** value: { PLDM_BASE_CODES }

23.9 FindPLDMEventLogEntry Command

This command can be used to obtain the Entry ID value for the first entry in the Event Log that meets the identified search parameter. This value can then be used in the ReadPLDMEventLog command to start reading the log from that entry onward. The search parameters support finding the first entry that is newer or older than a specified timestamp value, or the entry that corresponds to a particular offset from the start or the present end of the log. Table 54 describes the format of this command.

Table 54 – FindPLDMEventLogEntry Command Format

Type	Request Data
enum8	**searchType** value: {newerThan, olderThan, offsetFromStart, offsetFromEnd}
uint32	**startingPoint** The EntryID for the log entry or the offset from which searching will start. Searches include the entry at the identified starting point. The search always occurs in the direction from the start of the log (first entries) to the end of the log (last entries). If searchType = newerThan or olderThan: A non-zero value indicates an EntryID to start searching from. Use the value 0x00000000 to start searching from the first entry in the log. Use the value 0xFFFFFFFF to start searching from the last entry in the log. If searchType = offsetFromStart: The value identifies the Nth entry from the start of the log. For example, if starting point = 10 the search will start with the 10th entry at the beginning of the log. An error completionCode shall be returned if the value exceeds the number of entries in the log. If searchType = offsetFromEnd: The value identifies the Nth entry from the end of the log. For example, if starting point = 10 and the log contains 100 entries, the search will start with the 91st entry. An error completionCode shall be returned if the value exceeds the number of entries in the log.
compareTimestamp *The compareTimestamp fields are only present when searchType = newerThan or olderThan.* *If searchType = newerThan, the response will hold the entryID for the first log entry that was found with a timestamp that is more recent than or equal to compareTimestamp.* *If searchType = olderThan, the response will hold the entryID for the first log entry that was found with a timestamp that is older than or equal to compareTimestamp.*	
sint8	**compareTimestampUTCOffset** The UTC offset for the log entry timestamp in increments of 1/2 hour. special value: 0xFF = unspecified

uint40	**compareTimestampSeconds** This value corresponds to a 40-bit unsigned integer representing the number of seconds since midnight UTC of January 1, 1970 (not counting leap seconds). 0x0000000000 = unspecified.
uint8	**compareTimestamp100s** This value provides a number of 1/100ths of a second added to **entryTimestampSeconds**. value: 0 to 99. special value: 0xFF = unspecified. This value is used if the implementation timestamps entries to no finer than a one second resolution.
Type	**Response Data**
uint32	**entryID** The entryID for the found log entry. This value can be used in the ReadPLDMEventLog command. special value: 0xFFFFFFFF = Not found. The command did not find a record matching the searchType.
enum8	**completionCode** value: { PLDM_BASE_CODES, INVALID_SEARCH_TYPE = 0x80 }

24 PLDM State Sets

PLDM State Sets are specified enumerations for sets of state information that can be returned from PLDM state sensors. State sets may also be used to provide a common definition for state information used by other parts of PLDM.

The state sets are the basis of state data that can be mapped as a data source into CIM properties that return state information, and also provide state information that can be used for monitoring and controlling the operation of PLDM itself.

PLDM State Sets are defined in DSP0249. This specification defines a numeric ID for each different state set, defines the enumeration values for the states that make up the set, and provides definitions for each state within the set. Because the state sets are expected to be extended over time as new CIM properties are defined, the state sets are maintained in a separate document to allow them to be extended without having to revise other PLDM specifications.

25 Platform Descriptor Records (PDRs)

PLDM can return collections of semantic and association information about the platform by using collections of information called Platform Descriptor Records (PDRs). This information can include records that return semantic information about sensors, such as their sensor resolution, tolerance, accuracy, and conversion factors, as well as records that return information about the associations between sensors and monitored entities, management controllers, effecters, and other platform associations or capabilities.

PDRs are called descriptor records because they are mainly used to describe the subsystem, rather than to control it or configure it.

25.1 PDR Repository Updates

A PDR Repository is not necessarily a static set of records. A platform that includes hot plug devices or supports field updates may have its PDRs change over time as devices are added or removed. Even if the implementation of a particular platform management subsystem is static, the PDRs must still be generated and installed so that they represent the semantic information and relationships of the particular platform implementation.

PLDM does not specify the mechanisms by which PDRs get generated, installed, or updated. This was done intentionally to allow the vendor of the PDR Repository devices to create update or configuration utilities that are appropriate for the particular implementation. PLDM does, however, specify how the information is accessed and used.

25.2 Internal Storage and Organization of PDRs

The PLDM specifications do not place any requirements on how PDRs are internally stored or organized within the device or devices that implement the PDR Repository. PDRs may be compressed, stored with additional pointers, sorted, cross indexed, split, replicated, and so on, as long as the information meets the byte order and formats specified for the PDR commands. The byte order and formats for PDRs are specified in tables for the different PDR types in section 28.

25.3 PDR Types

PDRs are identified by a PDR Type value that is given in a field in the header for each different PDR. PDR types include type values for records that identify PDRs for PLDM numeric and state sensors, records that direct sensor initialization, records that describe PLDM effecters, and so on. The PDR Type values are given in Table 64.

25.4 PDR Record Handles

All PDRs are assigned an opaque numeric value called the recordHandle. This value is used for accessing individual PDRs within the PDR Repository. Additional information about recordHandles and their use is provided in the specification of the GetPDR command (see 26.2).

25.5 Accessing PDRs

For most implementations, PDR data rarely changes. A party that uses PDR information may want to cache certain information to reduce the need for accessing the PDR Repository . The GetPDRRepositoryInfo command provides time stamps that can be used to identify whether any record data in a particular PDR Repository has changed. If a change is detected the party can then update its cached information as necessary.

26 PDR Repository Commands

This section describes the commands for accessing PDRs from a PDR Repository per this specification. The command numbers for the PLDM messages are given in section 30.

If a PDR Repository is implemented, the Mandatory/Optional/Conditional (M/O/C) requirements shown in Table 55 apply.

Table 55 – PDR Repository Commands

Command	M/O/C	Reference
GetPDRRepositoryInfo	M	See 26.1.
GetPDR	M	See 26.2.
FindPDR	O [1]	See 26.3.
RunInitAgent	C [2]	See 26.4.

[1] Because this command reduces or eliminates the need to 'walk' the PDRs in order to find particular records, it is recommended for Primary PDR Repositories that include multiple entity-association hierarchies, use a wide range of PDR types, incorporate a large number of PDRs, or where specific PDRs, such as OEM PDRs, need to be accessed by entities that do not care about other PDRs types.

[2] The RunInitAgent command is required for the terminus that provides the primary PDR Repository.

26.1 GetPDRRepositoryInfo Command

The GetPDRRepositoryInfo command returns information about the size and number of records in the PDR Repository of a particular PLDM terminus, and time stamps that indicate the last time that an update to the repository occurred. Two time stamps are returned, one that indicates whether any PLDM standard PDRs have changed, and another that indicates whether any OEM PDRs (if any) have changed.

See 25.5 for more information about accessing PDRs. Table 56 describes the format of this command.

Table 56 – GetPDRRepositoryInfo Command Format

Type	Request Data
–	none
Type	**Response Data**
enum8	**completionCode** value: { PLDM_BASE_CODES }
enum8	**repositoryState** value: { available, // Record data can be read from the repository. updateInProgress , // Record data is unavailable because an update is in progress. failed // Record data is unavailable because of a detected failure // condition. }
timestamp104	**updateTime** This time stamp identifies when the standard PDR Repository data was originally created, or the time of the most recent update if the data has been updated after it was created. This time does not include changes of PDRs that have a PDR Type of "OEM".
timestamp104	**OEMUpdateTime** This time stamp identifies when OEM PDRs in the PDR Repository were originally created, or the time of the most recent update if the data has been updated after it was created.
uint32	**recordCount** Total number of PDRs in this repository

uint32	**repositorySize** Size of the PDR Repository in bytes. This value provides information that can be used for helping estimate buffer size requirements when accessing PDRs. This size covers only the cumulative sizes of the PDR record fields. This size does not include the size for any internal header structures that are used for maintaining the PDRs. This number does not report and may not directly correlate to the amount of internal storage used for PDRs because, for example, an implementation may elect to internally compress or use other encodings of the PDR data. An implementation is allowed to round this number up to the nearest kilobyte (1024 bytes).
uint32	**largestRecordSize** Size of the largest record in the PDR Repository in bytes. This value provides information that can be used for helping estimate buffer size requirements when accessing PDRs. An implementation is allowed to round this number of up to the nearest 64-byte increment.
uint8	**dataTransferHandleTimeout** The minimum interval, in seconds, that a dataTransferHandle value remains valid after it was delivered in the response of a GetPDR or FindPDR command. special values: { 0x00 = no timeout, 0x01 = default minimum timeout (**MC1**, see section 29), 0xFF = timeout >254 seconds. Any timeout values that are less than the specified default minimum timeout are illegal. }

1964 ## 26.2 GetPDR Command

1965 The GetPDR command is used to retrieve individual PDRs from a PDR Repository. The record is
1966 identified by the PDR recordHandle value that is passed in the request. The command can also be used
1967 to dump all the PDRs within a PDR Repository.

1968 ### 26.2.1 GetPDR Command Format

1969 Table 57 describes the format of the GetPDR command.

1970 **Table 57 – GetPDR Command Format**

Type	Request Data
uint32	**recordHandle** The recordHandle value for the PDR to be retrieved. For more information, see 26.2.3 and 26.2.4. special value: {0x0000_0000 = Get first PDR in the repository}
uint32	**dataTransferHandle** A handle that is used to identify a particular multipart PDR data transfer operation. For more information, see 26.2.7 and 26.2.8. special value: { use 0x0000_0000 if the transferOperationFlag is GetFirstPart }
enum8	**transferOperationFlag** Indicates whether this request is for the first portion of the PDR value: { GetNextPart = 0x00, GetFirstPart = 0x01}
uint16	**requestCount** The maximum number of record bytes requested to be returned in the response to this instance of the GetPDR command. NOTE: The responder may return fewer bytes than were requested.

Type	Request Data
uint16	**recordChangeNumber** value: If the transferOperationFlag field is set to GetFirstPart, set this value to 0x0000. If the transferOperationFlag field is set to GetNextPart, set this to the recordChangeNumber value that was returned in the header data from the first part of the PDR (see 28.1).
Type	**Response Data**
enum8	**completionCode** value: { PLDM_BASE_CODES, INVALID_DATA_TRANSFER_HANDLE = 0x80, INVALID_TRANSFER_OPERATION_FLAG=0x81, INVALID_RECORD_HANDLE = 0x82, INVALID_RECORD_CHANGE_NUMBER = 0x83, TRANSFER_TIMEOUT = 0x84, REPOSITORY_UPDATE_IN_PROGRESS = 0x85 }
uint32	**nextRecordHandle** The recordHandle for the PDR that is next in the PDR Repository. The value can be used as the recordHandle in a subsequent GetPDR command as a means of sequentially reading PDRs from the repository. PDRs are not required to be returned in any particular order. special value: { 0x0000_0000 = no more PDRs following this one. }
uint32	**nextDataTransferHandle** A handle that identifies the next portion of the PDR data to be transferred, if any portions are remaining special value: { returns 0x0000_0000 if there is no remaining data. }
enum8	**transferFlag** Indicates what portion of the PDR is being transferred value: {Start = 0x00, Middle = 0x01, End = 0x04, StartAndEnd = 0x05}
uint16	**responseCount** The number of recordData bytes returned in this response special value: { returns 0x0000 if the requestCount was 0x0000 }
(var)	**recordData** PDR data bytes. This field is absent if responseCount = 0x0000. The number of PDR data bytes returned in this field must match responseCount.
If transferFlag = End	
uint8	**transferCRC** A CRC-8 for the overall PDR. This is provided to help verify data integrity for a PDR when it is transferred using a multipart transfer. The CRC is calculated over the entire PDR data using the polynomial $x^8 + x^2 + x^1 + 1$ (This is the same polynomial used in the MCTP over SMBus/I^2C transport binding specification). The CRC is calculated from most-significant bit to least-signficant bit on bytes in the order that they are received. This field is only present when transferFlag = End.

26.2.2 Single-Part and Multipart Transfers

The data from a given PDR may be accessed using a single-part or multipart transfer. A single transfer occurs when the entire PDR content is delivered using a single GetPDR command response. A multipart transfer is required when the record data exceeds either the amount of data that the responder can return using a single response, or when it exceeds the amount of data that the requester can accept in a single response. In this case, the GetPDR command is used iteratively to retrieve the first portion of the record and then subsequent portions. Additional information and requirements for multipart transfers is provided in 26.2.7.

Partial transfers from the beginning of a record are allowed. That is, a requester is not required to read out an entire record if only the beginning portion of the record data is of interest.

26.2.3 PDR recordHandle

The recordHandle is an opaque value that is used by the implementation of the PDR Repository to identify individual records and to track where the next data of a multipart transfer will come from. This value is obtained from the response data of a previous instance of the GetPDR command. A special value of 0x0000_0000 is used to retrieve the first PDR in the repository.

Some implementations may use the recordHandle as a direct offset into storage memory, others may use it as offset that is relative to the start of the PDR data, and others may use it as a table or list index.

26.2.4 PDR recordHandle Retention

The recordHandle values that are used to access a particular PDR may change when the recordChangeNumber is changed. recordHandle values are also not guaranteed to endure across connections to the given PLDM terminus that is implementing the command. A party that needs to re-establish a connection to the terminus must assume that any PDR recordHandle values that it previously had are no longer valid. If any multipart transfers were not completed before the connection was re-established, those transfers must be restarted from the beginning.

26.2.5 PDR recordChangeNumber

The recordChangeNumber provides a mechanism for preventing the use of invalid PDR data if a record's data gets updated while the record was in the process of being read out. The mechanism helps ensure that a requester does not get the first parts from an earlier version of the record and remaining parts from a later version of the record. The recordChangeNumber can also be used to help a requester scan and identify which PDRs may have changed after an update to the PDR Repository has occurred.

To accomplish this, the PDR recordChangeNumber that is returned in the GetPDR response is required to change whenever the data of a PDR changes during a multipart access of the PDR. The party that is accessing a PDR gets the recordChangeNumber when the first part of the record is returned. This number is then used as one of the input parameters when retrieving the remaining parts of the record.

The PLDM responder compares this number against the present recordChangeNumber that is associated with the record. If there is a mismatch, the PLDM responder returns an error completionCode. The requester can then handle the error by starting the PDR transfer over.

It is recommended that an implementation update the recordChangeNumber only for records that have changed due to an update. However, implementations may elect to update the recordChangeNumber for some or all unchanged records. This latter approach can be used for small and simple implementations in which PDR exits and updates are rare, but should be avoided in large implementations in which the party that is accessing the PDR data may see significant delays due to the unnecessary re-reading and handling of PDRs that have not actually changed.

26.2.6 PDR Repository Time Stamp and PDR Repository Locking

The recordChangeNumber mechanism protects against inconsistent data only on a per record basis; it does not automatically protect against inconsistencies that may occur due to individual updates of interrelated records. For example, if record A and B are interrelated and both need synchronized updates, it is possible that a party could access the records at a time when A has been updated but B has not. The individual records would be correct, but their interrelationship could be incorrect.

The party that is updating the PDRs can lock the repository while updates are occurring (the mechanisms used for updating and locking the PDRs are outside this specification). In this case, commands such as the GetPDR command will return an error completionCode indicating that the repository records are inaccessible because an update is in progress. Update-in-progress status is also available in the GetPDRRepositoryInfo command.

A party that updates records in a PDR Repository while PLDM command handling is active must either lock the PDRs and update the time stamp and recordChangeNumber values before making the repository available, or it must update the time stamp and recordChangeNumber values as each individual updated record is made available through PLDM.

The PDR Repository has a time stamp that can be read using the GetPDRRepositoryInfo command. The time stamp value is updated whenever changes are made to the repository. A party that is accessing multiple PDRs and relying on an interrelationship between those records should check the time stamp value after retrieving the records to verify that a repository update did not occur while the records were being accessed.

If an update has occurred while records were being read, the records should either be re-read or their recordChangeNumber values checked to see if they have changed. Because the recordChangeNumber is in the beginning portion of a PDR, it is not necessary to read the entire record to get the value.

26.2.7 Multipart PDR Transfers

The command is intended to support multipart transfer of PDR data only in a sequential manner, starting from the beginning of the PDR. Random access to a middle portion of a PDR is not required by implementations, nor is it intentionally supported as an option in this specification.

The dataTransferHandle value is therefore required to remain valid only for use with the next GetNextPart operation from a given requester. Although many implementations will likely return the same data for an identical sequence of PDR access commands regardless of the ID of the requester, an implementation may allocate and track dataTransferHandles on a per-requester basis. The dataTransferHandle information given to one requester might not be usable by another requester.

26.2.8 PDR dataTransferHandle Retention

The dataTransferHandle value for a multipart transfer is required to remain valid for at least MC1 seconds after it has been delivered in a response. After this interval, an implementation may elect to implement a timeout and terminate the multipart transfer. To support this, an implementation would use some aspect of the recordHandle value to track the particular multipart transfer in progress.

The provisions that allow a dataTransferHandle value to become invalid or expire allow implementations the option of temporarily queuing PDR data in memory and freeing up that memory if the record data is no longer being accessed. The provisions eliminate the need for the recordHandle values for a given request to remain valid indefinitely.

26.2.9 Multipart PDR Transfer Termination and Timeouts

No formal release mechanism exists for multipart PDR transfers. Multipart transfers may be terminated by the responder under the following conditions:

- The responder implementation may restrict a given requester to having only one PDR transfer in process at a time. If the requester starts a different transfer, the earlier multipart transfer that was in progress may be aborted.
- The responder implementation may terminate any multipart PDR transfer in progress following expiration of the PDR dataTransferHandle retention interval, MC1.
- Execution of the Initialization Agent function may terminate a multipart PDR transfer in progress.

26.2.10 Reuse of Prior Request Values

Except for the first part of a PDR, an implementation is not required to support returning a previously transferred portion of a PDR after the transfer has progressed to a later portion. For example, if the first three portions of a PDR have been transferred, the implementation may not allow a re-transfer of the second portion without restarting the transfer from the beginning. If an implementation does accept request parameters that were used for reading an earlier portion of a given PDR, it must return the same PDR data that was returned for the original request.

26.3 FindPDR Command

The FindPDR command is provided to improve the efficiency of common types of access to a Primary PDR Repository. The FindPDR command is primarily designed to provide operations that can assist a MAP in using information from the PDRs to instantiate CIM objects and associations.

The FindPDR command returns the PLDMHandleType and PLDMHandle values for a particular PDR or set of PDRs, depending on the parameters that were passed in the request. The response can also include the first portion of the PDR data. The response from the FindPDR command can then be used with the GetPDR command to read the PDR or the remaining portions of the PDR.

To reduce implementation and validation complexity, the FindPDR command does not provide a generic search engine but supports only a limited number of different preconfigured queries that are restricted to using particular key fields within the PDRs.

For example, the FindPDR command can be used to find all the PDRs that have a particular PLDMTerminusHandle, or Entity Association PDRs that have a common Container ID. It can also be used to find Numeric Sensor PDRs that share a particular type of monitored numeric unit, such as temperature, or state sensors that use a particular state set. However, the FindPDR command does not support less common operations such as finding records that have a particular hysteresis value setting or state sensors that implement a particular state from within a state set.

The findParameters field holds the PDRType-specific search fields. The format of findParameters is identified by the parameterFormatNumber that is passed in the request. The findParameters value may be applicable to more than one PDRType. The parameterFormatNumber and PDRType field in the request are used together to identify which PDRs should be searched. Table 59 lists the values for parameterFormatNumber and the PDRType values that are associated with each parameterFormatNumber. Table 60 lists the different PDR fields that make up the findParameters value for each different parameterFormatNumber.

If the PDRType field value is set to 0, all of the PDRType values that are specified for the parameterFormatNumber in Table 59 are searched. Otherwise, only PDRs that have the given PDRType value are searched.

For example, if PDRType = 0 and parameterFormatNumber = 7, all PDRs with PDRType values that are identified for searching with parameterFormatNumber = 7 are searched: Numeric Effecter Initialization, State Effecter Initialization, and Effecter Auxiliary Names. If the PDRType is set to the value for State Effecter Initialization PDR, only State Effecter Initialization PDRs are searched.

The findParameters value is included in each request to eliminate the need for implementations to retain the findParameters value when a multi-PDR find operation is being done.

Table 58 describes the format of this command.

Table 58 – FindPDR Command Format

Type	Request Data
uint32	**findHandle** A handle that is used to track the point from which searching should resume. With the exception of the first find, the nextFindHandle value is set with the nextFindHandle value from the previous response for the find operation in process. special values: { use 0x0000_0000 if the findOperation is findFirst, 0xFFFF_FFFF = reserved. } NOTE: This field has the same retention specifications as the dataTransferHandle field used in the GetPDR command. See 26.2.4 for more information.
enum8	**findOperationFlag** Indicates whether this request is for locating the first matching PDR. value: { findNext = 0x00, findFirst = 0x01}
uint16	**requestCount** The maximum number of record bytes requested to be returned in the response to this instance of the FindPDR command. NOTE: The responder may return fewer bytes than were requested.
uint16	**PDRType** The PDRType for the records to be located. special value: 0x0000 = match any PDRType.
uint8	**parameterFormatNumber** A number that identifies the format and number of parameters in the findParameters field. Table 60 lists the different PDR fields that make up the findParameters value for each different parameterFormatNumber.
bitfield16	**wildcards** Each Nth bit position indicates whether the Nth parameter from the findParameters field should be matched or ignored (treated as a wildcard). Use 0b for ay bit position for which a parameter is not defined. [15] – 1b = sixteenth parameter value in findParameters must be matched 0b = sixteenth parameter value in findParmeters is ignored **…** [0] – 1b = first parameter value in findParameters must be matched 0b = first parameter value in findParameters is ignored

varies	**findParameters** A series of parameters that correspond to fields in the PDRs that are used for the find operation. Table 60 lists the PDR fields that make up the findParameters value for each parameterFormatNumber. Each field within findParameters is provided in the order listed in Table 60, starting from the top of the table to the bottom for the column that is identified by parameterFormatNumber. Dots in the column identify which parameters are to be provided in findParameters. The data type and size (for example, uint8) and meaning of each parameter are given by the definition of the PDR that is identified by the PDRTypes for the given parameterFormatNumber, as listed in Table 59. Values for all parameters must be provided even if a particular parameter is to be ignored in the search. The values for ignored parameters shall not be checked for validity by the responder. An implementation may optionally check non-wildcard parameters for validity and return an error completionCode if the parameter is not a legal value for the corresponding field in the PDR.
Type	**Response Data**
enum8	**completionCode** value: { PLDM_BASE_CODES, INVALID_FIND_HANDLE = 0x80, INVALID_FIND_OPERATION_FLAG = 0x81, INVALID_PDR_TYPE = 0x82, INVALID_PARAMETER_FORMAT_NUMBER = 0x83, INVALID_FIND_PARAMETERS = 0x84, REPOSITORY_UPDATE_IN_PROGRESS = 0x85 }
uint32	**nextFindHandle** A handle that identifies the next part of a Find operation that may return more than one PDR. The implementation uses this field to track the point from which it needs to resume searching. An implementation may elect to look ahead to see if there are any more matching PDRs before sending the response, or it may elect to wait until getting the next request before searching to see if there are any remaining matching records. The “look-ahead” approach is recommended. special values: { returns 0x0000_0000 if no matching PDR was found. returns 0xFFFF_FFFF if this response holds data for the last matching PDR. That is, there are no more matching PDRs beyond this one.}
uint32	**nextDataTransferHandle** A handle that identifies the next portion of the PDR data to be transferred, if any portions are remaining. This value is used in the GetPDR command to retrieve any remaining portions of the PDR. special value: { returns 0x0000_0000 if there is no remaining recordData beyond the recordData that is being returned in this response data. }
enum8	**transferFlag** Indicates what portion of the PDR is being transferred value: {Start = 0x00, Middle = 0x01, End = 0x04, StartAndEnd = 0x05}
uint16	**responseCount** The number of recordData bytes returned in this response special value: { returns 0x0000 if the requestCount was 0x0000 }

(var)	**recordData** PDR data bytes. This field is absent if responseCount = 0x0000. Otherwise, the number of PDR data bytes returned in this field must match responseCount.

Table 59 – FindPDR Command Parameter Format Numbers

<table>
<tr><th>PDRType</th><th>parameterFormatNumber</th></tr>
<tr><td>ANY = 0</td><td>1 [1]</td></tr>
<tr><td>Event Log</td><td>1 [2]</td></tr>
<tr><td>Terminus Locator</td><td>2</td></tr>
<tr><td>Numeric Sensor</td><td>3</td></tr>
<tr><td>Numeric Sensor Initialization</td><td rowspan="3">4</td></tr>
<tr><td>State Sensor Initialization</td></tr>
<tr><td>Sensor Auxiliary Names</td></tr>
<tr><td>State Sensor</td><td>5</td></tr>
<tr><td>Numeric Effecter</td><td>6</td></tr>
<tr><td>Numeric Effecter Initialization</td><td rowspan="3">7</td></tr>
<tr><td>State Effecter Initialization</td></tr>
<tr><td>Effecter Auxiliary Names</td></tr>
<tr><td>State Effecter</td><td>8</td></tr>
<tr><td>Entity Association</td><td>9</td></tr>
<tr><td>Interrupt Association</td><td>10</td></tr>
<tr><td>OEM Unit</td><td>11</td></tr>
<tr><td>OEM State Set</td><td>12</td></tr>
<tr><td>OEM Entity</td><td>13</td></tr>
<tr><td>OEM Device</td><td rowspan="2">14</td></tr>
<tr><td>OEM</td></tr>
<tr><td>OEM Unit</td><td rowspan="5">15 [3]</td></tr>
<tr><td>OEM State Set</td></tr>
<tr><td>OEM Entity</td></tr>
<tr><td>OEM Device</td></tr>
<tr><td>OEM</td></tr>
</table>

[1] The entire contents of the repositorycan be read by using this format along with PDRType = ANY and PLDMTerminusHandle set for “wildcard.”

[2] The PLDMTerminusHandle parameter must be set for “wildcard” when using this format to search for Event Log PDRs.

[3] This search format can be used to return all PDRs that have any of the indicated "OEM" PDRType values or all PDRs that have any of the indicated "OEM" PDRType values and match a particular vendorIANA.

Table 60 – FindPDR Command Parameter Formats

Parameter (PDR field)	parameterFormatNumber														
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
PLDMTerminusHandle	●	●	●	●	●	●	●	●		●	●	●	●	●	●
TID		●													
sensorID			●	●	●					●					
effecterID						●	●	●							
stateSetID					●			●							
containerID			●			●		●	●						
associationType									●						
entityType			●			●									
entityInstanceNumber			●			●									
baseUnit			●			●									
unitModifier			●			●									
rateUnit			●			●									
baseOEMUnitHandle			●			●									
auxUnit			●			●									
auxUnitModifier			●			●									
auxrateUnit			●			●									
auxOEMUnitHandle			●			●									
containerEntityType									●						
containerEntityInstanceNumber									●						
containerEntityEntityID									●						
interruptTargetEntityType										●					
interruptTargetEntityInstanceNumber										●					
interruptTargetEntityContainerID										●					
interruptSourceEntityType										●					
interruptSourceEntityInstanceNumber										●					
interruptSourceEntityContainerID										●					
OEMUnitHandle											●				
OEMStateSetIDHandle												●			
OEMEntityIDHandle													●		
vendorIANA											●	●	●	●	●
OEMUnitID											●				
OEMStateSetID												●			
OEMEntityID													●		
OEMRecordID														●	

26.4 RunInitAgent Command

The RunInitAgent command directs the terminus that provides the Primary PDR Repository to run the Initialization Agent function. This command can be used to trigger a re-initialization of the monitoring and control capabilities in the PLDM subsystem. Table 61 describes the format of the command.

Table 61 – RunInitAgent Command Format

Type	Request Data

bitfield8	**initConditionEmulation**
	This value selects a condition that emulates a transition that triggers the Initialization Agent to run. The Initialization Agent then performs its steps accordingly. For example, if the initConditionEmulation is set to SystemHardReset, the Initialization Agent initializes only those sensors and effecters that have SystemHardReset set in the initCondition parameter of their Initialization PDRs.
	value: {
	0x00 = InitializationAgentRestart, // Directs the Initialization Agent to take the same steps // as it would if the controller that holds the Initialization // Agent was restarted or reinitialized.
	0x01 = PLDMSubsystemPowerUp, // Directs the Initialization Agent to take the same steps // as it would when the PLDM subsystem becomes // powered up.
	0x02 = SystemHardReset, // Directs the Initialization Agent to take the same steps // as it would following a system hard reset.
	0x03 = SystemWarmReset, // Directs the Initialization Agent to take the same steps // as it would following a system warm reset.
	0x04 = PLDMTerminusOnline // Directs the Initialization Agent to initialize the // terminus that has a TID that matches the TID // parameter in this request.
	}

bitfield8	**TID** Terminus ID for the terminus to be initialized when the initConditionEmulation field in this request is set to PLDMTerminusOnline. special value: The value in this field is ignored when the initConditionEmulation field in this request is set to any value other than PLDMTerminusOnline.
Type	**Response Data**
enum8	**completionCode** value: { PLDM_BASE_CODES }

27 PDR Definitions

This section describes certain important characteristic parameters that are provided within the PDRs for interpreting the readings and settings of sensors and effecters.

27.1 Sensor Types

PLDM contains two basic types of sensors that are described using PDRs:

- The PLDM Numeric Sensor is used to obtain a numeric value for a monitored parameter. The sensor definition also optionally includes returning state information based on whether the numeric reading has crossed one or more defined threshold levels.
- The PLDM State Sensor/PLDM Composite State Sensor is used to obtain the present state of a monitored parameter. The PLDM sensor access commands allow an implementation to provide multiple sets of state information using a single access command. When this is done, the implementation is referred to as providing a Composite State Sensor.

27.2 Effecter Types

PLDM contains two basic types of effecters that are described using PDRs:

- The PLDM Numeric Effecter is used to set a numeric value for a monitored parameter. The effecter definition also optionally includes returning state information based on whether the numeric reading has crossed one or more defined threshold levels.
- The PLDM State Effecter/PLDM Composite State Effecter is used to set the present state of a monitored parameter. The PLDM effecter access commands allow an implementation to provide multiple sets of state information using a single access command. When this is done, the implementation is referred to as providing a Composite State Effecter.

27.3 State Sets

State information is returned using an enumeration called a “state set.” Each state set has a different ID number. This number is used within the PDRs to identify what particular state set a sensor or effecter is using. See section 24 for more information.

27.4 Sensor and Effecter Units

This section and following sections describe the fields that are used within PDRs to define and describe sensor and effecter units and related characteristics such as accuracy, tolerance, and resolution.

The type of units that are associated with the value that a sensor returns or monitors, or that an effecter controls, such as volts or amps, is identified in the PDRs by a sensorUnits enumeration, listed in Table 62. Unless otherwise indicated, the units apply to all numeric properties of the sensor, such as the sensor reading, threshold values, and resolution.

Vendor defined units are identified by a special value for OEMUnit. A special PDR called the OEM Unit PDR is used to define the meaning of the OEMUnit when it is used in the PDRs that describe a sensor or effecter. Refer to 28.9 for more information about how OEMUnits are used in PDRs.

Table 62 – sensorUnits Enumeration

0	None	30	Cubic Feet	60	Bits
1	Unspecified	31	Meters	61	Bytes
2	Degrees C	32	Cubic Centimeters	62	Words (data)
3	Degrees F	33	Cubic Meters	63	DoubleWords
4	Degrees K	34	Liters	64	QuadWords
5	Volts	35	Fluid Ounces	65	Percentage
6	Amps	36	Radians	66	Pascals
7	Watts	37	Steradians	67	Counts
8	Joules	38	Revolutions	68	Grams
9	Coulombs	39	Cycles	69	Newton-meters
10	VA	40	Gravities	70	Hits
11	Nits	41	Ounces	71	Misses
12	Lumens	42	Pounds	72	Retries
13	Lux	43	Foot-Pounds	73	Overruns/Overflows
14	Candelas	44	Ounce-Inches	74	Underruns
15	kPa	45	Gauss	75	Collisions
16	PSI	46	Gilberts	76	Packets
17	Newtons	47	Henries	77	Messages
18	CFM	48	Farads	78	Characters
19	RPM	49	Ohms	79	Errors
20	Hertz	50	Siemens	80	Corrected Errors
21	Seconds	51	Moles	81	Uncorrectable Errors
22	Minutes	52	Becquerels	82	Square Mils
23	Hours	53	PPM (parts/million)	83	Square Inches
24	Days	54	Decibels	84	Square Feet
25	Weeks	55	DbA	85	Square Centimeters
26	Mils	56	DbC	86	Square Meters
27	Inches	57	Grays	-	all other = reserved
28	Feet	58	Sieverts		
29	Cubic Inches	59	Color Temperature Degrees K	255	OEMUnit

27.4.1 Base Units

The base unit of measurement associated with the reading values returned by a PLDM Numeric Sensor or set into a PLDM Numeric Effecter are represented by the combination of three fields from the PDR for the sensor: baseUnits, unitModifier, and rateUnits. These fields are interpreted according to the following formula:

$$\text{Sensor/Effecter Units} = \text{baseUnit} * 10^{\text{unitModifier}}\ \text{rateUnit}$$

For example, if baseUnits is Volts and the unitModifier is -6, the units of the values returned are microvolts.

If the rateUnits property is set to a value other than None, the units are further qualified as rate units. In the preceding example, if rateUnits is set to Per Second, the values returned by the sensor are in microvolts/second.

27.4.2 Auxiliary Units

In some cases, additional modification of the base unit of the sensor might be required. For example, acceleration is commonly given in units such as "meters per second per second". The PDRs include a provision for modifying the base units with an additional set of units called auxiliary units. Auxiliary units are defined by three elements: auxUnit, auxUnitModifier, and auxRateUnit. These elements are used in combination with the base units as follows:

$$\text{Sensor/Effecter Units} = \text{baseUnit} * 10^{\text{unitModifier}}\ [\text{rel}]\ \text{auxUnit} * 10^{\text{auxUnitModifier}}\ \text{rateUnit auxRateUnit}$$

[rel] is the relationship between the base unit and the auxiliary unit, as follows:

rel = enum8 { dividedBy, multipliedBy}

And:

dividedBy implies a "/" or "per" relationship, such as "per foot"

multipliedBy implies a "*" operation, such as "foot*lbs (foot-lbs)"

auxUnit and auxRateUnit shall not be used if an equivalent definition can be made using only base units.

27.4.3 Units for Use with CIM

Developers are cautioned that PLDM units may include types of units that are not presently supported by standard CIM objects such as CIM_Sensor. PLDM supports additional types of units because certain types of sensors or effecters may be used within a platform management subsystem but are not exposed through CIM, or are mapped into CIM using proprietary CIM extensions. Parties developing platform management subsystems in which sensors are intended to be exposed as CIM objects should first verify which types of sensors and units are supported by CIM and the CIM profiles.

27.4.4 OEM (Vendor-Defined) Sensor Units

OEM (vendor-defined) sensor units are identified in PLDM sensor PDRs when the OEMUnit value from Table 62 is used for the baseUnit or auxUnit. The semantic information of an OEMUnit can then be further described using an OEM Sensor Units PDR that is associated with the particular sensor that is returning the OEMUnit. Multiple OEM Sensor Units PDRs can be defined if there is a need for defining more than one type of OEM unit. Additionally, multiple PLDM Sensor PDRs can be associated with a particular OEM Sensor Units PDR.

27.5 Counters

A counter is a numeric sensor that returns a value that returns a count. PLDM does not define any requirements on whether a counter must increment, decrement, or both, or whether it does so sequentially or monotonically, and so on.

Many common types of counters can use predefined sensor unit values, such as Hits, Misses, Corrected Errors, Uncorrected Errors, and others. If no predefined unit fits, it is recommended that the auxiliary sensor unit (auxUnit) be designated using the predefined unit "Counts" in the PDR for the sensor, and that an OEM unit type is defined for the base unit.

For example, if an implementation needed a counter for "widgets," it would be noted that no predefined sensor unit type for "widgets" exists. In this case, an OEM Unit PDR for "widgets" is created and used for the base unit type, and "Counts" is used as the auxUnit.

Counters enable a party that accesses PDR information for the sensor to get a partial interpretation of the sensor semantics. Thus, although the party interpreting the sensor may not know what a widget is, it will know that the sensor is returning Counts of something.

27.6 Accuracy, Tolerance, Resolution, and Offset

The PDRs for numeric sensors and effecters include fields for reporting the accuracy, tolerance, and resolution associated with the numeric value for the reading or setting. This section provides definitions for accuracy, tolerance, and resolution as used within this specification and information on how the values are calculated and used. Accuracy, tolerance, and resolution are summarized as follows:

Accuracy An error in the reading that scales proportionally with the magnitude of the input. Typically given as a ± percentage of the reading.

Tolerance A ± error in the reading that, unlike accuracy, does not scale with the magnitude of the reading. Tolerance typically comes from a combination of quantization (round off) errors including errors due to offsets in the measurement.

Resolution The nominal size of the “steps” between sequential reading values.

Accuracy specifies a degree of error that varies in proportion to the reading, and tolerance specifies a constant error. The combination of these two generally provides enough flexibility to cover a range of conversion errors in most linear analog-to-digital (A/D) converters.

Although other error types, such as non-linearity, can exist in converters, the contribution of those errors can be accounted for by increasing the size of the reported values for tolerance, accuracy, or both as necessary.

27.6.1 Additional Information about Numeric Sensor / Effecter Tolerance

Tolerance can be considered to be a constant portion of the quantization error in the conversion of an analog input to a numeric sensor. Consider a sensor where 0x00 ideally corresponds to 0.000 to 0.500 V and 0x01 corresponds to 0.500 V to 1.000 V. When the input is 0.500 V exactly, the sensor could either report 0x00 or 0x01. Now assume that the input is 0.501 V. Ideally, this would result in a value of 0x01 from the sensor, but because of offsets in an implementation, it is possible that some implementations could return either a value of 0x00 or 0x01. If 0x00 is reported, the sensor is effectively returning a value that is -1 count from ideal. It is possible that the sensor implementation could be asymmetric with respect to tolerance. For example, a sensor implementation may sometimes map 0.501 V to 0x00, but would never map anything less than 0.500 V to 0x01. In this case, the tolerance would be +0 counts and -1 counts. Generally, an implementation is subject to both positive and negative offsets because of component manufacturing variation, noise, and so on. Thus, it is common to see a tolerance of ± 1 count.

27.6.2 Examples of Accuracy, Tolerance, and Resolution Use

Figure 22 shows an example of a "3-bit" (eight step) converter. In this example, the converter is hooked up for monitoring a nominal signal that can vary from 0.0 V to 8.0 V. The resolution is defined as the size of the steps between nominal readings. The resolution is 1.0 V because there is 1.0 V difference between each successive reading value.

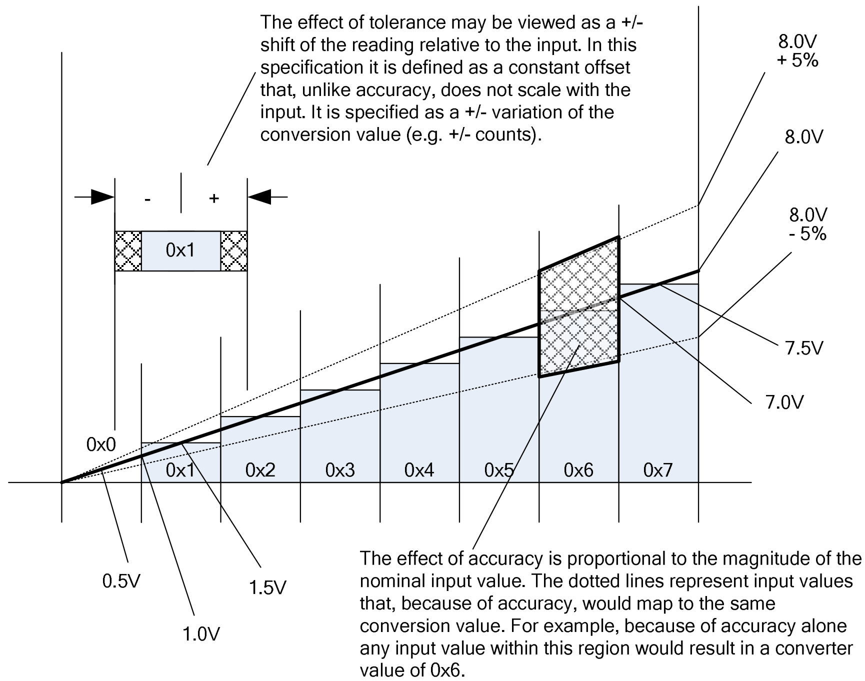


Figure 22 – Accuracy, Tolerance, and Resolution Example

In this example, the input value that corresponds to a reading of 0x0 is actually centered around 0.50 V, not 0.0 V. That is, the meaning of a reading of 0x0 does not mean 0.0 V, as might be expected, but actually means "0.5 V plus or minus 0.5 V". This represents a typical way that A/D converters are connected in systems. It is a common mistake to assume that a reading of zero actually corresponds to 0.0 V.

If this converter had no additional offsets or accuracy errors, the reading values would correspond to input values as follows:

0x0 → 0 V to 1.0 V (0.5 V ± 0.5 V)

0x1 → 1.0 V to 2.0 V (1.5 V ± 0.5 V)

0x2 → 2.0 V to 3.0 V (2.5 V ± 0.5 V)

0x3 → 3.0 V to 4.0 V (3.5 V ± 0.5 V)

0x4 → 4.0 V to 5.0 V (4.5 V ± 0.5 V)

0x5 → 5.0 V to 6.0 V (5.5 V ± 0.5 V)

0x6 → 6.0 V to 7.0 V (6.5 V ± 0.5 V)

0x7 → 7.0 V to 8.0 V (7.5 V ± 0.5 V)

If these readings were converted to their corresponding nominal input voltage (Vin) values, the formula would be as follows:

Vin(nominal) → (resolution * reading) + 1/2 resolution

Note that this follows the Cartesian coordinate formula for a line: y = Mx + B

Now, suppose that the implementation could add a negative D.C. offset of 0.5 V to the input. Then the center point for a reading of 0.0 V would correspond to 0.0 V, and a reading of 0x0 would correspond to a range of 0.0 V ± 0.5 V instead of 0.0 V to 1.0 V. In this case, the conversion would then be V = (resolution * reading) + 0.0 V. There is now no offset relative to the center of the reading value because of a D.C. offset. If the converted negative offset of 4.0 V was connected to the input, a reading of 0x0 would now correspond to -3.5 V ± 0.5 V and a reading of 111b would correspond to 3.5 V ± 0.5 V.

It is very common for an A/D converter implementation to have a D.C. offset that needs to be accounted for when converting a reading to the corresponding nominal input value. The party that implements the hardware for the sensor needs to provide this offset value as well at the resolution (step size per count) so that the basic conversion of the reading can be accomplished.

After the basic conversion of the reading is done, the effects of accuracy and tolerance may need to be taken into account. For example, if someone is depending on the reading to determine whether something has failed, it is important to understand how much error might be in the reading so that a failure is not falsely assessed for a healthy component.

For PLDM, the effects of accuracy and tolerance are considered to be orthogonal to one another and additive. First consider the effect of accuracy. Suppose the accuracy of the sensor is specified as ±5%. Using that figure, a value of 001b will nominally correspond to 1.5 V ± 5%, but because of quantization and accuracy, any value from 1.0 V ± 5% to 2.0 V ± 5% (a range of 0.95 V to 2.10 V) could result in a reading of 0x1.

The next step is to factor in tolerance. The quantization within a converter is never perfect; some slight variation always exists in the comparison points that yield a particular converter output. Instead of the conversion ranges being evenly spaced as shown in Figure 22, some ranges may be a little wider and others a little narrower. The effect of this is that in an actual implementation, borderline values such as 1.99 V or 2.01 V, for example, may sometimes yield a value of 0x1 and sometimes 0x2.

Tolerance in PLDM is defined as an error in the quantization that is applied to all counts of the converter equally. Because PLDM sensors are all specified as returning integer values, any errors in the reading will always result in an integral number of counts. Thus, tolerance is specified as a +/- effect on the count.

The tolerance value is typically used to account for quantization errors in A/D conversion circuitry that occur because of effects such as D.C. voltage offsets within the circuit. For example, suppose the input to an A/D converter that monitors voltage was shifted up by a constant amount, as would be the case if a D.C. offset was added to the input. Per the figure, if a D.C. offset error of 0.25 V were added when converting, the input reading 0x01 would represent a range that actually goes from 0.75 V to 1.75 V instead of the nominal range 1.0 V to 2.0 V. This means that an input between 0.75 V and 1.0 V will cause a reading of 0x1 to be returned instead 0x0. Thus, because of this offset error, the reading would be one count higher than it was intended to be for inputs in that range. Similarly, with the same offset, a reading of 0x2 would correspond to an input of 1.75 V to 2.75 V, and so an input between 1.75 V and 2.00 V would also result in a reading that is one count higher than intended.

This does not mean that all conversions are off by one count. In this example, the reading is incorrect only for inputs that are in the range caused by the offset. A reading of 0x1 would be correctly returned for an input of 1.5 V. The reading can thus be incorrect by 0 counts or +1 counts depending on what range the input value is in. In this case, the tolerance would be specified as +1/-0 counts.

Manufacturing variations and tolerances in A/D conversion circuitry mean that both positive and negative offsets are possible. This is why it is typical to see a specification of ± 1 count for tolerance. In many implementations, tolerance is specified as ± 1 count for these types of conversions. Because resolution is given in units of 1 count, tolerance and resolution may sometimes appear to equate to the same value. However, tolerance and resolution should not be misinterpreted as being the same thing.

Lastly, in some cases PLDM Numeric Sensors will return values such as counts or other measurements that to not use a conversion process that can introduce errors in the reading. In this case, the tolerance is specified as ± 0 counts.

27.6.3 Accuracy, Tolerance, and Resolution Relationship to Thresholds

Accuracy, tolerance, and resolution must all be taken into account to generate a threshold that does not generate a "false positive" (a false indication of a failure). For example, if accuracy, tolerance, and resolution are not taken into account when calculating the threshold for a warning level, it is possible that an input could be assessed as being within the warning range when the input was actually near the limit of the normal range.

A consequence of avoiding false positives is that for a particular range a value that is actually within the intended warning range can be assessed as being within the normal range. That is, false positives are avoided at the cost of having the possibility of 'false negatives'. However, in most implementations it is considered better to avoid the false alarms that false positives would cause. Whether to design thresholds to avoid false positives or false negatives is a choice of the system implementation.

Because it is the more common case, the following examples describe how thresholds may be calculated to avoid false positives.

EXAMPLE: An 8-bit A/D converter monitoring a 5.0 V nominal signal where the sensor has been designed such that the 5.0 V level corresponds to a reading of C0h and the 0.0 V level corresponds to a reading of 00h (as shown by Figure 23A). Assume the converter implementation has a specified worst-case accuracy of ± 4%, and a tolerance of ± 1 count.

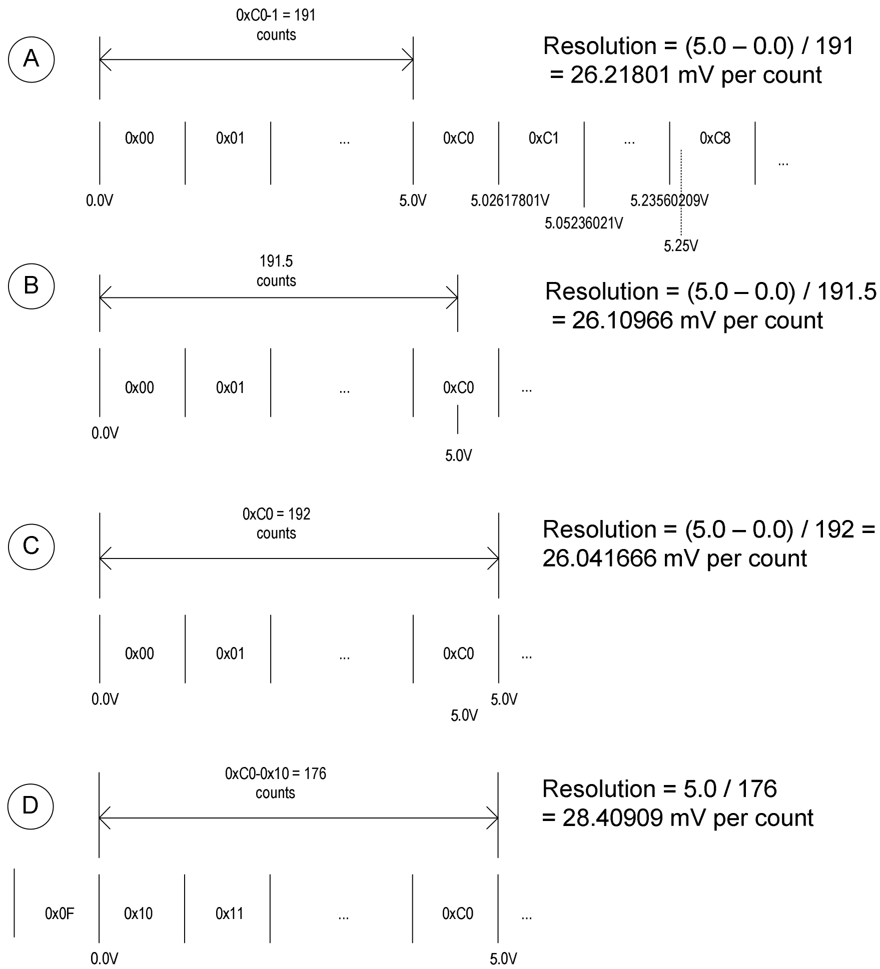


Figure 23 – Figuring Resolution from the Design

For Figure 23A, this yields resolution, tolerance, and accuracy values as follows:

Resolution

= 5.0 V / (C0h -1) = 26.17801 mV

Accuracy

= ± 4% (given, from the design)

Tolerance

= ± 1 count (given) = ± 26.17801 mV

Now, suppose it is necessary to calculate an upper critical threshold for the 5.0 V + 5% point (5.25 V) where this threshold will not produce "false positives" (falsely return 'critical') across the range of accuracy, tolerance, and resolution. The following example shows steps that can be used to calculate a threshold suitable for a PLDM Numeric Sensor:

Step 1: Divide the target threshold value by the resolution to find how many counts correspond to 5.25 V:

5.25 V / 26.17801 mV = 200.55 counts
(which puts the 5.25 V point within the nominal range of reading 0xC8, as shown in Figure 23A)

Step 2: Factor in the tolerance:

Important: Because tolerance is specified as an error, a "+" count for tolerance means that the reading may be higher than it should be, and a "-" count means that the reading may be lower than it should be. To account for these errors, the "-" tolerance value should be added to upper thresholds, and the "+" tolerance value subtracted from lower thresholds. This is particularly important when the plus and minus tolerance values are different from one another.

200.55 + 1 = 201.55 counts

Step 3: Account for the effect of accuracy:

201.55 * 1.04 = 209.612 counts

Step 4: Round up (because an A/D converter cannot give a non-integer count)

209.612 → 210 counts = 0xD2

This yields a Threshold value of 210 which corresponds to 5.497 V. This shows that even though a threshold of 5.25 V is being targeted, it is necessary to set the threshold to a value that, because of the effects of accuracy, tolerance, and resolution, could allow the actual monitored value to be as high as 5.497 V in some implementations before a threshold match would be detected.

The calculations for lower thresholds are the same, except that negative values for the accuracy, tolerance, and resolution are used.

Figure 23 illustrates what to be aware of when deriving the values for resolution from an implementation. To get an accurate value for resolution, it is important to know whether the input values that correspond to a particular reading are given as values that are at the point of change (quantization point) between successive readings, are a nominal “center point” of a reading, or a combination of the two. (The difference in the resolution value between Figure 23A and Figure 23C is almost 0.5%. This shows that a non-trivial amount of error could be introduced if the implementer uses the wrong calculation point for its implementation).

Lastly, area D in Figure 23 shows that offsets in the implementation also need to be taken into account. Offset adds a new first step to the threshold calculation:

Step 0: Take the target threshold and subtract (or add, depending on the implementation) the D.C. offset value before calculating the counts for the threshold.

27.7 Numeric Reading Conversion Formula

The following formula is used with data from the Numeric Sensor PDR to convert the corresponding PLDM Numeric Sensor's raw reading to the units specified in the Numeric Sensor PDR.

Reading Conversion formula: Y = (m * X + B)

Where:

Y = converted reading in Units

X = reading from sensor

m = resolution from PDR in Units

B = offset from PDR in Units

Units = sensor/effecter Units, based on the Units and auxUnits fields from the PDR for the numeric sensor

For example, a sensor with the following units, resolution, offset, and reading:

Reading = 0xBF

Units = Volts

Resolution: 26.17801 mV

Offset = -1.00 V

would have the following the converted reading:

$Y = (26.17801 * 10^{-3}$ V * 0xBF + (-1.00 V)) = [(.02617801 * 191) - 1.00] V = 4.00 V

A full interpretation of the reading should also take tolerance and accuracy into account. For example, if the PDR indicates the following:

Accuracy: ± 4%

Tolerance: ± 1 count (given)

combined with the previous example, the full interpretation of the reading would be:

(4.00 V ± 26.17801 mV) ± 4%

where ± 26.17801 mV corresponds to the effect of a Tolerance of ± 1 count.

27.7.1 Rounding

Some precision may often be lost in the conversion of binary to decimal. For example, the previous conversion that was shown as 4.00 V actually calculates out to 3.99999991 V using the given value for the resolution, but the result was rounded up to 4.00. This raises a question about how much rounding should be applied, or how many digits of precision should be used for a converted value.

The number of digits of precision for the converted value can be based on the overall size of the binary number. For example, an eight-bit unsigned value has a range of 0 to 255, which is three decimal digits. Thus, rounding the converted reading to three significant digits is appropriate.

27.8 Numeric Effecter Conversion Formula

A reverse process from that used to convert a sensor reading is used to generate the raw value to be set into a PLDM Numeric Effecter. In this case, the formula is as follows:

Setting Conversion formula: X = Round [(Y - B) / m]

Where:

- X = integer setting value for the effecter
- Y = target setting in Units
- m = resolution from PDR in Units
- B = offset from PDR in Units
- Round = rounding operation to round the value in [ ] to the nearest integer value
- Units = sensor/effecter Units, based on the Units and auxUnits fields from the Numeric Effecter PDR

28 Platform Descriptor Record (PDR) Formats

This section defines the content and format of the PDRs that are used for supporting sensor monitoring and control in PLDM.

28.1 Common PDR Header Format

All PDRs have a common, fixed format header followed by variable length record data. The size and definition of the bytes within the PDR data field are specific to each PDR Type. Table 63 describes the format of the common PDR header.

The PDR data length can vary on a per record basis. It is generally recommended that the definition of PDRs of a given type use a fixed length when practical.

The header fields are not shown in the succeeding PDR format sections.

Table 63 – Common PDR Header Format

Type	PDR Fields
uint32	**recordHandle** An opaque number that is used for accessing individual PDRs within a PDR Repository. The PDR Handle value is required to be unique for all PDRs within a PDR Repository. PDR Handle values are not required to be unique across PDR Types or across other PDRs in the system. See 26.2.3 for more information. special value: {0x0000_0000 = reserved }
uint8	**PDRHeaderVersion** This field is provided in case a future version of this specification requires a modification to the format of the PDR Header. Any PDR fields that follow this field are eligible for change. value: The value 0x01 shall be used as the PDRHeaderVersion for PDRs that are defined in this specification.
uint8	**PDRType** The type of the PDR. See 25.3 and 28.2.

Type	PDR Fields
uint16	**recordChangeNumber** See 26.2.3 for more information.
uint16	**dataLength** The total number of PDR data bytes following this field.

28.2 PDR Type Values

Table 64 lists the different types of PDRs defined in this document and the corresponding PDR Type values used for those PDRs. Unspecified values are reserved for future definition by this specification.

Table 64 – PDR Type Values

PDR Type Number	PDR Type Name	Reference
1	Terminus Locator PDR	See 28.3.
2	Numeric Sensor PDR	See 28.4.
3	Numeric Sensor Initialization PDR	See 28.5.
4	State Sensor PDR	See 28.6.
5	State Sensor Initialization PDR	See 28.7.
6	Sensor Auxiliary Names PDR	See 28.8.
7	OEM Unit PDR	See 28.9.
8	OEM State Set PDR	See 28.10.
9	Numeric Effecter PDR	See 28.11.
10	Numeric Effecter Initialization PDR	See 28.12.
11	State Effecter PDR	See 28.13.
12	State Effecter Initialization PDR	See 28.14.
13	Effecter Auxiliary Names PDR	See 28.15.
14	Effecter OEM Semantic PDR	See 28.16.
15	Entity Association PDR	See 28.17.
16	Entity Auxiliary Names PDR	See 28.18.
17	OEM Entity ID PDR	See 28.19.
18	Interrupt Association PDR	See 28.20.
19	PLDM Event Log PDR	See 28.21.
126	OEM Device PDR	See 28.22.
127	OEM PDR	See 28.23.

28.3 Terminus Locator PDR

The Terminus Locator PDR provides information that associates a PLDMTerminusHandle with values that uniquely identify the device or software that contains the PLDM terminus. Table 65 describes the format of this PDR.

Table 65 – Terminus Locator PDR Format

Type	Description
–	**commonHeader** See 28.1.
uint16	**PLDMTerminusHandle** A handle that identifies PDRs that belong to a particular PLDM terminus.
enum8	**validity** Indicates whether the PDR contains valid information for the terminus. This is also used as part of identifying (enumerating) which termini are present. See 12.5 for more information. value: { notValid, // The PDR should be ignored. valid // The PDR is valid. **}**
uint8	**TID** PLDM Terminus ID. This value is used to identify asynchronous messages from a given terminus.
uint16	**containerID** The containerID for the containing entity that holds this terminus. See 9.1 for more information.

enum8	**terminusLocatorType**
	value: {
	UID,
	MCTP_EID,
	SMBusRelative, // Used when the device has a fixed slave address and bus connection // that is relative to a device that is identified through a UID (for example, // if the terminus was an SMBus device on an add-in card and was // located on bus #3 of another device on that same add-in card that had // a UID)
	systemSoftware // Used when the terminus is a software or firmware agent that is running // under the host processors of the managed system
	}

enum8	**terminusLocatorValueSize** Size of the following terminusLocatorValue, in bytes. NOTE: This helps facilitate backward compatibility in case terminusLocatorTypes get extended. The combination of terminusLocatorType and all fields of the terminusLocatorValue is persisent and unique for a given terminus in PLDM.
terminusLocatorValue for terminusLocatorType = UID:	
uint8	**terminusInstance** This field is used to differentiate between different PLDM termini if the device contains more than one PLDM terminus.

Type	Description
UUID	**deviceUID** Although using the UUID format, the value may not be universally unique among different platforms. For example, a device manufacturer could assign the same value to all the devices of a particular type that it manufactures, provided that only one instance of that device would be used within a given PLDM implementation. Similarly, a device manufacturer could manufacture a device that contains a set of UUIDs and provide a mechanism such as configuration pins or non-volatile memory that would enable one UUID from the set to be selected when the device was integrated into the system. The value may also be derived from another UID or UUID, such as the unique ID for the device containing the terminus, a UUID for the overall system, and so on. A PLDM terminus that is identified using this type of ID must support the GetTerminusUID command.
terminusLocatorValue for terminusLocatorType = MCTP_EID:	
uint8	**EID** A MCTP EID that is assigned to an MCTP Endpoint that provides the transport protocol termination for a PLDM terminus
terminusLocatorValue for terminusLocatorType = SMBusRelative	
UUID	**UID** A UID for the controller that owns the bus to which the device is connected. For more information, see the preceding description for "*terminusLocatorType* = UID".
uint8	**busNumber** A bus number for the bus to which the device is connected, relative to the controller that owns the bus. If the PLDM terminus is accessed through an MCTP Endpoint, the busNumber must be the port number used in the routing table for accessing the endpoint.
uint8	**slaveAddress** The SMBus or I^2C slave address for the device that is providing the [7:1] - SMBus or I^2C slave address value. [0] - 0b.
terminusLocatorValue for terminusLocatorType = systemSoftware	

enum8	**softwareClass**
	{
	unspecified, other, systemFirmware, OSloader, OS, CIMprovider, otherProvider, virtualMachineManager
	}

UUID	**UUID** A UID for the software or instance of software that is acting as a PLDM terminus. This ID is required to be unique for the particular instance of software within the system that is providing or emulating a PLDM terminus within a single PLDM platform management subsystem implementation. For example, a software application running on a platform may emulate sensors for the purpose of generating events to be handled by PLDM. This piece of software can be assigned a fixed UUID by the software vendor that is used to identify it as a unique PLDM terminus. If multiple instances of that software could exist on the platform where each instance individually provides an emulation of a PLDM terminus, each instance must have a different UUID. Similarly, if a common piece of software implements multiple PLDM termini, each terminus must have a different UUID.

28.4 Numeric Sensor PDR

The Numeric Sensor PDR is primarily used to describe the semantics of a PLDM Numeric Sensor to a party such as a MAP. It also includes the factors that are used for converting raw sensor readings to normalized units. The record also identifies the Entity that is being monitored by the sensor. Table 66 describes the format of this PDR.

Table 66 – Numeric Sensor PDR Format

Type	Description
–	**commonHeader** See 28.1.
uint16	**PLDMTerminusHandle** A handle that identifies PDRs that belong to a particular PLDM terminus.
uint8	**sensorID** ID of the sensor relative to the given PLDM Terminus ID.
uint16	**entityType** The Type value for the entity that is associated with this sensor. See 9.1 for more information.
uint16	**entityInstanceNumber** The Instance Number for the entity that is associated with this sensor. See 9.1 for more information.
uint16	**containerID** The containerID for the containing entity that is associated with this sensor. See 9.1 for more information.
enum8	**sensorInit** Indicates whether the sensor requires initialization by the initializationAgent. value: { noInit, // The Initialization Agent does not take any steps to initialize, enable, // or disable this particular sensor. useInitPDR, // The sensor has an associated Numeric Sensor Initialization PDR // that should be used to initalize the sensor. enableSensor, // Whenever the Initialization Agent runs, it will enable this sensor // using a SetNumericSensorEnable command to set the // operationalState. disableSensor. // Whenever the Initialization Agent runs, it will disable this sensor by // using the SetNumericSensorEnable command. }
bool8	**sensorDescriptionPDR** true = sensor has a sensorDescription PDR false = sensor does not have an associated sensorDescription PDR
enum8	**baseUnit** The base unit of the reading returned by this sensor. See 27.1 for more information. value: { see Table 62 }
sint8	**unitModifier** A power-of-10 multiplier for the baseUnit. See 27.1 for more information.

Type	Description
enum8	**rateUnit** value: { None, Per MicroSecond, Per MilliSecond, Per Second, Per Minute, Per Hour, Per Day, Per Week, Per Month, Per Year }
uint8	**baseOEMUnitHandle** This value is used to locate the corresponding PLDM OEM Unit PDR that defines the OEMUnit when the OEMUnit value is used for the baseUnit.
enum8	**auxUnit** The base unit of the reading returned by this sensor. See 27.2 for more information. value: { see Table 62 }
sint8	**auxUnitModifier** A power-of-10 multiplier for the auxUnit. See 27.2 for more information.
enum8	**auxrateUnit** value: { None, Per MicroSecond, Per MilliSecond, Per Second, Per Minute, Per Hour, Per Day, Per Week, Per Month, Per Year }
uint8	**auxOEMUnitHandle** This value is used to locate the PLDM OEM Unit PDR that defines the OEMUnit if the OEMUnit value is used for the auxUnit.
bool8	**isLinear** This value is used to provide information that can be used by a MAP to populate the IsLinear attribute of CIM_NumericSensor. Currently, the CIM_NumericSensor description of this field is "Indicates that the Sensor is linear over its dynamic range." value: This field is typically set to "true".
enum8	**sensorDataSize** The bit width and format of reading and threshold values that the sensor returns value: { uint8, sint8, uint16, sint16, uint32, sint32 }
real32	**resolution** The resolution of the sensor in Units (see 27.7).
real32	**offset** A constant value that is added in as part of the conversion process of converting a raw sensor reading to Units (see 27.7).
uint16	**accuracy** Given as a +/- percentage in 1/100ths of a % from 0.00 to 100.00. For example, the integer value 510 corresponds to ± 5.10%. See 27.6 for more information.
uint8	**plusTolerance** Tolerance is given in +/- counts of the reading value. It indicates a constant magnitude possible error in the quantization of an analog input to the sensor. It is possible that the tolerance could be asymmetric. The plusTolerance field provides the '+' value of the tolerance; the minusTolerance field provides the minus portion. For example, if plusTolerance is 0x02 and minusTolerance is 0x00, the tolerance is +2/-0 counts. See 27.6 for more information about how tolerance is defined and used.

Type	Description
uint8	**minusTolerance** Tolerance is given in +/- counts of the reading value. It indicates a constant magnitude possible error in the quantization of an analog input to the sensor. It is possible that the tolerance could be asymmetric. The plusTolerance field provides the '+' value of the tolerance; the minusTolerance field provides the minus portion. For example, if plusTolerance is 0x02 and minusTolerance is 0x00, the tolerance is +2/-0 counts. See 27.6 for more information about how tolerance is defined and used.
uint8 | sint8 | uint16 | sint16 | uint32 | sint32	**hysteresis** The amount of hysteresis associated with the sensor thresholds, given in raw sensor counts. See 17.9 for more information. This value may be overridden if the sensor supports the SetSensorThresholds command. The size of this field is identified by sensorDataSize. value: 1 or greater special value: 0 = sensor does not use hysteresis
bitfield8	**supportedThresholds** For PLDM: bit field where bit position represents whether a given threshold is supported 0x1b = threshold is supported 0x0b = threshold is not supported [6:7] – reserved [5] – lowerThresholdFatal [4] – lowerThresholdCritical [3] – lowerThresholdWarning [2] – upperThresholdFatal [1] – upperThresholdCritical [0] – upperThresholdWarning
bitfield8	**thresholdAndHysteresisVolatility** Identifies under which conditions any threshold or hysteresis settings that were set through the SetSensorThresholds or SetSensorHysteresis command may be lost. The threshold values either return to default values or will require reinitialization through the Initialization Agent function. special value: 00000b = non-volatile. The threshold settings retained indefinitely regardless of system state. [7:5] – reserved [4] – 1b = PLDM terminus returns to online condition [3] – 1b = System warm resets [2] – 1b = System hard resets [1] – 1b = PLDM subsystem power up [0] – 1b = Initialization Agent controller restart/update (initialize/reinitialize this sensor whenever the device that holds the Initialization Agent has been restarted or reinitialized)

Type	Description
real32	**stateTransitionInterval** How long the sensor device takes to do an enabledState change (worst case), in seconds. NOTE: Because this is floating point format, fractional seconds can be represented. The real32 format also supports a value for 'unknown'.
real32	**updateInterval** Polling or update interval in seconds expressed using a floating point number (generally corresponds to the CIM PollingInterval property)
uint8 | sint8 | uint16 | sint16 | uint32 | sint32	**maxReadable** The maximum value that the sensor may return. The size of this field is given by the sensorDataSize field in this PDR. This number is given in the same format as the reading returned by the sensor. The conversion formula is used to convert this number to normalized units. See 27.7.
uint8 | sint8 | uint16 | sint16 | uint32 | sint32	**minReadable** The minimum value that the sensor may return. The size of this field is given by the sensorDataSize field in this PDR. This number is given in the same format as the reading returned by the sensor. The conversion formula is used to convert this number to normalized units. See 27.7.
enum8	**rangeFieldFormat** Indicates the format used for the following nominalReading, normalMax, normalMin, criticalMax, criticalMin, fatalMax, and fatalMin fields. value: { uint8, sint8, uint16, sint16, uint32, sint32, real32 }
bitfield8	**rangeFieldSupport** Indicates which of the fields that identify the operating ranges of the parameter monitored by the sensor are supported. (This bitfield indicates whether the following nominalValue, normalMax, and so on, fields contain valid range values.) [7] – reserved [6] – 1b = fatalLow field supported [5] – 1b = fatalHigh field supported [4] – 1b = criticalLow field supported [3] – 1b = criticalHigh field supported [2] – 1b = normalMin field supported [1] – 1b = normalMax field supported [0] – 1b = nominalValue field supported

Type	Description
uint8 | sint8 | uint16 | sint16 | uint32 | sint32 | real32	**nominalValue** This value presents the nominal value for the parameter that is monitored by the sensor. The size of this field is given by the rangeFieldFormat field in this PDR. This value is given directly in the specified units without the use of any conversion formula. For example, if the units are millivolts and the nominalValue is 5000, the nominalValue corresponds to 5000 mV, or 5.000 V. It is possible that the nominal value could be some fraction of the given units for the sensor (for example, if the units are volts and the nominal value is 2.5 V). For this reason, the nominalValue can be expressed using a real32. The value is defined as the nominal value for what is being monitored. Thus, nominalValue is not required to match a value that can be returned as a reading by the sensor implementation. For example, if the nominal value for a given monitored voltage is 5.00 V, the nominalValue would typically be reported as 5.00 V even though the closest reading the sensor implementation may be able to return is 5.05 V. A common use of the nominalValue is as a source of part of an identifying 'name' for a sensor. For example, it is common for voltage sensors to be identified by their nominal reading. So, a sensor with a nominal reading of +5.00 V would be referred to as a "+5 V sensor", while one with a nominal reading of +3.3 V would be referred to as a "+3.3 V sensor". The definition of nominalValue in the PDR supports this usage. An application that uses or displays this value will typically elect to round the value to some number of significant digits using an algorithm based on the resolution of the sensor. For example, if the nominalValue is given as a real32 as 2.50000 V, but the resolution of the sensor is 0.05 V, the nominalValue displayed would typically be rounded as 2.50 V. It is possible that a given sensor may not be considered as having a nominal reading, in which case this field should be ignored. For example, a numeric sensor that tracks a count or size of some parameter may not be considered as having a nominal reading depending on its application.
uint8 | sint8 | uint16 | sint16 | uint32 | sint32 | real32	**normalMax** The upper limit of the normal operating range for the parameter that is monitored by the numeric sensor. The monitored parameter is considered to be operating outside of normal range when this value is exceeded. For example, if a monitored voltage of a component is specified in its data sheet to have a normal maximum operating range of 4.75 to 5.25 V, this value would be set to 5.25 (assuming the units in the PDR are for “volts”). This value is given directly in the specified units without the use of any conversion formula. This value is used together with normalMin to indicate the normal operating range for the sensor.
uint8 | sint8 | uint16 | sint16 | uint32 | sint32 | real32	**normalMin** The lower limit of the normal operating range for the parameter that is monitored by the numeric sensor. Sensor thresholds are typically set for a value that is lower than normalMin to accommodate the affects of sensor accuracy, tolerance, and resolution, in order to prevent false reporting of an “out-of-range” event state. This value is given directly in the specified units without the use of any conversion formula.
uint8 | sint8 | uint16 | sint16 | uint32 | sint32 | real32	**warningHigh** A warning condition that occurs when the monitored value is *greater than* the value reported by warningHigh. In many implementations, this value may be the same value as normalMax. Sensor thresholds that may be derived from this value are typically set for a value that is higher than warningHigh to accommodate the affects of sensor accuracy, tolerance, and resolution, in order to prevent false reporting of an out-of-range condition. This value is given directly in the specified units without the use of any conversion formula.

Type	Description
uint8 | sint8 | uint16 | sint16 | uint32 | sint32 | real32	**warningLow** A warning condition that occurs when the monitored value is *less than or equal to* the value reported by warningLow. In many implementations, this value may be the same value as normalMin. Sensor thresholds that may be derived from this value are typically set for a value that is lower than warningLow to accommodate the affects of sensor accuracy, tolerance, and resolution, in order to prevent false reporting of an out-of-range condition. This value is given directly in the specified units without the use of any conversion formula.
uint8 | sint8 | uint16 | sint16 | uint32 | sint32 | real32	**criticalHigh** A critical condition that occurs when the monitored value is *greater than or equal to* the value reported by criticalHigh. In some implementations, this value may be the same value as normalMax. Sensor thresholds that may be derived from this value are typically set for a value that is higher than criticalHigh to accommodate the affects of sensor accuracy, tolerance, and resolution, in order to prevent false reporting of an out-of-range condition. This value is given directly in the specified units without the use of any conversion formula.
uint8 | sint8 | uint16 | sint16 | uint32 | sint32 | real32	**criticalLow** A critical condition that occurs when the monitored value is *less than* the value reported by criticalLow. In some implementations, this value may be the same value as normalMin. Sensor thresholds that may be derived from this value are typically set for a value that is lower than criticalLow to accommodate the affects of sensor accuracy, tolerance, and resolution, in order to prevent false reporting of an out-of-range condition. This value is given directly in the specified units without the use of any conversion formula.
uint8 | sint8 | uint16 | sint16 | uint32 | sint32 | real32	**fatalHigh** A fatal condition that occurs when the monitored value is *greater than* the value reported by fatalHigh. In many implementations, this value may be the same value as normalMax. Sensor thresholds that may be derived from this value are typically set for a value that is higher than fatalHigh to accommodate the affects of sensor accuracy, tolerance, and resolution, in order to prevent false reporting of an out-of-range condition. This value is given directly in the specified units without the use of any conversion formula.
uint8 | sint8 | uint16 | sint16 | uint32 | sint32 | real32	**fatalLow** A fatal condition that occurs when the monitored value is *less than* the value reported by fatalLow. In many implementations, this value may be the same value as normalMin. Sensor thresholds that may be derived from this value are typically set for a value that is lower than fatalLow to accommodate the affects of sensor accuracy, tolerance, and resolution, in order to prevent false reporting of an out-of-range condition. This value is given directly in the specified units without the use of any conversion formula.

28.5 Numeric Sensor Initialization PDR

The Numeric Sensor Initialization PDR is used when a PLDM Numeric Sensor requires initialization by a PLDM Initialization Agent. Table 67 describes the format of this PDR.

Table 67 – Numeric Sensor Initialization PDR Format

Type	Description
–	**commonHeader** See 28.1.
uint16	**PLDMTerminusHandle** A handle that identifies PDRs that belong to a particular PLDM terminus

Type	Description
uint16	**sensorID** ID of the sensor relative to the given PLDM Terminus ID
bitfield8	**initConditions** Identifies under which conditions the Initialization Agent must initialize or reinitialize this sensor [7:5] – reserved [4] – 1b = PLDM terminus returns to online condition [3] – 1b = System warm resets [2] – 1b = System hard resets [1] – 1b = PLDM subsystem power up [0] – 1b = Initialization Agent controller restart/update (initialize/reinitialize this sensor whenever the device that holds the Initialization Agent has been restarted or reinitialized)
enum8	**sensorEnable** The operational state that the sensor is to be left in after it has been initialized. This state is written to the sensor sensorOperationalState using the SetNumericSensorEnable command. special value: { 0xFF = do not change the sensorOperationalState }
bitfield8	**thresholdInitMask** Indicates which thresholds should be initialized NOTE: Be careful to match the bit up with the correct threshold. [7:6] – reserved [5] – 1b = initialize lowerThresholdFatal threshold [4] – 1b = initialize lowerThresholdCritical threshold [3] – 1b = initialize lowerThresholdWarning threshold [2] – 1b = initialize upperThresholdFatal threshold [1] – 1b = initialize upperThresholdCritical threshold [0] – 1b = initialize upperThresholdWarning threshold
enum8	**sensorDataSize** The bit width of reading and threshold values that the sensor returns value: { uint8, sint8, uint16, sint16, uint32, sint32 }
uint8 | sint8 | uint16 | sint16 | uint32 | sint32	**upperThresholdWarning** This value is given in raw units for the sensor. The size of this field is given by the sensorDataSize field in this PDR.
uint8 | sint8 | uint16 | sint16 | uint32 | sint32	**upperThresholdCritical** This value is given in raw units for the sensor.The size of this field is given by the sensorDataSize field in this PDR.
uint8 | sint8 | uint16 | sint16 | uint32 | sint32	**upperThresholdFatal** This value is given in raw units for the sensor.The size of this field is given by the sensorDataSize field in this PDR.

Type	Description
uint8 | sint8 | uint16 | sint16 | uint32 | sint32	**lowerThresholdWarning** This value is given in raw units for the sensor.The size of this field is given by the sensorDataSize field in this PDR.
uint8 | sint8 | uint16 | sint16 | uint32 | sint32	**lowerThresholdCritical** This value is given in raw units for the sensor.The size of this field is given by the sensorDataSize field in this PDR.
uint8 | sint8 | uint16 | sint16 | uint32 | sint32	**lowerThresholdFatal** This value is given in raw units for the sensor.The size of this field is given by the sensorDataSize field in this PDR.

28.6 State Sensor PDR

The State Sensor PDR provides the sensorID for a composite state sensor within a PLDM terminus and the number of sensors, and the state set and the possible state values for each sensor that is accessed through the given sensorID. The record also identifies the entity that is being monitored by the sensor. Only one set of fields exists for the entity identification information. Therefore, all sensors in this record must be associated with the same entity. Table 68 describes the format of this PDR.

Table 68 – State Sensor PDR Format

Type	Description
–	**commonHeader** See 28.1.
uint16	**PLDMTerminusHandle** A handle that identifies PDRs that belong to a particular PLDM terminus
uint16	**sensorID** ID of the sensor relative to the given PLDM Terminus ID
uint16	**entityType** The Type value for the entity that is associated with this sensor. See 9.1 for more information.
uint16	**entityInstanceNumber** The Instance Number for the entity that is associated with this sensor. See 9.1 for more information.
uint16	**containerID** The containerID for the containing entity that is associated with this sensor. See 9.1 for more information.

Type	Description
enum8	**sensorInit** Indicates whether the sensor requires initialization by the initializationAgent. value: { noInit, // The Initialization Agent does not take any steps to initialize, // enable, or disable this particular sensor. useInitPDR, // The sensor has an associated State Sensor Initialization PDR // that should be used to initalize the sensor. enableSensor, // When the Initialization Agent runs, it enables this sensor using // a SetStateSensorEnables command to set the // operationalState. disableSensor. // When the Initialization Agent runs, it disables this sensor using // the SetStateSensorEnables command. }
bool8	**sensorDescriptionPDR** true = sensor has a sensorDescription PDR false = sensor does not have an associated sensorDescription PDR
uint8	**compositeSensorCount** The number of state sensors in the terminus that are accessed under the sensorID given in this PDR value: 0x01 to 0x08
var	**possibleStates** One instance of State Sensor Possible States Fields (see Table 69) for each sensor in the PLDM State Sensor, up to sensorCount.

2456 **Table 69 – State Sensor Possible States Fields Format**

Type	Description
uint16	**stateSetID** A numeric value that identifies the PLDM State Set that is used with this sensor
uint8	**possibleStatesSize** The number of bytes (M) in the following possibleStates bitfield value: 0x01 to 0x20 special value : 0x00 can be used to indicate a sensor that is unavailable or disabled from use and should be ignored when accessing the parent compositeSensor through PLDM.

Type	Description
bitfield8 x M	**possibleStates [subset of the State Set that is supported]** A variable length bitfield consisting of one or more bytes, based on the size of the stateSet. If stateSetSize is non-zero, possibleStates consists of one or more 8-bit fields where X = 0 for the first field, X = 1 for the second field (if any), and so on, up to M fields as required by the size of the largest value in the state set. For example, if the largest value in the State Set is 7 or less, this is a one byte bitfield. If the largest value in the State Set is 15 or less, this is a two-byte bitfield, and so on. The value 0b is also used when there is no state set value that corresponds to the corresponding bit position. For example, if a state set has a maximum value of 5, bits [6] and [7] are unused and shall be set to 0b. [7] – 1b = The state that corresponds to value X*8+7 in the state set is supported. 0b = The state that corresponds to value X*8+7 in the state set is not supported. … [2] – 1b = The state that corresponds to value X*8+2 in the state set is supported. 0b = The state that corresponds to value X*8+2 in the state set is not supported. [1] – 1b = The state that corresponds to value X*8+1 in the state set is supported. 0b = The state that corresponds to value X*8+1 in the state set is not supported. [0] – 1b = The state that corresponds to value X*8+0 in the state set is supported. 0b = The state that corresponds to value X*8+0 in the state set is not supported.

28.7 State Sensor Initialization PDR

The State Sensor Initialization PDR contains values that direct the Initialization Agent's initialization of a particular PLDM Single or Composite State Sensor. This action includes enabling or disabling PLDM Event Message generation for individual sensors within the PLDM Composite State Sensor and directing whether a particular sensor will assess an event if the initialization state value does not match the present state of the sensor.

The PDR always has eight state values (stateValue0 through stateValue7). Dummy values must be used (0x00 is recommended) if the implementation does not have a sensor that corresponds to a particular offset. Table 70 describes the format of the PDR.

Table 70 – State Sensor Initialization PDR Format

Type	Description
–	**commonHeader** See 28.1.
uint16	**PLDMTerminusHandle** A handle that identifies PDRs that belong to a particular PLDM terminus
uint16	**sensorID** ID of the sensor relative to the given PLDM terminus

Type	Description
bitfield8	**initConditions** Identifies under which conditions the Initialization Agent must initialize or reinitialize these sensors The initConditions are shared across all sensors that are identified as requiring initialization through the sensorInitMask field. If some sensors require different initialization conditions, a separate PLDM Composite State Sensor Initialization PDR must be used for those sensors. [7:5] – reserved [4] – 1b = PLDM terminus returns to online condition [3] – 1b = System warm resets [2] – 1b = System hard resets [1] – 1b = PLDM subsystem power up [0] – 1b = Initialization Agent controller restart/update (initialize/reinitialize this sensor whenever the device that holds the Initialization Agent has been restarted or reinitialized)
enum8	**sensorEnable** The operational state of the overall composite state sensor after it has been initialized. This state is written to the sensorOperationalState of each sensor that is identified for initialization through the sensorInitMask field of this PDR using the SetStateSensorEnables command. special value: {0xFF = do not set the sensorOperationalStates}
bitfield8	**sensorInitMask** Identifies which sensors within the composite state sensor require initialization [7] – 1b = state sensor at offset 7 requires initialization 0b = state sensor at offset 7 does not require initialization [6] – 1b = state sensor at offset 6 requires initialization 0b = state sensor at offset 6 does not require initialization … [2] – 1b = state sensor at offset 2 requires initialization 0b = state sensor at offset 2 does not require initialization [1] – 1b = state sensor at offset 1 requires initialization 0b = state sensor at offset 1 does not require initialization [0] – 1b = state sensor at offset 0 requires initialization 0b = state sensor at offset 0 does not require initialization

Type	Description
bitfield8	**sensorOpStateEventEnableMask** Identifies which sensors within the composite state sensor should have their operational state event message generation enabled after initialization [7] – 1b = enable event message generator for state sensor at offset 7 0b = disable event message generator for state sensor at offset 7 [6] – 1b = enable event message generator for state sensor at offset 6 0b = disable event message generator for state sensor at offset 6 … [2] – 1b = enable event message generator for state sensor at offset 2 0b = disable event message generator for state sensor at offset 2 [1] – 1b = enable event message generator for state sensor at offset 1 0b = disable event message generator for state sensor at offset 1 [0] – 1b = enable event message generator for state sensor at offset 0 0b = disable event message generator for state sensor at offset 0
bitfield8	**sensorStateEventEnableMask** Identifies which sensors within the composite state sensor should have their state event message generation enabled after initialization [7] – 1b = enable event message generator for state sensor at offset 7 0b = disable event message generator for state sensor at offset 7 [6] – 1b = enable event message generator for state sensor at offset 6 0b = disable event message generator for state sensor at offset 6 … [2] – 1b = enable event message generator for state sensor at offset 2 0b = disable event message generator for state sensor at offset 2 [1] – 1b = enable event message generator for state sensor at offset 1 0b = disable event message generator for state sensor at offset 1 [0] – 1b = enable event message generator for state sensor at offset 0 0b = disable event message generator for state sensor at offset 0
bitfield8	**sensorEventRearm** Directs the sensor to assess an event if the initialization stateValue does not match the present state, or to accept the initialization stateValue as its initial state and ignore any prior state sensorEventRearm value: 1b = trigger an event if the initialization stateValue does not match the present state 0b = accept the initialization stateValue as the present state [7] – sensorEventRearm value for the state sensor at offset 7 [6] – sensorEventRearm value for the state sensor at offset 6 … [2] – sensorEventRearm value for the state sensor at offset 2 [1] – sensorEventRearm value for the state sensor at offset 1 [0] – sensorEventRearm value for the state sensor at offset 0

Type	Description
uint8	**stateValue0** State value to write to sensor offset 0 for initialization special value: Use 0x00 as a placeholder value for sensors that do not require initialization.
uint8	**stateValue1** State value to write to sensor offset 1 for initialization special value: Use 0x00 as a placeholder value for sensors that do not require initialization.
uint8	**stateValue2** State value to write to sensor offset 2 for initialization special value: Use 0x00 as a placeholder value for sensors that do not require initialization.
	…
uint8	**stateValue6** State value to write to sensor offset 14 for initialization special value: Use 0x00 as a placeholder value for sensors that do not require initialization.
uint8	**stateValue7** State value to write to sensor offset 15 for initialization special value: Use 0x00 as a placeholder value for sensors that do not require initialization.

28.8 Sensor Auxiliary Names PDR

The Sensor Auxiliary Names PDR may be used to provide optional information that names the sensor. This record may be used for a single numeric or state sensor, or multiple sensors if the sensor is a composite state sensor.

The nameLanguageTag field can be used to identify the language (such as French, Italian, or English) that is associated with the particular sensorName. Table 71 describes the format of this PDR.

Table 71 – Sensor Auxiliary Names PDR Format

Type	Description
–	**commonHeader** See 28.1.
uint16	**PLDMTerminusHandle** A handle that identifies PDRs that belong to a particular PLDM terminus
uint16	**sensorID** ID of the sensor relative to the given PLDM terminus

Type	Description
uint8	**sensorCount [1..M]** For each sensor x in sensorCount, there can be 1..nameStringCount[x] strings, where each set of strings corresponds to a sensor in a composite sensor. The record must be populated sequentially starting from 1 regardless of whether a sensor requires auxiliary names. Thus, each entry has at least one byte (the nameStringCount). Sensors that have offsets that are greater than sensorCount are treated as if they have no auxiliary names. For example, if a composite sensor contains four sensors and only the third sensor requires an auxiliary name, the sensorCount can be 3 and the nameStringCount for the first two sets of sensor name information is 0.
uint8	**nameStringCount** Number of following pairs [0..N] of nameLanguageTag + sensorName fields for sensor[1].
strASCII	**nameLanguageTag [1]** This field is absent if nameStringCount = 0. A null-terminated ISO646 ASCII string that holds a language tag, per RFC4646, that identifies the primary language in which the sensorName was defined (for example, "en" for English, "zh-cmn-Hans" for simplified Mandarin Chinese, and so on). This field may be used to help select which string to use when multiple character encodings for the sensorName are provided. special value: null string = 0x0000 = unspecified
strUTF-16BE	**sensorName [1]** This field is absent if nameStringCount = 0. A null-terminated unicode string for the auxiliary name of the sensor special value: null string = 0x0000 = name not provided
…	**…**
strASCII	**nameLanguageTag [N]**
strUTF-16BE	**sensorName [N]**

2474
28.9 OEM Unit PDR

2475 The OEM Unit PDR is used to define one or more strings that are used as the name for an OEM Unit
2476 used for PLDM sensors or effecters. The OEM Unit is defined relative to the given Vendor ID and for a
2477 given terminus. The OEMUnitHandle value is required to be unique among all OEM Unit PDRs within a
2478 PDR Repository. The OEMUnitHandle value is not required to be unique across PDR Repositories.

2479 The record also includes a vendor-defined OEMUnitID value that identifies different types of OEM Units
2480 from the given vendor.

2481 The record allows the unit name to be specified using multiple character sets. The unitLanguageTag can
2482 be used to identify the language that is associated with the particular unitName (for example, whether the
2483 unitName is in French, Italian, English, and so on). Table 72 describes the format of this PDR.

2484 **Table 72 – OEM Unit PDR Format**

Type	Description
–	**commonHeader** See 28.1.

Type	Description
uint16	**PLDMTerminusHandle** The terminus that originated this PDR
uint8	**OEMUnitHandle** An opaque number that is used to identify different OEM Units PDRs
uint32	**vendorIANA** The IANA Enterprise Number for the vendor that is defining the OEM Sensor Unit
uint8	**OEMUnitID** A search field for the FindPDR command. This number is assigned by the vendor and provides a numeric ID for the vendor-defined Unit. This value can be used by the vendor to provide a constant ID that always identifies a particular Unit definition from that vendor.
uint8	**stringCount** The number 1..N of unitLanguageTag and unitName field pairs that follow this field
strASCII	**unitLanguageTag[1]** A null terminated ISO646 ASCII string that holds a language tag, per RFC4646, that identifies the primary language in which the unitName was defined (for example, "en" for English, "zh-cmn-Hans" for simplified Mandarin Chinese, and so on). This field may be used to help select which string to use when multiple character encodings for the unitName are provided. special value: null string = unspecified
strUTF-16BE	**unitName[1]** A null terminated unicode string that contains the name of the OEM Sensor Unit
…	**…**
strASCII	**unitLanguageTag[N]**
strUTF-16BE	**unitName[N]**

28.10 OEM State Set PDR

The OEM State Set PDR is used to identify the vendor and OEM State Set ID value when the stateSetID is treated as an OEMStateSetIDHandle. The PDR can also optionally be used to provide names for the different OEM-defined states. Each different state can be assigned a name in one or more languages. A contiguous range of state values can also be assigned a single set of names. It is also possible for the PDR to provide a “hint” to help an entity such as a MAP decide how to treat state values that are not explicitly specified in the PDR. The OEM State Set PDR is applicable to OEM State Sets for both sensors and effecters.

Depending on what range the stateSetID value falls in, the stateSetID value in a PDR, such as the PLDM State Sensor PDR, either identifies the state set number for a particular state set defined in DSP0249 or is a value that is interpreted as an OEMStateSetIDHandle. The OEMStateSetIDHandle value is used to form an association with a particular PLDMOEMStateSetPDR within the PDR Repository. OEMStateSetIDHandle values are thus required to be unique for each different PLDM OEM State Set PDR within a given PDR Repository.

The following example describes the steps that could be taken to interpret the state value information from an event message that originated from a PLDM State Sensor. This includes showing the difference between using one of the standard state set numbers and an OEM State Set number.

1) A PLDM Event Message is received from a state sensor.

2) The TID, sensorID, sensorOffset, and state values (that is presentState and previousState) are read from the message.

3) The TID is used to look up the Terminus Locator Record and obtain the PLDMTerminusHandle value that is associated with the TID.

4) PLDMTerminusHandle and sensorID values are used to look up the PLDM State Sensor PDR for the sensor.

5) The Sensor Offset is used to get the stateSetID from the PLDM State Sensor PDR. If the stateSetID is in the range of standard IDs, the meaning of the state value is given according to the stateSetID defined by the state set identified in DSP0249.

6) Otherwise the stateSetID from the PLDM State Sensor PDR is used as an OEMStateSetIDHandle to look up the OEM State Set PDR that defines the OEM State Set. The PDR identifies the OEM that defined the state set and provides the OEM-specified State Set number (OEMStateSetID) for the state set. The state value from the event message can be used to locate the OEM State Value Record in the PLDM OEM State Set PDR that provides a name string for the particular OEM-defined state.

Table 73 describes the format of the PDR.

Table 73 – OEM State Set PDR Format

Type	Description
–	**commonHeader** See 28.1.
uint16	**PLDMTerminusHandle** The terminus that originated this PDR
uint16	**OEMStateSetIDHandle** An OEM State Set within this PDR Repository. The value is taken from the range of OEMStateSet numbers defined in DSP0249. This value is used in place of standard State Set ID numbers in the PDR for the sensor. When a value in the OEM State Set range is used as the State Set ID in a PDR, it indicates that the corresponding PLDM OEM State Set PDR should be referenced in order to get the OEM identification and definition for the OEM State Set.
uint32	**vendorIANA** The IANA Enterprise Number for the vendor that is defining the OEM State Set given in this PDR
uint16	**OEMStateSetID** A number, assigned by the vendor, that provides a numeric ID for the vendor-defined state set. The vendor can use this value to provide a constant ID that always identifies a particular state set from that vendor. The value shall be in the range defined for OEM State Set numbers defined in DSP0249.
enum8	**unspecifiedValueHint** This field can be used to provide a hint to a higher level entity, such as a MAP, regarding how OEM state values should be treated if they are not explicitly covered by the OEMStateValueRecords field. value: { treatAsUnspecified, treatAsError }

Type	Description
uint8	**stateCount** The number of OEM State Value Records following this field in the PDR. Records shall be stored starting from the lowest stateValue to the highest.
variable	**OEMStateValueRecord** Zero or more OEM State Value Records as specified by the stateCount field. See Table 74.

Table 74 – OEM State Value Record Format

Type	Description
uint8	**minStateValue** The lowest state enumeration value that corresponds to the definition given in this OEM State Value Record instance.
uint8	**maxStateValue** The highest state enumeration value that corresponds to the definition given in this OEM State Value Record instance. State value ranges are not allowed to overlap. If maxStateValue = minStateValue, the following strings apply only to a single state. If maxStateValue > minStateValue**,** the following strings apply to state values in the range from minStateValue through maxStateValue.
uint8	**stringCount** The number 1..N of stateLanguageTag and stateName field pairs that follow this field.
strASCII	**stateLanguageTag[1]** A null terminated ISO646 ASCII string that holds a language tag, per RFC4646, that identifies the primary language in which the stateName was defined (for example, "en" for English, "zh-cmn-Hans" for simplified Mandarin Chinese, and so on). This field may be used to help select which string to use when multiple character encodings for the stateName are provided. special value: null string = unspecified
strUTF-16BE	**stateName[1]** A null terminated unicode string that contains the name for the state
…	…
strASCII	**stateLanguageTag[N]**
strUTF-16BE	**stateName[N]**

28.11 Numeric Effecter PDR

The Numeric Effecter PDR is used to describe the semantics of a PLDM Numeric Effecter to a party such as a MAP. It also includes the factors that are used for converting raw sensor readings to normalized units. The PDR also identifies the entity on which the effecter is operating. Table 75 describes the format of the PDR.

Table 75 – Numeric Effecter PDR Format

Type	Description
–	**commonHeader** See 28.1.
uint16	**PLDMTerminusHandle** A handle that identifies PDRs that belong to a particular PLDM terminus
uint8	**effecterID** ID of the effecter relative to the given PLDM Terminus ID.
uint16	**entityType** The Type value for the entity that is associated with this effecter. See 9.1 for more information.
uint16	**entityInstanceNumber** The Instance Number for the entity that is associated with this effecter. See 9.1 for more information.
uint16	**containerID** The containerID for the containing entity that is associated with this effecter. See 9.1 for more information.
uint16	**effecterSemanticID** This field either identifies a PLDM-defined effecter semantic or provides an OEMEffecterSemanticHandle value, depending on what range the value falls in. If the effecterSemanticID field is set to a value in the OEM range, this value does not directly identify a particular vendor-defined semantic but instead is interpreted as an OEMEffecterSemanticHandle that can be used to locate an OEM Effecter Semantic PDR that identifies the vendor and provides optional name information for the semantic. See DSP0249 for the definition of Effecter Semantic ID values and ranges, and 21.3 for more information. special value: {0x0000 = unspecified }
enum8	**effecterInit** value: { noInit, // The Initialization Agent does not take any steps to initialize, // enable, or disable this particular sensor. useInitPDR, // The sensor has an associated Numeric Effecter Initialization // PDR that should be used to initalize the sensor. enableEffecter, // When the Initialization Agent runs, it enables this effecter using // a SetNumericEffecterEnable command to set the // operationalState. disableEffecter // When the Initialization Agent runs, it disables this effecter using // the SetNumericEffecterEnable command. }
bool8	**effecterDescriptionPDR** true = effecter has an effecterDescription PDR false = effecter does not have an associated effecterDescription PDR
enum8	**baseUnit** The base unit of the reading returned by this effecter. See 27.1 for more information. value: { see Table 62 }

Type	Description
sint8	**unitModifier** A power-of-10 multiplier for the baseUnit. See 27.1 for more information.
enum8	**rateUnit** value: { None, Per MicroSecond, Per MilliSecond, Per Second, Per Minute, Per Hour, Per Day, Per Week, Per Month, Per Year }
uint8	**baseOEMUnitHandle** This value is used to locate the PLDM OEM Unit PDR that defines the OEMUnit if the OEMUnit value is used for the baseUnit.
enum8	**auxUnit** The base unit of the reading returned by this effecter. See 27.2 for more information. value: { see Table 62 }
sint8	**auxUnitModifier** A power-of-10 multiplier for the auxUnit. See 27.2 for more information.
enum8	**auxrateUnit** value: { None, Per MicroSecond, Per MilliSecond, Per Second, Per Minute, Per Hour, Per Day, Per Week, Per Month, Per Year }
uint8	**auxOEMUnitHandle** This value is used to locate the PLDM OEM Unit PDR that defines the OEMUnit if the OEMUnit value is used for the auxUnit.
bool8	**isLinear** This value is used to provide information that can be used by a MAP to populate the IsLinear attribute of CIM_NumericSensor. Currently, the CIM_NumericSensor description of this field is "Indicates that the Sensor is linear over its dynamic range." value: This field is typically set to "true".
enum8	**effecterDataSize** The bit width and format of reading and threshold values that the effecter returns value: { uint8, sint8, uint16, sint16, uint32, sint32 }
real32	**resolution** The resolution of the effecter in Units (see 27.7)
real32	**offset** A constant value that is added as part of the conversion process of converting a raw effecter reading to Units (see 27.7).
uint16	**accuracy** Given as a +/- percentage in 1/100ths of a % from 0.00 to 100.00. For example, the integer value 510 corresponds to ± 5.10%. See 27.6 for more information.
uint8	**plusTolerance** Tolerance is given in +/- counts of the setting value. It indicates a constant magnitude possible error in the generation of an analog output from an effecter. It is possible that the tolerance could be asymmetric. The plusTolerance field provides the "+" value of the tolerance; the minusTolerance field provides the minus portion. For example, if plusTolerance is 0x02 and minusTolerance is 0x00, the tolerance is +2/-0 counts. See 27.6 for more information about how tolerance is defined and used.

Type	Description
uint8	**minusTolerance** Tolerance is given in +/- counts of the setting value. It indicates a constant magnitude possible error in the generation of an analog input from an effecter. It is possible that the tolerance could be asymmetric. The plusTolerance field provides the “+” value of the tolerance; the minusTolerance field provides the minus portion. For example, if plusTolerance is 0x02 and minusTolerance is 0x00, the tolerance is +2/-0 counts. See 27.6 for more information about how tolerance is defined and used.
real32	**stateTransitionInterval** The length of time the effecter takes to do an enabledState change (worst case), in seconds NOTE: Because this is floating point format, fractional seconds can be represented. The real32 format also supports a value for ”unknown”.
real32	**TransitionInterval** The length of time the effecter takes to have a setting change take effect (worst case), in seconds.
uint8 | sint8 | uint16 | sint16 | uint32 | sint32	**maxSettable** The maximum legal setting value that the effecter accepts. The size of this field is given by the effecterDataSize field in this PDR. This number is given in the same format as the reading returned by the effecter. The conversion formula is used to convert this number to normalized units. See definition in 27.1.
uint8 | sint8 | uint16 | sint16 | uint32 | sint32	**minSettable** The minimum legal setting value that the effecter accepts. The size of this field is given by the effecterDataSize field in this PDR. This number is given in the same format as the reading returned by the effecter. The conversion formula is used to convert this number to normalized units. See definition in 27.1.
enum8	**rangeFieldFormat** Indicates the format used for the following nominalValue, normalMax, and normalMin fields. value: { uint8, sint8, sint16, uint32, sint32, real32 }
bitfield8	**rangeFieldSupport** This field indicates which of the fields that identify the operating ranges of the parameter set by the effecter are supported. (This bitfield indicates whether the following nominalValue, normalMax, and so on, fields contain valid range values.) [7:5] – reserved [4] – 1b = ratedMin field supported [3] – 1b = ratedMax field supported [2] – 1b = normalMin field supported [1] – 1b = normalMax field supported [0] – 1b = nominalValue field supported

Type	Description
uint8 | sint8 | uint16 | sint16 | uint32 | sint32 | real32	**nominalValue** This value presents the nominal value for the parameter that is accepted by the effecter. The size of this field is given by the rangeFieldFormat field in this PDR. This value is given directly in the specified units without the use of any conversion formula. For example, if the units are millivolts and the nominalValue is 5000, the nominalValue corresponds to 5000 mV, or 5.000 V. It is possible that the nominal value could be some fraction of the given units for the effecter (for example, if the units are volts and the nominal value is 2.5 V). For this reason, the nominalValue can be expressed using a real32. The value is defined as the nominal value for what is being set. The nominalValue is not required to match a value that can be returned as a reading by the effecter implementation. For example, if the nominal value for a voltage setting effecter was 5.00 V, the nominalValue would typically be reported as 5.00 V even though the closest setting the effecter implementation may be able to accept is 5.05 V. A common use of the nominalValue is as a source of part of the identifying “name” for an effecter. For example, it is common for voltage effecters to be identified by their nominal reading. So, an effecter with a nominal reading of +5.00 V would be referred to as a "+5 V effecter", while one with a nominal reading of +3.3 V would be referred to as a "+3.3 V effecter". The definition of nominalValue in the PDR supports this usage. An application that uses or displays this value will typically elect to round the value to some number of significant digits using an algorithm based on the resolution of the effecter. For example, if the nominalValue is given as a real32 as 2.50000 V, but the resolution of the effecter is 0.05 V, the nominalValue displayed would typically be rounded as 2.50 V. It is possible that a given effecter may not be considered as having a nominal setting, in which case this field should be ignored. For example, a numeric effecter that sets a count or size of some parameter may not be considered as having a nominal setting depending on its application.
uint8 | sint8 | uint16 | sint16 | uint32 | sint32 | real32	**normalMax** The upper limit of the normal operating range for the parameter that is set by the numeric effecter. The setting is considered to be operating outside of normal range when this value is exceeded. For example, if a monitored voltage of a component is specified in its data sheet to have a normal maximum operating range of 4.75 to 5.25 V, this value would be set to 5.25 (assuming the units in the PDR are for volts). This value is given directly in the specified units without the use of any conversion formula. This value is used together with normalMin to indicate the normal operating range for the effecter.
uint8 | sint8 | uint16 | sint16 | uint32 | sint32 | real32	**normalMin** The lower limit of the normal operating range for the parameter that is set by the numeric effecter. Effecter thresholds are typically set for a value that is lower than normalMin to accommodate the affects of effecter accuracy, tolerance, and resolution, in order to prevent false reporting of an “out-of-range” event state. This value is given directly in the specified units without the use of any conversion formula.
uint8 | sint8 | uint16 | sint16 | uint32 | sint32 | real32	**ratedMax** The upper limit of the rated operating range for the parameter that is set by the numeric effecter. The monitored parameter is considered to be operating outside of rated operating range when this value is exceeded.
uint8 | sint8 | uint16 | sint16 | uint32 | sint32 | real32	**ratedMin** The lower limit of the rated operating range for the parameter that is set by the numeric effecter. The monitored parameter is considered to be operating outside of rated operating range below this value.

28.12 Numeric Effecter Initialization PDR

The Numeric Effecter Initialization PDR reports the values that are used when a PLDM Effecter Sensor is initialized by a PLDM Initialization Agent. Table 76 describes the format of this PDR.

Table 76 – Numeric Effecter Initialization PDR Format

Type	Description
–	**commonHeader** See 28.1.
uint16	**PLDMTerminusHandle** A handle that identifies PDRs that belong to a particular PLDM terminus
uint16	**effecterID** ID of the effecter relative to the given PLDM Terminus ID
enum8	**effecterEnable** The operational state of the effecter after it has been initialized. This state is written to the effecter using the SetEffecterEnable command. special value: {0xFF = do not issue a SetEffecterEnable command to set the Effecter Operational State }
bitfield8	**initConditions** Identifies under which conditions the Initialization Agent must initialize or reinitialize this effecter [7:5] – reserved [4] – 1b = PLDM terminus returns to online condition [3] – 1b = System warm resets [2] – 1b = System hard resets [1] – 1b = PLDM subsystem power up [0] – 1b = Initialization Agent controller restart/update (initialize/reinitialize this effecter whenever the device that holds the Initialization Agent has been restarted or reinitialized)
enum8	**effecterDataSize** The bit width of reading and threshold values that the effecter returns value: { uint8, sint8, uint16, sint16, uint32, sint32 }
uint8 | sint8 | uint16 | sint16 | uint32 | sint32	**effecterData** The numeric value written to the effecter. The size of this field is determined by the value of the effecterDataSize field.

28.13 State Effecter PDR

The State Effecter PDR is used to provide information about a PLDM Composite State Effecter. Table 77 describes the format of this PDR.

Table 77 – State Effecter PDR Format

Type	Description
–	**commonHeader** See 28.1.
uint16	**PLDMTerminusHandle** A handle that identifies PDRs that belong to a particular PLDM terminus
uint16	**effecterID** ID of the effecter relative to the given PLDM Terminus ID
uint16	**entityType** The Type value for the entity that is associated with this sensor. See 9.1. for more information.
uint16	**entityInstanceNumber** The Instance Number for the entity that is associated with this sensor. See 9.1. for more information.
uint16	**containerID** The containerID for the containing entity that is associated with this sensor. See 9.1. for more information.
uint16	**effecterSemanticID** This field either identifies a PLDM-defined effecter semantic or provides an OEMEffecterSemanticHandle value, depending on what range the value falls in. If the effecterSemanticID field is set to a value in the OEM range, this value does not directly identify a particular vendor-defined semantic but instead is interpreted as an OEMEffecterSemanticHandle that can be used to locate an OEM Effecter Semantic PDR that identifies the vendor and provides optional name information for the semantic. See DSP0249 for the definition of Effecter Semantic ID values and ranges, and 21.3 for more information. special value: {0x0000 = unspecified }
bool8	**effecterInit** value: { noInit, // The Initialization Agent does not take any steps to initialize, // enable, or disable this particular effecter. useInitPDR, // The effecter has an associated State Effecter Initialization PDR // that should be used to initalize the effecter. enableEffecter, // When the Initialization Agent runs, it enables this effecter using // a SetStateEffecterEnables command to set the // operationalState. disableEffecter. // When the Initialization Agent runs, it disables this effecter using // the SetStateEffecterEnables command. }
bool8	**effecterDescriptionPDR** true = effecter has an effecterDescription PDR false = effecter does not have an associated effecterDescription PDR

Type	Description
uint8	**compositeEffecterCount** The number of state effecters in the terminus that are accessed under the effecterID given in this PDR. value: 0x01 to 0x08
var	**possibleStates** One instance of State Effecter Possible States Fields (see Table 78) for each effecter in the PLDM State Effecter, up to effecterCount.

2535 **Table 78 – State Effecter Possible States Fields Format**

Type	Description
uint16	**stateSetID** A numeric value that identifies the PLDM State Set that is used with this effecter.
uint8	**possibleStatesSize** The number of bytes (M) in the possibleStates bitfield. value: 0x01 to 0x20 special value : 0x00 can be used to indicate a effecter that is unavailable or disabled from use and should be ignored when accessing the parent composite effecter with PLDM.
bitfield8 x M	**possibleStates [subset of the State Set that is supported]** A variable length bitfield that consists of one or more bytes, based on the size of the state set. If stateSetSize is non-zero, possibleStates consists of one or more 8-bit fields where X=0 for the first field, X=1 for the second field (if any), and so on, up to M fields as required by the size of the largest value in the state set. For example, if the largest value in the state set is 7 or less, this will be a one-byte bitfield. If the largest value in the state set is 15 or less, this will be a two-byte bitfield, and so on. The value 0b is also used when no state set value corresponds to the corresponding bit position. For example, if a state set has a maximum value of 5, bits [6] and [7] are unused and shall be set to 0b. [7] – 1b = state that corresponds to value X*8+7 in the state set is supported 0b = state that corresponds to value X*8+7 in the state set is not supported … [2] – 1b = state that corresponds to value X*8+2 in the state set is supported 0b = state that corresponds to value X*8+2 in the state set is not supported [1] – 1b = state that corresponds to value X*8+1 in the state set is supported. 0b = state that corresponds to value X*8+1 in the state set is not supported [0] – 1b = state that corresponds to value X*8+0 in the state set is supported 0b = state that corresponds to value X*8+0 in the state set is not supported

2536 28.14 State Effecter Initialization PDR

2537 The State Effecter Initialization PDR describes settings that the Initialization Agent uses to initialize a
2538 PLDM Single or Composite State Effecter.

2539 The PDR always has eight state values. Dummy values must be used (0x00 is recommended) if the
2540 implementation does not have an effecter that corresponds to a particular offset. Table 79 describes the
2541 format of the PDR.

2542 **Table 79 – State Effecter Initialization PDR Format**

Type	Description
–	**commonHeader** See 28.1.
uint16	**PLDMTerminusHandle** A handle that identifies PDRs that belong to a particular PLDM terminus
uint16	**effecterID** ID of the effecter relative to the given PLDM terminus
uint16	**entityType** The Type value for the entity that is associated with this sensor. See 9.1 for more information.
uint16	**entityInstanceNumber** The Instance Number for the entity that is associated with this sensor. See 9.1 for more information.
uint16	**containerID** The containerID for the containing entity that is associated with this sensor. See 9.1 for more information.
bitfield8	**initConditions** Identifies the conditions under which the Initialization Agent must initialize or reinitialize this effecter [7:5] – reserved [4] – 1b = PLDM terminus returns to online condition [3] – 1b = System warm resets [2] – 1b = System hard resets [1] – 1b = PLDM subsystem power up [0] – 1b = Initialization Agent controller restart/update (initialize/reinitialize this effecter whenever the device that holds the Initialization Agent has been restarted or reinitialized)
enum8	**effecterEnable** The operational state of the overall composite state sensor after it has been initialized. This state is written to the sensorOperationalState of each sensor that is identified for initialization through the effecterInitMask field of this PDR using the SetStateEffecterEnables command. special value: {0xFF = do not set the effecterOperationalStates}

Type	Description
bitfield8	**effecterInitMask** Identifies which effecters within the composite state effecter require initialization [7] – 1b = state effecter at offset 7 requires initialization 0b = state effecter at offset 7 does not require initialization [6] – 1b = state effecter at offset 6 requires initialization 0b = state effecter at offset 6 does not require initialization … [2] – 1b = state effecter at offset 2 requires initialization 0b = state effecter at offset 2 does not require initialization [1] – 1b = state effecter at offset 1 requires initialization 0b = state effecter at offset 1 does not require initialization [0] – 1b = state effecter at offset 0 requires initialization 0b = state effecter at offset 0 does not require initialization
bitfield8	**effecterOpStateEventEnableMask** Identifies which sensors within the composite state effecter should have their operational state event message generation enabled after initialization [7] – 1b = enable event message generator for state sensor at offset 7 0b = disable event message generator for state sensor at offset 7 [6] – 1b = enable event message generator for state sensor at offset 6 0b = disable event message generator for state sensor at offset 6 … [2] – 1b = enable event message generator for state sensor at offset 2 0b = disable event message generator for state sensor at offset 2 [1] – 1b = enable event message generator for state sensor at offset 1 0b = disable event message generator for state sensor at offset 1 [0] – 1b = enable event message generator for state sensor at offset 0 0b = disable event message generator for state sensor at offset 0
uint8	**stateValue0** State value to write to effecter offset 0 for initialization special value: Use 0x00 as a placeholder value for effecters that do not require initialization.
uint8	**stateValue1** State value to write to effecter offset 1 for initialization special value: Use 0x00 as a placeholder value for effecters that do not require initialization.
uint8	**stateValue2** State value to write to effecter offset 2 for initialization special value: Use 0x00 as a placeholder value for effecters that do not require initialization.
	…
uint8	**stateValue6** State value to write to effecter offset 6 for initialization special value: Use 0x00 as a placeholder value for effecters that do not require initialization.
uint8	**stateValue7** State value to write to effecter offset 7 for initialization special value: Use 0x00 as a placeholder value for effecters that do not require initialization.

28.15 Effecter Auxiliary Names PDR

The Effecter Auxiliary Names PDR may be used to provide optional information that names an effecter. This record may be used for a single effecter or multiple effecters if the effecter is a composite state effecter.

The nameLanguageTag field can be used to identify the language (such as French, Italian, or English) that is associated with the particular effecter name. Table 80 describes the format of this PDR.

Table 80 – Effecter Auxiliary Names PDR Format

Type	Description
–	**commonHeader** See 28.1.
uint16	**PLDMTerminusHandle** A handle that identifies PDRs that belong to a particular PLDM terminus
uint16	**effecterID** ID of the effecter relative to the given PLDM terminus
uint8	**effecterCount [1..M]** For each effecter x in effecterCount, there can be 1..nameStringCount[x] strings, where each set of strings corresponds to a effecter in a composite effecter. The record must be populated sequentially starting from 1 regardless of whether an effecter requires auxiliary names. Thus, each entry has at least one byte (the nameStringCount). Effecters that have offsets that are greater than effecterCount are treated as if they have no auxiliary names. For example, if a composite effecter contains four effecters and only the third effecter requires an auxiliary name, the effecterCount can be 3 and the nameStringCount for the first two sets of effecter name information is 0.
effecter [1] names:	
uint8	**nameStringCount** Number of following pairs [0..N] of nameLanguageTag + effecterName fields for effecter[1].
strASCII	**nameLanguageTag[1]** This field is absent if nameStringCount = 0. A null terminated ISO646 ASCII string that holds a language tag, per RFC4646, that identifies the primary language in which the effecterName was defined (for example, "en" for English, "zh-cmn-Hans" for simplified Mandarin Chinese, and so on). This field may be used to help select which string to use when multiple character encodings for effecterName are provided. special value: null string = 0x0000 = unspecified
strUTF-16BE	**effecterName[1]** This field is absent if nameStringCount = 0. A null terminated unicode string for the name of the auxiliary effecter special value: null string = 0x0000 = name not provided.
…	**…**
strASCII	**nameLanguageTag[N]**
strUTF-16BE	**effecterName[N]**
effecter [2] names:	
…	
effecter [M] names:	

28.16 OEM Effecter Semantic PDR

The OEM Effecter Semantic PDR is used to provide information about an OEM effecter semantic used with one or more PLDM effecters that are represented in the PDRs. The information includes an ID for the vendor and a vendor-defined ID number for identifying the effecter semantic. The PDR also allows one or more descriptive name strings to be provided for the vendor-defined effecter semantic. The name strings may be provided in different character sets and languages.

The OEMEffecterSemanticHandle value in the PDR is used by other PDRs, such as the PLDM State Effecter PDR, to point to the particular PLDM OEM Effecter Semantic PDR within the PDR Repository. OEMStateSetIDHandle values are thus required to be unique for each different PLDM OEM State Set PDR within a given PDR Repository.

The OEMSemanticID field enables the vendor that defined the semantic to assign an ID value to its semantic. The OEMSemanticID field is thus defined relative to the given vendor ID.

The OEM Effecter Semantic PDR also contains a PLDMTerminusHandle value. The PLDMTerminusHandle is used to provide a record of the terminus from which the PDR was imported. It is expected that most vendors will define their OEMSemanticID values in a global manner in which the ID has the same meaning regardless of the PLDMTerminusHandle value.

Table 81 describes the format of this PDR.

Table 81 – OEM Effecter Semantic PDR Format

Type	Description
–	**commonHeader** See 28.1.
uint16	**PLDMTerminusHandle** This value is used to identify the terminus that originated this PDR.
uint8	**OEMEffecterSemanticHandle** An opaque number that is used to identify different OEM effecter semantics that are defined by the given vendor on the given terminus. The value is used in PDRs such as the PLDM State Effecter PDR to indicate that a vendor-defined effecter semantic is being used and to locate the PLDM OEM Effecter Semantic PDRs (if any) that provide the vendor-defined ID number and optional descriptive names for the effecter semantic.
uint32	**vendorIANA** The IANA Enterprise Number for the vendor that is defining the OEM Sensor Unit
uint8	**OEMEffecterSemanticID** A value that can be used as a search field for the FindPDR command. This number is assigned by the vendor and provides a numeric ID for the vendor-defined effecter semantic. Thus, the vendor can use this value to provide a constant ID that always identifies a particular Unit definition from that vendor.
uint8	**stringCount** The number 1..N of languageTag and name field pairs that follow this field. { 0 = no name information provided }
strASCII	**languageTag[1]** A null terminated ISO646 ASCII string that holds a language tag, per RFC4646, that identifies the primary language in which the unitName was defined (for example, "en" for English, "zh-cmn-Hans" for simplified Mandarin Chinese, and so on). This field may be used to help select which string to use when multiple character encodings for the unitName are provided. special value: null string = unspecified
strUTF-16BE	**name[1]** A null terminated unicode string that contains the name of the OEM Sensor Unit
…	**…**
strASCII	**languageTag[N]**

Type	Description
strUTF-16BE	**name[N]**

28.17 Entity Association PDR

The Entity Association PDR is used to form associations between entities, such as physical and logical entities. See section 10 for more information. Table 82 describes the format of this PDR.

Table 82 – Entity Association PDR Format

Type	Description
–	**commonHeader** See 28.1.
uint16	**containerID** value: 0x0001 to 0xFFFF = An opaque number that identifies a particular container entity in the hierarchy of containment. See 11.1 for more information. special value: 0x0000 = "SYSTEM". This value is used to identify the topmost containing entity in PLDM Entity Association containment hierarchies.
enum8	**associationType** value: { physicalToPhysicalContainment, logicalContainment }
Container Entity Identification Information	
uint16	**containerEntityType**
uint16	**containerEntityInstanceNumber**
uint16	**containerEntityContainerID**
Contained Entity Identification Information	
uint8	**containedEntityCount** The number of contained entities (1 to N) listed in this particular PDR. This may not be the total number of contained entities because multiple containment association PDRs may exist for the same container entity. See 11.3 for more information.
uint16	**containedEntityType[1]**
uint16	**containedEntityInstanceNumber[1]**
uint16	**containedEntityContainerID[1]**
	…
uint16	**containedEntityType[N]**
uint16	**containedEntityInstanceNumber[N]**
uint16	**containedEntityContainerID[N]**

28.18 Entity Auxiliary Names PDR

The Entity Auxiliary Names PDR may be used to provide optional information that names a particular instance of an entity. The PDR can also be used to name a particular range of instances of an entity, provided that the instances share the same containerID.

The nameLanguageTag field can be used to identify the language (such as French, Italian, or English) that is associated with the particular entity name. Table 83 describes the format of this PDR.

Table 83 – Entity Auxiliary Names PDR Format

Type	Description
–	**commonHeader** See 28.1.
uint16	**entityType**
uint16	**entityInstanceNumber**
uint16	**entityContainerID**
uint8	**sharedNameCount** This number is added to the EntityInstanceNumber to identify how many additional EntityInstanceNumber values share this auxiliary name PDR, where EntityInstanceNumber identifies the starting value for the range. For example, if the EntityInstanceNumber is 100 and the sharedNameCount is 2, this PDR applies to EntityInstanceNumbers 100, 101, and 102. If the sharedNameCount is 0, this PDR applies only to the given EntityInstanceNumber.
Entity auxiliary names:	
uint8	**nameStringCount** Number of following pairs [0..N] of nameLanguageTag + entityAuxName fields for entityAuxName[1].
strASCII	**nameLanguageTag [1]** This field is absent if nameStringCount = 0. A null terminated ISO646 ASCII string that holds a language tag, per RFC4646, that identifies the primary language in which the entityAuxName was defined (for example, "en" for English, "zh-cmn-Hans" for simplified Mandarin Chinese, and so on). This field may be used to help select which string to use when multiple character encodings for the entityAuxName are provided. special value: null string = 0x0000 = unspecified
strUTF-16BE	**entityAuxName [1]** This field is absent if nameStringCount = 0. A null terminated unicode string for the auxiliary name of the entity. special value: null string = 0x0000 = name not provided
…	**…**
strASCII	**nameLanguageTag [N]**
strUTF-16BE	**entityAuxName [N]**

28.19 OEM EntityID PDR

The OEM EntityID PDR can be used to provide a vendor-specific EntityID definition when no PLDM predefined EntityID corresponds to the type of entity that the vendor wants to represent.

When the entityType value is in the OEM range of values, the EntityID portion of the entityType field is OEM-defined. The EntityID value is then used as an OEMEntityIDHandle to locate the corresponding OEM EntityID PDR.

OEM Entity Type PDRs need to be able to be exported by a terminus, such as a terminus on a hot-plug card. The numbers in a given vendor's Device PDRs must be picked a priori by the vendor. Thus, duplications may exist among the OEM EntityID values that different vendors choose. The Discovery Agent function is responsible for adjusting the OEM Entity Type values to resolve any conflicts that may occur when it integrates PDRs into the Primary PDR Repository. Users of OEM EntityID values must be aware that these values may differ between different PDR Repositories. That is, an OEM EntityID for "widget" from vendor "ABC" will not always have the same Entity ID value across PDRs.

To facilitate the identification of particular OEM EntityIDs from a given vendor, each PDR includes a vendor-specific ID value that does not get altered by the Discovery Agent function. When used in conjunction with the vendor's ID, this provides a value that can always be used to identify the particular vendor-defined EntityID definition.

Table 84 describes the format of this PDR.

Table 84 – OEM EntityID PDR Format

Type	Description
–	**commonHeader** See 28.1.
uint16	**PLDMTerminusHandle** This value is used to identify the terminus that originated this PDR.
uint16	**OEMEntityIDHandle** [15] – 0b = reserved [14:0] – OEM entityID handle value. The value that is used in entity associations and other PDRs to identify the entity defined by this PDR. This value may be changed if the PDR is migrated and integrated into a Primary PDR Repository.
uint32	**vendorIANA** The IANA Enterprise Number for the vendor that is defining the OEM PDR vendor-specific data
uint16	**vendorEntityID** This value can be used as a search field for the FindPDR command. This number is assigned by the vendor and provides a numeric ID for the vendor-defined entity. This field is intended to provide a consistent and constant ID that can be relied on to identify the vendor-defined entity even if the name strings need to be changed or updated. [15] – 0b = reserved [14:0] – vendorEntityID value
uint8	**stringCount** The number 1..N of entityIDLanguageTag and entityIDName field pairs that follow this field.

Type	Description
strASCII	**entityIDLanguageTag[1]** A null terminated ISO646 ASCII string that holds a language tag, per RFC4646, that identifies the primary language in which the EntityID name was defined (for example, "en" for English, "zh-cmn-Hans" for simplified Mandarin Chinese, and so on). This field may be used to help select which string to use when multiple character encodings for the entityIDName are provided. special value: null string = unspecified
strUTF-16BE	**entityIDName[1]** A null terminated unicode string that contains the name of the EntityID name
…	**…**
strASCII	**entityIDLanguageTag[N]**
strUTF-16BE	**entityIDName[N]**

28.20 Interrupt Association PDR

The Interrupt Association PDR is used to form associations between interrupt source entities and interrupt target entities. See 11.10 for more information. Table 85 describes the format of this PDR.

Table 85 - Interrupt Association PDR Format

Type	Description
–	**commonHeader** See 28.1.
uint16	**PLDMTerminusHandle** This value is used to identify the terminus that provides access to the sensor that is monitoring the interrupt that is related to this association.
uint16	**sensorID** The ID of the sensor that monitors this interrupt at a source or target
enum8	**sourceOrTargetSensor** Identifies whether the sensor is monitoring the interrupt at the source or the target. The association record for a sensor that monitors an interrupt source is required to identify only a single target entity and a single source entity. value: { targetSensor, sourceSensor }
Target Entity Identification Information	
uint16	**interruptTargetEntityType**
uint16	**interruptTargetEntityInstanceNumber**
uint16	**interruptTargetEntityContainerID**
Source Entity Identification Information	
uint8	**interruptSourceEntityCount** The number of interruptSource entities (1 to N) listed in this particular PDR. This number may not be the total number of interruptSource entities associated with a particular interrupt target entity because multiple interrupt association PDRs may exist for the same target entity. See 11.3 and 11.10 for more information.
uint32	**interruptSourcePLDMTerminusHandle[1]**

Type	Description
uint16	**interruptSourceEntityType[1]**
uint16	**interruptSourceEntityInstanceNumber[1]**
uint16	**interruptSourceEntityContainerID[1]**
uint16	**interruptSourceSensorID[1]**
	…
uint32	**interruptSourcePLDMTerminusHandle[N]**
uint16	**interruptSourceEntityType[N]**
uint16	**interruptSourceEntityInstanceNumber[N]**
uint16	**interruptSourceEntityContainerID[N]**
uint16	**interruptSourceSensorID[N]**

28.21 Event Log PDR

The Event Log PDR is used to describe characteristics of the PLDM Event Log (if implemented). The specification defines the existence of only a single, central PLDM Event Log function. Therefore, only one occurrence of a PLDM Event Log PDR shall exist in a Primary PDR Repository.

Table 86 describes the format of this PDR.

Table 86 – Event Log PDR Format

Type	Description
–	**commonHeader** See 28.1.

uint32	**logSize**
	The size in bytes of the log storage area that is used for storing log entries. This number is exclusive of any fixed overhead for maintaining the overall log, but may include per entry overhead.
	special value:
	{
	0x0000_0000 = unspecified.
	0xFFFF_FFFE = reserved for future definition
	0xFFFF_FFFF = log size is greater than or equal to 4 GB-1 bytes
	}

bitfield8	**supportedLogClearingPolicies** See 13.4 for a description of the log clearing policies. [7:3] – reserved [2] – 1b = clearOnAge supported [1] – 1b = FIFO supported [0] – 1b = fillAndStop supported

Type	Description
uint8	**entryIDTimeout** The minimum interval, in seconds, that the entryID used in the middle of a partial transfer remains valid after it was delivered in the response for a GetPLDMEventLogEntry command that returns partial data. This corresponds to the entryID value returned in any GetPLDMEventLogEntry responses where the splitEntry field in the response is firstFragment or middleFragment. special values: { 0x00 = no timeout, 0x01 = default minimum timeout is the same as the PDR Handle Timeout, **MC1**, (see section 29), 0xFF = timeout >254 seconds. Any timeout values that are less than the specified default minimum timeout are illegal. }
uint8	**perEntryOverhead** The number of bytes of storage overhead per entry if that overhead is counted as using space from the log area specified by logSize. For example, if this value is 2 and an N-byte entry was added to the log, the amount of logSize consumed would be N+2 bytes. An implementation may elect to hide some or all of the impact of per-entry overhead on the available log space. For example, the implementation may have an internal overhead of 2 bytes but keep that overhead in a separate data structure that does not affect the amount of space consumed from the log. In this case, adding an N-byte entry to the log would be counted as consuming only N-bytes of log space, not N+2 bytes. special value: 0xFF = unspecified
uint8	**allocationGranularity** The byte multiple or increment by which storage space is allocated to entries. This value typically corresponds to some byte, word, or block boundary related to the physical medium used for storing entries. For example, if this value is 16 and a 24-byte entry were added, the result would be that the entry would consume 32-bytes of storage space. special value: 0xFF = unspecified
uint8	**percentUsedResolution** Indicates the resolution of the storagePercentUsed value from the GetPLDMEventLogInfo command value: 1 to 100; all other values = reserved A percentUsedResolution value of 0x01 indicates that the storagePercentUsed value is given with a resolution of 1 count (1%), which means a storagePercentUsed value of 0x00 indicates that the log is from 0 to <1% full, a storagePercentUsed value of 0x01 indicates that the log is 1% to <2% full, and so on. A percentUsedResolution value of 0x05 indicates that the storagePercentUsed value is given with a resolution of 5 count (5%), which means a storagePercentUsed value of 0x00 indicates that the log is from 0 to <5% full, a storagePercentUsed value of 0x01 indicates that the log is 5% to <10% full, and so on.

28.22 OEM Device PDR

The OEM Device PDR can be used to provide OEM (vendor-specific) information. The OEM-specific data portion in an OEM Device PDR is limited to a maximum size of 64 KB. Higher-level system specifications may place additional limits on the size and number of OEM Device PDRs that may be supported in a given PLDM subsystem implementation. An OEM Device PDR must have at least one byte of VendorSpecificData.

This type of PDR shall be copied by the Discovery Agent into the Primary PDR Repository dependent on the setting of the copyPDR field. The PDR may also be preconfigured into the Primary PDR Repository. That is, this PDR is not restricted to being only used or migrated from repositories that are separate from the Primary PDR Repository.

The OEM PDR is a slightly smaller version of the OEM Device PDR that can be used in situations where it is not necessary or desired to associate the PDR to a particular terminus or have the information copied from a Device PDR Repository into the Primary PDR Repository.

Table 87 describes the format of this PDR.

28.22.1 Copy Behavior

If the copyPDR parameter is set to copyToPrimaryRepository, the Discovery Agent shall overwrite any pre-existing PDRs for the terminus that have the same vendorIANA and VendorHandle values.

28.22.2 Removal Behavior

The OEM Device PDR is allowed to be removed from the Primary PDR Repository if the Discovery Agent detects that the terminus that is associated with the PDR has been removed or is no longer available.

Table 87 – OEM Device PDR Format

Type	Description
–	**commonHeader** See 28.1.
uint16	**PLDMTerminusHandle** The PLDMTerminusHandle for the terminus from which this record was obtained. special value: 0x0000 may be used to indicate "unspecified' when this record is in a device's PDR Repository. The Discovery Agent typically assigns a different value to this field when merging the record into the Primary PDR Repository.
enum8	**copyPDR** value: { doNotCopy, copyToPrimaryRepository }
uint32	**vendorIANA** The IANA Enterprise Number for the vendor that is defining the OEM PDR vendor -specific data special value: 0 = unspecified
uint16	**OEMRecordID** This value can be used as a search field for the FindPDR command. This value must be unique among all OEM Device PDRs for a given terminus that share the same vendorIANA value. Any other semantics associated with this value are vendor-specific and defined by the vendor or group that is identified by vendorIANA.
uint16	**dataLength** The number of following vendorSpecificData bytes starting from 0. 0 = 1 byte, 1 = 2 bytes, and so on
byte	**vendorSpecificData[0]**
…	**…**
byte	**vendorSpecificData[N]**

28.23 OEM PDR

The OEM PDR can be used to provide OEM (vendor-specific) information. The OEM-specific data portion in an OEM PDR is limited to a maximum size of 64 KB. Higher-level system specifications may place additional limits on the size and number of OEM PDRs that may be supported in a given PLDM subsystem implementation. An OEM PDR must have at least one byte of VendorSpecificData. The OEM Device PDR is an extended version of the OEM PDR that is used when it is necessary to associate the PDR to a particular terminus or to have the information copied from a Device PDR Repository into the Primary PDR Repository.

Table 88 describes the format of this PDR.

Table 88 – OEM PDR Format

Type	Description
–	**commonHeader** See 28.1.
uint32	**vendorIANA** The IANA Enterprise Number for the vendor that is defining the OEM PDR vendor-specific data special value: 0 = unspecified
uint16	**OEMRecordID** This value can be used as a search field for the FindPDR command. This value must be unique among all OEM PDRs within the PDR Repository that share the same vendorIANA value. Any other semantics associated with this value are vendor-specific and defined by the vendor or group that is identified by vendorIANA.
uint16	**dataLength** The number of following vendor-specific data bytes starting from 0 0 = 1 byte, 1 = 2 bytes, and so on.
byte	**vendorSpecificData[1]**
…	…
byte	**vendorSpecificData[N]**

29 Timing

Table 89 defines timing values that are specific to this document.

Table 89 – Monitoring and Control Timing Specifications

Timing Specification	Symbol	Min	Max	Description
PDR record handle retention	MC1	30 sec	–	See 26.2.8.

30 Command Numbers

Table 90 defines the command numbers used in the requests and responses for the PLDM monitoring and control commands defined in this specification.

Table 90 – Command Numbers

#	Command	Reference
Terminus Commands		
0x01	SetTID (see DSP0240)	See 16.1.
0x02	GetTID (see DSP0240)	See 16.2
0x03	GetTerminusUID	See 16.3.
0x04	SetEventReceiver	See 16.4.

#	Command	Reference
0x05	GetEventReceiver	See 16.5.
0x0A	PlatformEventMessage	See 16.6.
Numeric Sensor Commands		
0x10	SetNumericSensorEnable	See 18.1.
0x11	GetSensorReading	See 18.2.
0x12	GetSensorThresholds	See 18.3.
0x13	SetSensorThresholds	See 18.4.
0x14	RestoreSensorThresholds	See 18.5.
0x15	GetSensorHysteresis	See 18.6.
0x16	SetSensorHysteresis	See 18.7.
0x17	InitNumericSensor	See 18.8.
State Sensor Commands		
0x20	SetStateSensorEnables	See 20.1.
0x21	GetStateSensorReadings	See 20.2.
0x22	InitStateSensor	See 20.3.
PLDM Effecter Commands		
0x30	SetNumericEffecterEnable	See 22.1.
0x31	SetNumericEffecterValue	See 22.2.
0x32	GetNumericEffecterValue	See 22.3.
0x38	SetStateEffecterEnables	See 22.4.
0x39	SetStateEffecterStates	See 22.5.
0x3A	GetStateEffecterStates	See 22.6.
PLDM Event Log Commands		
0x40	GetPLDMEventLogInfo	See 23.1.
0x41	EnablePLDMEventLogging	See 23.2.
0x42	ClearPLDMEventLog	See 23.3.
0x43	GetPLDMEventLogTimestamp	See 23.4.
0x44	SetPLDMEventLogTimestamp	See 23.5.
0x45	ReadPLDMEventLog	See 23.6.
0x46	GetPLDMEventLogPolicyInfo	See 23.7.
0x47	SetPLDMEventLogPolicy	See 23.8.
0x48	FindPLDMEventLogEntry	See 23.9
PDR Repository Commands		
0x50	GetPDRRepositoryInfo	See 26.1.
0x51	GetPDR	See 26.2.
0x52	FindPDR	See 26.3.
0x58	RunInitAgent	See 26.4.

2646

ANNEX A
(informative)

Change Log

Version	Date	Author	Description
0.5.0		T. Slaight	review draft
0.7.0	2008/07/18	T. Slaight	review draft
0.8.0	2008/10/01	T. Slaight	review draft
0.9.6	2008/11/26	T. Slaight	version after cPubs review and edits with 11/24 markup changes incorporated
0.9.7	2008/12/04	T. Slaight	initial version for ballot. Includes added section on terminus messaging and validity field for Terminus Locator PDR.
1.0.0	2009/01/28	T. Slaight	draft standard
1.0.0	2009/03/16	T. Slaight	DMTF Standard