



Document Identifier: DSP0248

Date: 2017-01-10

Version: 1.1.1

1  
2  
3  
4

# 5 Platform Level Data Model (PLDM) for Platform 6 Monitoring and Control Specification

7 **Supersedes: 1.1.0**  
8 **Document Class: Normative**  
9 **Document Status: Published**  
10 **Document Language: en-US**  
11  
12

## 13 Copyright Notice

14 Copyright © 2009-2011, 2017 Distributed Management Task Force, Inc. (DMTF). All rights reserved.

15 DMTF is a not-for-profit association of industry members dedicated to promoting enterprise and systems  
16 management and interoperability. Members and non-members may reproduce DMTF specifications and  
17 documents, provided that correct attribution is given. As DMTF specifications may be revised from time to  
18 time, the particular version and release date should always be noted.

19 Implementation of certain elements of this standard or proposed standard may be subject to third party  
20 patent rights, including provisional patent rights (herein "patent rights"). DMTF makes no representations  
21 to users of the standard as to the existence of such rights, and is not responsible to recognize, disclose,  
22 or identify any or all such third party patent right, owners or claimants, nor for any incomplete or  
23 inaccurate identification or disclosure of such rights, owners or claimants. DMTF shall have no liability to  
24 any party, in any manner or circumstance, under any legal theory whatsoever, for failure to recognize,  
25 disclose, or identify any such third party patent rights, or for such party's reliance on the standard or  
26 incorporation thereof in its product, protocols or testing procedures. DMTF shall have no liability to any  
27 party implementing such standard, whether such implementation is foreseeable or not, nor to any patent  
28 owner or claimant, and shall have no liability or responsibility for costs or losses incurred if a standard is  
29 withdrawn or modified after publication, and shall be indemnified and held harmless by any party  
30 implementing the standard from any and all claims of infringement by a patent owner for such  
31 implementations.

32 For information about patents held by third-parties which have notified the DMTF that, in their opinion,  
33 such patent may relate to or impact implementations of DMTF standards, visit  
34 <http://www.dmtf.org/about/policies/disclosures.php>.

35 PCI-SIG, PCIe, and the PCI HOT PLUG design mark are registered trademarks or service marks of PCI-  
36 SIG.

37 All other marks and brands are the property of their respective owners.

38 This document's normative language is English. Translation into other languages is permitted.

CONTENTS

40 Foreword ..... 9

41 Introduction..... 10

42 1 Scope ..... 11

43 2 Normative references ..... 11

44 3 Terms and definitions ..... 12

45 4 Symbols and abbreviated terms..... 13

46 5 Conventions ..... 14

47 6 PLDM for Platform Monitoring and Control version ..... 15

48 7 PLDM for Platform Monitoring and Control overview ..... 15

49 8 PDR architecture ..... 16

50 8.1 General ..... 17

51 8.2 Primary PDR Repository and Device PDR repositories ..... 17

52 8.3 Use of PDRs ..... 17

53 9 Entities..... 20

54 9.1 Entity Identification Information..... 21

55 9.2 Entity Type and Entity IDs ..... 22

56 9.3 Entity Instance Numbers..... 23

57 9.4 Container ID..... 23

58 9.5 Use of Container ID in PDRs ..... 23

59 10 PLDM associations..... 24

60 10.1 Association examples ..... 24

61 10.2 Internal and External associations..... 24

62 10.3 Sensor/Effecter to Entity associations ..... 25

63 10.4 FRU Record Set to Entity associations..... 26

64 11 Entity Association PDRs..... 28

65 11.1 Physical to Physical Containment associations..... 28

66 11.2 Entity identification relationships between PDRs ..... 30

67 11.3 Linked Entity Association PDRs ..... 31

68 11.4 Logical containment associations ..... 32

69 11.5 Sensor/effecter associations with logical entities ..... 33

70 11.6 Merged entity associations ..... 34

71 11.7 Separation of logical and physical associations ..... 36

72 11.8 Designing association PDRs for monitoring and control ..... 36

73 11.9 Terminus associations ..... 37

74 11.10 Interrupt associations..... 40

75 12 PLDM terminus..... 41

76 12.1 TIDs, PLDM Terminus Handles, and Terminus Locator PDRs ..... 42

77 12.2 Requirements for unique TIDs..... 42

78 12.3 Terminus messaging requirements ..... 42

79 12.4 Terminus Locator PDRs..... 42

80 12.5 Enumerating termini..... 43

81 13 PLDM events..... 44

82 13.1 PLDM Event Messages ..... 45

83 13.2 PLDM Event Receiver..... 45

84 13.3 PLDM Event Logging..... 45

85 13.4 PLDM Event Log clearing policies ..... 45

86 13.5 Oldest and newest log entries ..... 46

87 13.6 Event Receiver Location ..... 46

88 13.7 PLDM Event Log entry formats..... 46

89 13.8 PLDM Platform Event Entry Data format ..... 47

90	13.9	OEM Timestamped Event Entry Data format .....	48
91	13.10	OEM Event Entry Data format .....	48
92	14	Discovery Agent .....	48
93	14.1	Assignment of TIDs and Event Receiver location .....	49
94	14.2	UUIDs for devices in hot-plug or add-in card applications.....	50
95	14.3	UID implementation .....	50
96	14.4	More than one terminus in a device.....	50
97	14.5	Examples of PDR and UUID use with add-in cards .....	50
98	15	Initialization Agent .....	53
99	15.1	General .....	53
100	15.2	PLDM and power state interaction.....	53
101	15.3	RunInitAgent command .....	53
102	15.4	Recommended Initialization Agent steps.....	54
103	16	Terminus and event commands.....	54
104	16.1	SetTID command.....	55
105	16.2	GetTID command.....	55
106	16.3	GetTerminusUID command .....	55
107	16.4	SetEventReceiver command .....	56
108	16.5	GetEventReceiver command.....	57
109	16.6	PlatformEventMessage command.....	58
110	16.7	eventData format for sensorEvent .....	59
111	16.8	eventData format for effectorEvent.....	60
112	17	PLDM Numeric Sensors.....	61
113	17.1	Sensor readings, data sizes .....	61
114	17.2	Units and reading conversion .....	61
115	17.3	Reading-only or threshold-based numeric sensors .....	62
116	17.4	Readable and settable thresholds .....	62
117	17.5	Update / polling intervals and states updates .....	62
118	17.6	Thresholds, Present State, and Event State .....	62
119	17.7	Manual re-arm and auto re-arm sensors .....	64
120	17.8	Event message generation .....	64
121	17.9	Threshold values and hysteresis .....	64
122	18	PLDM Numeric Sensor commands.....	66
123	18.1	SetNumericSensorEnable command.....	66
124	18.2	GetSensorReading command .....	67
125	18.3	GetSensorThresholds command.....	70
126	18.4	SetSensorThresholds command .....	71
127	18.5	RestoreSensorThresholds command .....	72
128	18.6	GetSensorHysteresis command.....	72
129	18.7	SetSensorHysteresis command .....	73
130	18.8	InitNumericSensor command .....	74
131	19	PLDM State Sensors.....	75
132	20	PLDM State Sensor commands.....	76
133	20.1	SetStateSensorEnables command.....	76
134	20.2	GetStateSensorReadings command .....	77
135	20.3	InitStateSensor command .....	78
136	21	PLDM effecters.....	80
137	21.1	PLDM State Effecters .....	80
138	21.2	PLDM Numeric Effecters .....	80
139	21.3	Effector semantics .....	80
140	21.4	PLDM and OEM effector semantic IDs.....	81
141	22	PLDM effector commands.....	81
142	22.1	SetNumericEffectorEnable command.....	82
143	22.2	SetNumericEffectorValue command.....	83

144	22.3	GetNumericEffectorValue command .....	84
145	22.4	SetStateEffectorEnables command.....	85
146	22.5	SetStateEffectorStates command.....	86
147	22.6	GetStateEffectorStates command .....	87
148	23	PLDM Event Log commands .....	88
149	23.1	GetPLDMEventLogInfo command .....	89
150	23.2	EnablePLDMEventLogging command.....	91
151	23.3	ClearPLDMEventLog command .....	91
152	23.4	GetPLDMEventLogTimestamp command .....	92
153	23.5	SetPLDMEventLogTimestamp command.....	93
154	23.6	ReadPLDMEventLog command .....	94
155	23.7	GetPLDMEventLogPolicyInfo command .....	96
156	23.8	SetPLDMEventLogPolicy command.....	99
157	23.9	FindPLDMEventLogEntry command .....	101
158	24	PLDM State Sets .....	102
159	25	Platform Descriptor Records (PDRs) .....	102
160	25.1	PDR Repository updates .....	103
161	25.2	Internal storage and organization of PDRs.....	103
162	25.3	PDR types.....	103
163	25.4	PDR record handles.....	103
164	25.5	Accessing PDRs .....	103
165	26	PDR Repository commands.....	103
166	26.1	GetPDRRepositoryInfo command .....	104
167	26.2	GetPDR command.....	106
168	26.3	FindPDR command.....	109
169	26.4	RunInitAgent command .....	115
170	27	PDR definitions.....	115
171	27.1	Sensor types .....	116
172	27.2	Effector types .....	116
173	27.3	State sets .....	116
174	27.4	Sensor and effector units.....	116
175	27.5	Counters .....	119
176	27.6	Accuracy, tolerance, resolution, and offset.....	119
177	27.7	Numeric reading conversion formula .....	125
178	27.8	Numeric effector conversion formula .....	126
179	28	Platform Descriptor Record (PDR) formats.....	126
180	28.1	Common PDR header format .....	126
181	28.2	PDR type values .....	127
182	28.3	Terminus Locator PDR .....	128
183	28.4	Numeric Sensor PDR.....	130
184	28.5	Numeric Sensor Initialization PDR.....	136
185	28.6	State Sensor PDR.....	137
186	28.7	State Sensor Initialization PDR.....	139
187	28.8	Sensor Auxiliary Names PDR.....	142
188	28.9	OEM Unit PDR.....	143
189	28.10	OEM State Set PDR .....	144
190	28.11	Numeric Effector PDR.....	146
191	28.12	Numeric Effector Initialization PDR.....	151
192	28.13	State Effector PDR.....	152
193	28.14	State Effector Initialization PDR.....	153
194	28.15	Effector Auxiliary Names PDR.....	156
195	28.16	OEM Effector Semantic PDR.....	157
196	28.17	Entity Association PDR .....	158
197	28.18	Entity Auxiliary Names PDR .....	159
198	28.19	OEM EntityID PDR.....	160

199 28.20 Interrupt Association PDR ..... 161  
 200 28.21 Event Log PDR ..... 162  
 201 28.22 FRU Record Set PDR ..... 163  
 202 28.23 OEM Device PDR ..... 164  
 203 28.24 OEM PDR ..... 165  
 204 29 Timing..... 166  
 205 30 Command numbers..... 166  
 206 ANNEX A (informative) Change log..... 168  
 207 Bibliography ..... 169  
 208

209 **Figures**

210 Figure 1 – PLDM used for access only ..... 18  
 211 Figure 2 – PLDM with device PDRs..... 19  
 212 Figure 3 – PLDM with PDRs for subsystem..... 20  
 213 Figure 4 – Entity Identification Information..... 21  
 214 Figure 5 – Entity Identification Information format ..... 21  
 215 Figure 6 – Entity Identification Information in a Numeric Sensor PDR ..... 25  
 216 Figure 7 – Entity Identification Information in a FRU Record Set PDR..... 27  
 217 Figure 8 – Physical Containment Entity Association PDR..... 29  
 218 Figure 9 – containerID relationships ..... 30  
 219 Figure 10 – Entity identification relationship between PDRs ..... 31  
 220 Figure 11 – Linked Entity Association PDRs ..... 32  
 221 Figure 12 – Logical Containment PDR ..... 33  
 222 Figure 13 – Sensor/effector to logical entity association ..... 34  
 223 Figure 14 – Merged Entity Association PDR example ..... 35  
 224 Figure 15 – Block diagram for Merged Entity Association PDR example..... 36  
 225 Figure 16 – TID and PLDM Terminus Handle associations..... 38  
 226 Figure 17 – Block diagram of Terminus to Sensor associations..... 39  
 227 Figure 18 – Received interrupt association example ..... 41  
 228 Figure 19 – Example of TID and PLDM Terminus Handle relationships ..... 43  
 229 Figure 20 – Hot-plug add-in card with single PLDM terminus ..... 51  
 230 Figure 21 – Hot-plug add-in card with multiple PLDM termini ..... 52  
 231 Figure 22 – Numeric sensor threshold and hysteresis relationships ..... 65  
 232 Figure 23 – Accuracy, tolerance, and resolution example ..... 120  
 233 Figure 24 – Figuring resolution from the design ..... 123  
 234

235 **Tables**

236 Table 1 – PLDM monitoring and control data types..... 14  
 237 Table 2 – Parts of the Entity Identification Information format..... 21  
 238 Table 3 – Field & value descriptions for Entity Identification Information in a Numeric Sensor PDR..... 25  
 239 Table 4 – Field and value descriptions for Entity Identification Information in a FRU Record Set PDR..... 27  
 240 Table 5 – PLDM Event Log clearing policies ..... 45  
 241 Table 6 – PLDM Event Log entry format..... 47

242	Table 7 – Platform Event Entry Data format .....	47
243	Table 8 – OEM Timestamped Event Entry Data format .....	48
244	Table 9 – OEM Event Entry Data format .....	48
245	Table 10 – Terminus commands .....	54
246	Table 11 – GetTerminusUID command format .....	55
247	Table 12 – SetEventReceiver command format .....	56
248	Table 13 – GetEventReceiver command format .....	57
249	Table 14 – PlatformEventMessage command format .....	58
250	Table 15 – sensorEvent class eventData format .....	59
251	Table 16 – effectorEvent class eventData format .....	60
252	Table 17 – Threshold severity levels .....	63
253	Table 18 – Numeric Sensor commands .....	66
254	Table 19 – SetNumericSensorEnable command format .....	66
255	Table 20 – GetSensorReading command format .....	67
256	Table 21 – GetSensorThresholds command format .....	70
257	Table 22 – SetSensorThresholds command format .....	71
258	Table 23 – RestoreSensorThresholds command format .....	72
259	Table 24 – GetSensorHysteresis command format .....	73
260	Table 25 – SetSensorHysteresis command format .....	74
261	Table 26 – InitNumericSensor command format .....	75
262	Table 27 – State Sensor commands .....	76
263	Table 28 – SetStateSensorEnables command format .....	76
264	Table 29 – SetStateSensorEnables opField format .....	77
265	Table 30 – GetStateSensorReadings command format .....	77
266	Table 31 – GetStateSensorReadings stateField format .....	78
267	Table 32 – InitStateSensor command format .....	79
268	Table 33 – InitStateSensor initField format .....	79
269	Table 34 – Categories for effector semantics .....	81
270	Table 35 – State and Numeric Effector commands .....	82
271	Table 36 – SetNumericEffectorEnable command format .....	83
272	Table 37 – SetNumericEffectorValue command format .....	83
273	Table 38 – GetNumericEffectorValue command format .....	84
274	Table 39 – SetStateEffectorEnables command format .....	85
275	Table 40 – SetStateEffectorEnables opField format .....	86
276	Table 41 – SetStateEffectorStates command format .....	86
277	Table 42 – SetStateEffectorStates stateField format .....	87
278	Table 43 – GetStateEffectorStates command format .....	87
279	Table 44 – GetStateEffectorStates stateField format .....	88
280	Table 45 – PLDM Event Log commands .....	88
281	Table 46 – GetPLDMEventLogInfo command format .....	89
282	Table 47 – EnablePLDMEventLogging command format .....	91
283	Table 48 – ClearPLDMEventLog command format .....	92
284	Table 49 – GetPLDMEventLogTimestamp command format .....	92
285	Table 50 – SetPLDMEventLogTimestamp command format .....	93
286	Table 51 – ReadPLDMEventLog command format .....	95
287	Table 52 – PLDMEventLogData format .....	96
288	Table 53 – GetPLDMEventLogPolicyInfo command format .....	97
289	Table 54 – SetPLDMEventLogPolicy command format .....	99

290 Table 55 – FindPLDMEventLogEntry command format ..... 101

291 Table 56 – PDR Repository commands..... 104

292 Table 57 – GetPDRRepositoryInfo command format ..... 105

293 Table 58 – GetPDR command format..... 106

294 Table 59 – FindPDR command format ..... 110

295 Table 60 – FindPDR Command Parameter Format Numbers..... 113

296 Table 61 – FindPDR Command Parameter Formats..... 114

297 Table 62 – RunInitAgent command format ..... 115

298 Table 63 – sensorUnits enumeration ..... 117

299 Table 64 – Common PDR header format ..... 126

300 Table 65 – PDR Type Values..... 127

301 Table 66 – Terminus Locator PDR format ..... 128

302 Table 67 – Numeric Sensor PDR format ..... 130

303 Table 68 – Numeric Sensor Initialization PDR format ..... 136

304 Table 69 – State Sensor PDR format ..... 137

305 Table 70 – State Sensor possible states fields format..... 138

306 Table 71 – State Sensor Initialization PDR format ..... 139

307 Table 72 – Sensor Auxiliary Names PDR format..... 142

308 Table 73 – OEM Unit PDR format..... 144

309 Table 74 – OEM State Set PDR format ..... 145

310 Table 75 – OEM State Value Record format ..... 146

311 Table 76 – Numeric Effector PDR format ..... 147

312 Table 77 – Numeric Effector Initialization PDR format ..... 151

313 Table 78 – State Effector PDR format ..... 152

314 Table 79 – State Effector Possible States Fields format ..... 153

315 Table 80 – State Effector Initialization PDR format ..... 154

316 Table 81 – Effector Auxiliary Names PDR format..... 156

317 Table 82 – OEM Effector Semantic PDR format..... 157

318 Table 83 – Entity Association PDR format..... 158

319 Table 84 – Entity Auxiliary Names PDR format ..... 159

320 Table 85 – OEM EntityID PDR format ..... 160

321 Table 86 - Interrupt Association PDR format ..... 161

322 Table 87 – Event Log PDR format ..... 162

323 Table 88 – FRU Record Set PDR format..... 164

324 Table 89 – OEM Device PDR format ..... 165

325 Table 90 – OEM PDR format ..... 166

326 Table 91 – Monitoring and control timing specifications ..... 166

327 Table 92 – Command numbers ..... 166

328

329

330

## Foreword

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

334 DMTF is a not-for-profit association of industry members dedicated to promoting enterprise and systems  
335 management and interoperability. For information about the DMTF, see <http://www.dmtf.org>.

## 336 Acknowledgments

337 The DMTF acknowledges the following individuals for their contributions to this document:

### 338 Editor:

- 339 • Patrick Schoeller – Hewlett Packard Enterprise

### 340 Contributors:

- 341 • Alan Berenbaum - SMSC
- 342 • Patrick Caporale – Lenovo
- 343 • Phil Chidester - Dell
- 344 • Hoan Do - Broadcom Corporation
- 345 • Yuval Itkin – Mellanox Technologies
- 346 • Ed Klodnicki - IBM
- 347 • John Leung - Intel Corporation
- 348 • Eliel Louzoun – Intel Corporation
- 349 • Hemal Shah - Broadcom Corporation
- 350 • Tom Slaight – Intel Corporation

351

352

## Introduction

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

### 358 **Document conventions**

#### 359 **Typographical conventions**

360 The following typographical conventions are used in this document:

- 361 • Document titles are marked in *italics*.
- 362 • Important terms that are used for the first time are marked in *italics*.

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

## 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 clause 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 or versioned references, only the edition cited (including any corrigenda or DMTF update versions) applies. For references without a date or version, the latest published edition of the referenced document (including any corrigenda or DMTF update versions) applies.

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

DMTF DSP0236, *MCTP Base Specification 1.0*,  
[http://dmtof.org/sites/default/files/standards/documents/DSP0236\\_1.0.pdf](http://dmtof.org/sites/default/files/standards/documents/DSP0236_1.0.pdf)

DMTF DSP0240, *Platform Level Data Model (PLDM) Base Specification 1.0*,  
[http://dmtof.org/sites/default/files/standards/documents/DSP0240\\_1.0.0.pdf](http://dmtof.org/sites/default/files/standards/documents/DSP0240_1.0.0.pdf)

DMTF DSP0241, *Platform Level Data Model (PLDM) Over MCTP Binding Specification 1.0*,  
[http://dmtof.org/sites/default/files/standards/documents/DSP0241\\_1.0.pdf](http://dmtof.org/sites/default/files/standards/documents/DSP0241_1.0.pdf)

DMTF DSP0245, *Platform Level Data Model (PLDM) IDs and Codes Specification 1.0*,  
[http://dmtof.org/sites/default/files/standards/documents/DSP0245\\_1.0.pdf](http://dmtof.org/sites/default/files/standards/documents/DSP0245_1.0.pdf)

DMTF DSP0249, *Platform Level Data Model (PLDM) State Sets Specification 1.0*,  
[http://dmtof.org/sites/default/files/standards/documents/DSP0249\\_1.0.pdf](http://dmtof.org/sites/default/files/standards/documents/DSP0249_1.0.pdf)

- 401 DMTF DSP0257, *Platform Level Data Model (PLDM) FRU Data Specification 1.0*,  
402 [http://dmtf.org/sites/default/files/standards/documents/DSP0257\\_1.0.pdf](http://dmtf.org/sites/default/files/standards/documents/DSP0257_1.0.pdf)
- 403 IETF RFC2781, *UTF-16, an encoding of ISO 10646*, February 2000,  
404 <http://www.ietf.org/rfc/rfc2781.txt>
- 405 IETF RFC3629, *UTF-8, a transformation format of ISO 10646*, November 2003,  
406 <http://www.ietf.org/rfc/rfc3629.txt>
- 407 IETF RFC4122, *A Universally Unique Identifier (UUID) URN Namespace*, July 2005,  
408 <http://www.ietf.org/rfc/rfc4122.txt>
- 409 IETF RFC4646, *Tags for Identifying Languages*, September 2006,  
410 <http://www.ietf.org/rfc/rfc4646.txt>
- 411 ISO 8859-1, *Final Text of DIS 8859-1, 8-bit single-byte coded graphic character sets — Part 1: Latin  
412 alphabet No. 1*, February 1998
- 413 ISO/IEC Directives, Part 2, *Rules for the structure and drafting of International Standards*,  
414 <http://isotc.iso.org/livelink/livelink.exe?func=ll&objId=4230456&objAction=browse&sort=subtype>

### 415 3 Terms and definitions

416 In this document, some terms have a specific meaning beyond the normal English meaning. Those terms  
417 are defined in this clause.

418 The terms "shall" ("required"), "shall not," "should" ("recommended"), "should not" ("not recommended"),  
419 "may," "need not" ("not required"), "can" and "cannot" in this document are to be interpreted as described  
420 in [ISO/IEC Directives, Part 2](#), Annex H. The terms in parenthesis are alternatives for the preceding term,  
421 for use in exceptional cases when the preceding term cannot be used for linguistic reasons. Note that  
422 [ISO/IEC Directives, Part 2](#), Annex H specifies additional alternatives. Occurrences of such additional  
423 alternatives shall be interpreted in their normal English meaning.

424 The terms "clause," "subclause," "paragraph," and "annex" in this document are to be interpreted as  
425 described in [ISO/IEC Directives, Part 2](#), Clause 5.

426 The terms "normative" and "informative" in this document are to be interpreted as described in [ISO/IEC  
427 Directives, Part 2](#), Clause 3. In this document, clauses, subclauses, or annexes labeled "(informative)" do  
428 not contain normative content. Notes and examples are always informative elements.

429 Refer to [DSP0240](#) for terms and definitions that are used across the PLDM specifications. For the  
430 purposes of this document, the following additional terms and definitions apply.

#### 431 3.1

432 contained entity

433 an entity that is contained within a container entity

#### 434 3.2

435 container entity

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

#### 437 3.3

438 container ID

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

441 **3.4**

442 containing entity

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

444 **3.5**445 **entity**

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

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

453 **3.6**454 **Entity ID**

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

457 **3.7**458 **Entity Instance Number**

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

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

462 **3.8**463 **Entity Type**

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

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

467 **3.9**468 **Platform Descriptor Record**469 **PDR**

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

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

476 **4 Symbols and abbreviated terms**

477 Refer to [DSP0240](#) for symbols and abbreviated terms that are used across the PLDM specifications. For  
478 the purposes of this document, the following additional symbols and abbreviated terms apply.

479 **4.1**480 **CIM**

481 Common Information Model

- 482 **4.2**
- 483 **EID**
- 484 Endpoint ID
- 485 **4.3**
- 486 **IANA**
- 487 Internet Assigned Numbers Authority
- 488 **4.4**
- 489 **MAP**
- 490 Manageability Access Point
- 491 **4.5**
- 492 **MCTP**
- 493 Management Component Transport Protocol
- 494 **4.6**
- 495 **PDR**
- 496 Platform Descriptor Record
- 497 **4.7**
- 498 **PLDM**
- 499 Platform Level Data Model
- 500 **4.8**
- 501 **TID**
- 502 Terminus ID

503 **5 Conventions**

504 Refer to [DSP0240](#) for conventions, notations, and data types that are used across the PLDM  
 505 specifications. The following data types are also defined for use in this specification:

506 **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 <a href="#">RFC3629</a> . 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 <a href="#">RFC3629</a> 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 <a href="#">RFC2781</a> . 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.

Data type	Interpretation
strUTF-16LE	A null (0x0000) terminated, UTF-16, "little endian" encoded string per <a href="#">RFC2781</a> . 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 <a href="#">RFC2781</a> . 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.

## 507 6 PLDM for Platform Monitoring and Control version

508 The version of this *Platform Level Data Model (PLDM) for Platform Monitoring and Control Specification*  
 509 shall be 1.1.1 (major version number 1, minor version number 1, update version number 1, and no alpha  
 510 version).

511 For the GetPLDMVersion command described in [DSP0240](#), the version of this specification is reported  
 512 using the encoding as 0xF1F1F100.

513 If the endpoint declares support for PLDM for Platform Monitoring and Control version 1.1.1 or later  
 514 specification versions, all previous versions (e.g., 1.1.0) should not be listed as supported in the  
 515 GetPldmVersion command because of the sensorID (Numeric Sensor PDR) or the effectorID (Numeric  
 516 Effector PDR) size change from uint8 / uint16,

## 517 7 PLDM for Platform Monitoring and Control overview

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

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

- 527 • sensors and effecters

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

- 532 – Numeric sensors provide a number that is representative of a monitored value that can be  
 533 expressed using units such as degrees Celsius, volts, and amps.
- 534 – State sensors are used for accessing a number from an enumeration that represents the  
 535 state of a monitored entity. Different states are enumerated in predefined sets called state  
 536 sets. Example state sets can include states for Availability (enabled, disabled, shut down,  
 537 and so on), Door State (open, closed), Presence (present, not present) and so on. The  
 538 values for State Sets are defined in [DSP0249](#).

- 539           – Numeric effecters are used for setting a number that configures or controls the operation of  
540           a controlled entity. Like numeric sensors, numeric effecters also use units such as degrees  
541           Celsius, volts, and amps.
- 542           – State effecters are used for setting a number that configures or controls a state that is  
543           associated with a controlled entity. State effecters draw upon the same state set definitions  
544           as state sensors.
- 545           • Platform Descriptor Records (PDRs)
- 546           PDRs are data structures that can provide semantic information for sensors and effecters, their  
547           relationship to the entities that are being monitored or controlled, and associations that exist  
548           between entities within the platform. The PDRs also include information that describes the  
549           presence and location of different PLDM termini. This information can be used to discover the  
550           population of sensors and effecters and how to access them by using PLDM messaging. The  
551           information also facilitates building Common Information Model objects and associations for the  
552           sensors, effecters, and platform entities. PDRs can also hold information that is used to initialize  
553           sensors and effecters. PDRs are collected into a logical storage area called a PDR Repository.  
554           A central PDR Repository called the Primary PDR Repository can be used to hold an  
555           aggregation of all PDR information within the PLDM subsystem.
- 556           • platform events
- 557           This specification defines messages that are asynchronously sent upon particular state changes  
558           that occur within sensors, effecters, or the PLDM platform management subsystem. The  
559           messages are delivered to a central function called the PLDM Event Receiver.
- 560           • platform event logging
- 561           The specification includes the definition of a central, non-volatile storage function called the  
562           PLDM Event Log that can be used to log PLDM Event Messages. The specification also defines  
563           messages for accessing and maintaining the PLDM Event Log.
- 564           • support functions
- 565           This specification also includes the definition of support functions as required to support the  
566           initialization of sensors and effecters, and the maintenance of PDRs in the Primary PDR  
567           Repository. The main support functions are the Discovery Agent and the Initialization Agent.
- 568           – The Discovery Agent function is responsible for keeping the Primary PDR information up to  
569           date if entities are added, relocated, or removed from the PLDM platform management  
570           subsystem. The Discovery Agent function is also responsible for setting the Event Receiver  
571           location into PLDM termini that support PLDM monitoring and control messages.
- 572           – The Initialization Agent function is responsible for initializing sensors and effecters that may  
573           require initialization or re-initialization upon state changes to the PLDM terminus or the  
574           managed system, such as system hard resets, the terminus coming online for PLDM  
575           communication, and so on.
- 576           • OEM/vendor-specific functions
- 577           This specification includes provisions for supporting OEM or vendor-specific functions and  
578           semantic information. This includes the ability to define OEM units for numeric sensors or  
579           effecters, OEM state sets, and OEM entity types. An OEM PDR type is also available as an  
580           opaque storage mechanism for holding OEM-defined data in PDR Repositories.

## 581 **8 PDR architecture**

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

## 584 8.1 General

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

591 This separation provides several benefits:

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

## 609 8.2 Primary PDR Repository and Device PDR repositories

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

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

## 619 8.3 Use of PDRs

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

### 622 8.3.1 PLDM for access only

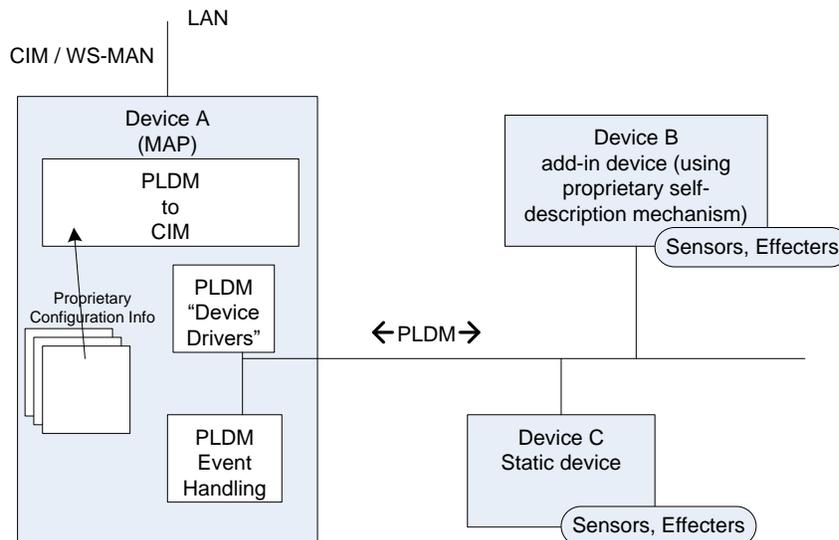
623 Figure 1 shows an implementation that does not use PDRs. PLDM is used only as a mechanism for  
624 accessing monitoring and control functions; it is not used for providing semantic information about those  
625 functions.

626 In this example, Device A provides a DMTF Manageability Access Point (MAP) function that makes  
627 platform information available over a network using CIM as the data model and WS-MAN as the transport

628 protocol for CIM. In this example, PLDM is used only for accessing the functions in Devices B and C, and  
 629 for Devices B and C to send PLDM Event Messages to Device A.

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

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



643

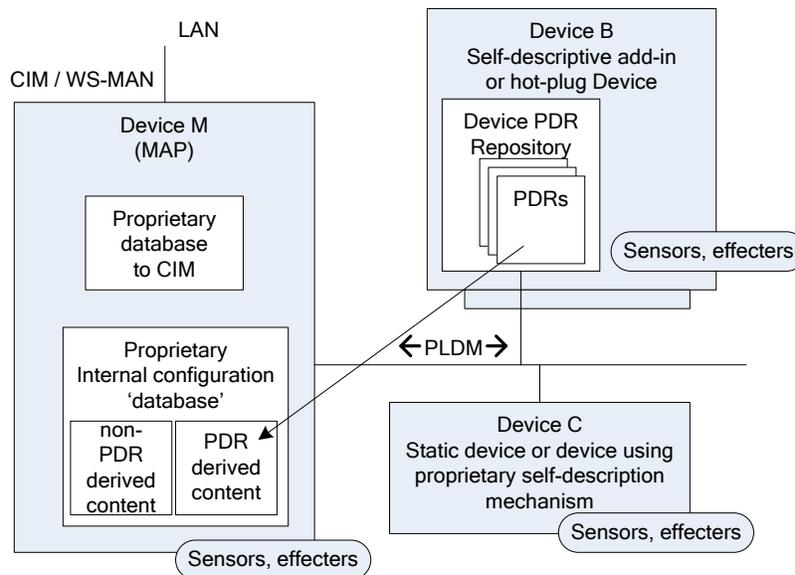
644

Figure 1 – PLDM used for access only

645 **8.3.2 PLDM with PDRs for add-in devices**

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

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



661

662

Figure 2 – PLDM with device PDRs

663 **8.3.3 PLDM with Primary PDR Repository**

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

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

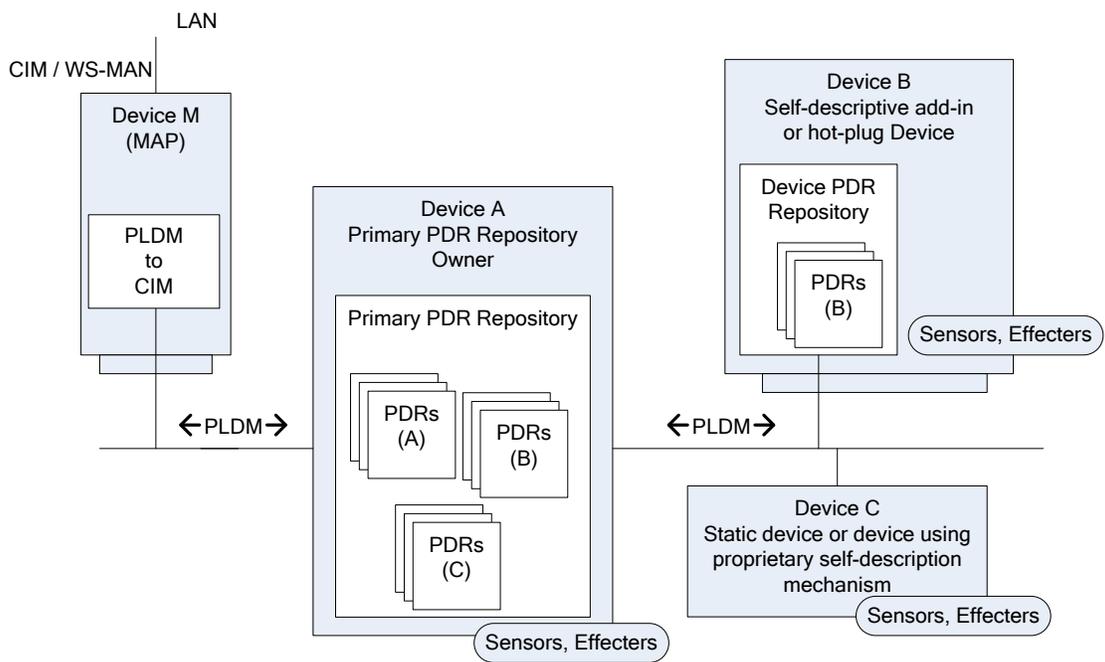
671 Such an implementation enables the MAP to come from one party while the platform management  
 672 subsystem comes from another without the need to explicitly configure the MAP with the semantic  
 673 information for the subsystem. For example, the platform management subsystem represented through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

678 Device A maintains the Primary PDR Repository that includes information about static sensors and  
 679 effectors (such as those within Device C and within Device A itself) and integrates that information into  
 680 the overall view of the platform management subsystem held in the Primary PDR Repository. This

681 involves discovering and extracting PDRs from "Self-descriptive" devices such as Device B, and  
 682 synthesizing additional PDRs, such as association and Terminus Locator PDRs, in order to integrate the  
 683 PDRs into the repository and create a coherent view of the overall subsystem.

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

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



692

693

**Figure 3 – PLDM with PDRs for subsystem**

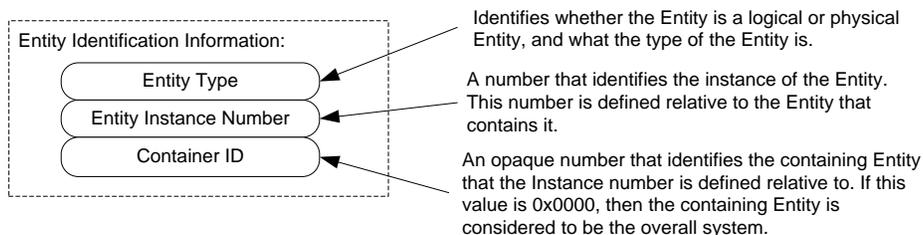
694 **9 Entities**

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

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

702 **9.1 Entity Identification Information**

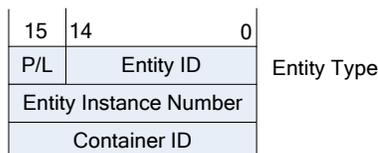
703 Individual entities are identified within PLDM PDRs using three fields: Entity Type, Entity Instance  
 704 Number, and Container ID. Together, these fields are referred to as the Entity Identification Information.  
 705 Figure 4 presents an overview of the meaning of the individual fields. The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



707

708 **Figure 4 – Entity Identification Information**

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



713

714 **Figure 5 – Entity Identification Information format**

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

716 **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 <a href="#">DSP0249</a> 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

Part	Description
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.

## 717 9.2 Entity Type and Entity IDs

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

722 The different general types of entities within PLDM are identified using an enumeration value referred to  
 723 as an "Entity ID." The different types of standardized entities and their corresponding Entity ID values are  
 724 specified in [DSP0249](#).

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

### 726 9.2.1 Vendor-specific (OEM) Entity IDs

727 The Entity ID values include a special range of values for identifying vendor- or OEM-specific entities. In  
 728 order to be interpreted, these values must be accompanied by an OEM EntityID PDR that identifies which  
 729 vendor defined the entity and, optionally, a string or strings that provide the name for the entity. Refer to  
 730 28.19 for additional information about how OEM Entity IDs are used.

### 731 9.2.2 Logical and physical entities

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

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

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

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

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

### 754 **9.3 Entity Instance Numbers**

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

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

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

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

### 768 **9.4 Container ID**

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

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

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

### 784 **9.5 Use of Container ID in PDRs**

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

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

796 The overall system is considered to be the top of the hierarchy of containment and thus does not appear  
797 as a contained entity in any Entity Association PDR. In this case, there is no explicit Entity Association

798 PDR for the overall system. A special value (0x0000) is used for the Container ID to indicate when the  
799 overall system is the container entity.

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

## 804 10 PLDM associations

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

### 807 10.1 Association examples

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

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

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

### 831 10.2 Internal and External associations

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

837 Therefore, Internal Associations require a common field to be defined among the elements that are  
838 associated with each other. The Internal Association mechanism is efficient, but not readily extensible,  
839 because a new type of association would typically require new fields to be defined and added to the  
840 PDRs that are to be associated with one another, along with specifications that document how the field is  
841 used to form links to other records. Because the fields that support Internal Associations must be pre-

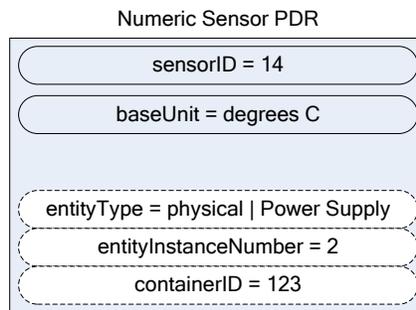
842 defined as part of the PDR, internal associations are generally used only for the most fundamental and  
 843 common types of associations. For other types of associations, a more generalized mechanism called  
 844 "External Associations" is provided.

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

852 **10.3 Sensor/Effecter to Entity associations**

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

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



864

865 **Figure 6 – Entity Identification Information in a Numeric Sensor PDR**

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

867 **Table 3 – Field & value descriptions for Entity Identification Information in a Numeric 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.

Field and value	Description
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 67 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 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 clause 11.

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

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

873 The information is based on the following fields:

874 container ID | entityType | entityInstanceNumber | baseUnit | sensorInstanceNumber

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

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

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

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

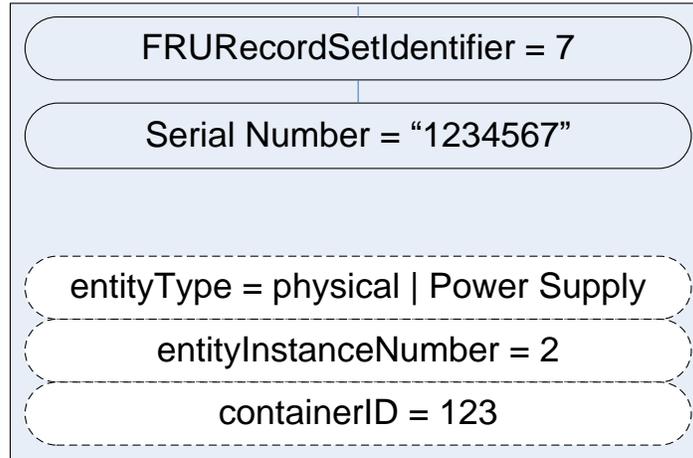
881 If the Container ID is for an entity other than system, the Container ID information can be used to locate  
 882 the Entity Association PDR that identifies the containing entity for the sensor.

883 **10.4 FRU Record Set to Entity associations**

884 Each FRU Record Set that is described using PDRs has a corresponding FRU Record Set PDR that  
 885 provides semantic information for individual FRUs, such as information that identifies which terminus is  
 886 associated with the FRU Record Set. Included in this information is Entity Identification Information for the  
 887 entity that is associated with the FRU Record Set.

888 Figure 7 shows a subset of the fields in the FRU Record Set PDR for a PLDM FRU Record Set. The  
 889 Entity Identification Information is represented by the fields highlighted with dashed lines.

FRU Record Set PDR



890

891

**Figure 7 – Entity Identification Information in a FRU Record Set PDR**

892

Table 4 describes the meaning of the fields shown in Figure 7.

893

**Table 4 – Field and value descriptions for Entity Identification Information in a FRU Record Set PDR**

894

Field and value	Description
FRURecordSetIdentifier = 7	All FRU Record Sets within a given terminus have unique Record Set Identifier. This field holds a value that is used in commands such as GetFRURecordByOption to access the particular Record Set within the terminus. The FRURecordSetIdentifier number is used only for accessing the FRU Record Set. The example shows that the value 7 would be used in commands to access this FRU Record Set.
Serial Number = "1234567"	The Serial Number field identifies the serial number of the FRU Record Set.
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 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 clause 11.

895 The details included in Table 4 provide a significant amount of the information that is typically used for  
896 identifying a FRU Record Set and its use within a management subsystem. For example, a string that  
897 contains the following identification information for the FRU Record Set could be derived from the FRU  
898 Record Set PDR without referring to any additional PDRs:

899 "Entity(123) physical power supply 2 Serial Number"

900 The information is based on the following fields:

901 container ID | entityType | entityInstanceNumber | Serial Number

902 Note that an application would typically use just Serial Number to make it more readable. For example:

903 "Entity(123) physical power supply 2 Serial Number"

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

906 "System Physical Power Supply 2 Serial Number"

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

## 909 **11 Entity Association PDRs**

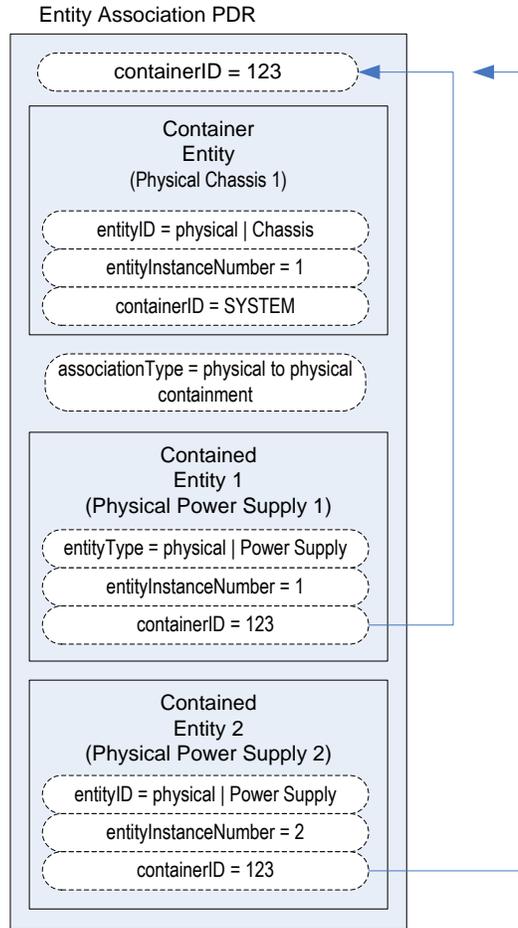
910 Entity Association PDRs associate entities with one another.

### 911 **11.1 Physical to Physical Containment associations**

912 One of the most common associations is the "physical containment association." This association is used  
913 to indicate that a physical entity contains one or more other physical entities. For example, the  
914 association can be used to represent that a physical chassis contains multiple power supplies. Figure 8  
915 shows an example of selected fields within an Entity Association PDR that describes a physical  
916 containment association.

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

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



927

928

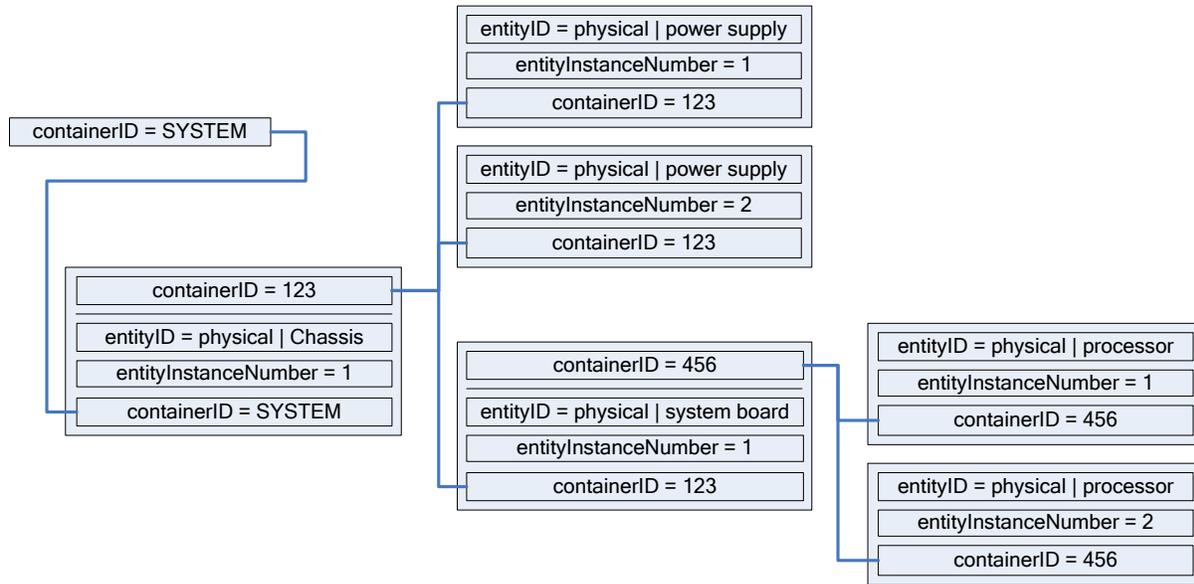
**Figure 8 – Physical Containment Entity Association PDR**

929

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 8, containerID 123 represents physical Chassis 1 (where instance number 1 is defined relative to SYSTEM).

934

Figure 9 provides an illustration of how the containerID value links entities in a containment hierarchy.



935

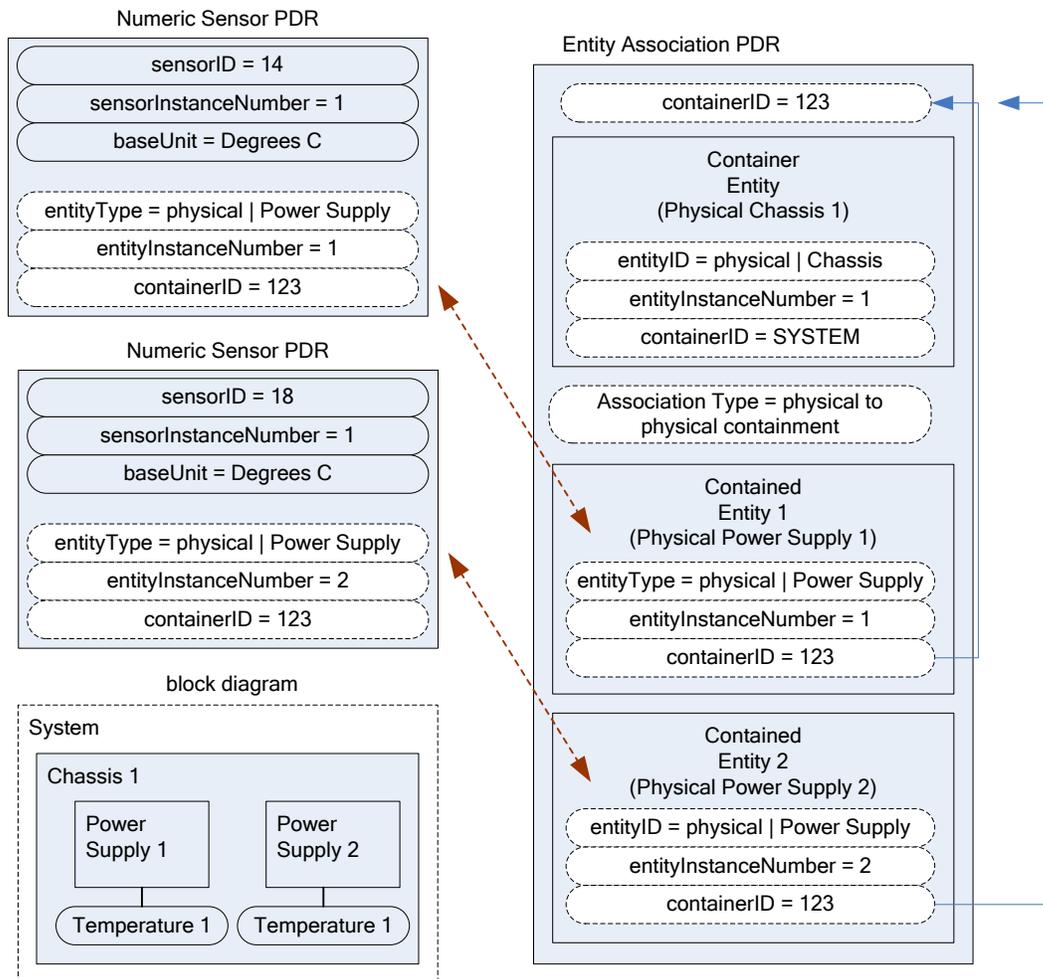
936

Figure 9 – containerID relationships

937 **11.2 Entity identification relationships between PDRs**

938 Figure 10 shows the kinds of association relationships that emerge when the PDRs are used in  
 939 combination. The Numeric Sensor PDR in this example has Entity Identification Information that  
 940 corresponds to "Power Supply 2." The containerID information in that Numeric Sensor PDR corresponds  
 941 to the containerID that is linked to Physical Chassis 1 through the Entity Association PDR. Note that  
 942 Physical Chassis 1 is identified as being contained only by the overall system. Hence, its containerID is  
 943 SYSTEM.

944 Putting this information together yields a view of the system that is represented by the block diagram  
 945 shown in Figure 10, which shows that the system contains a physical chassis that in turn contains two  
 946 physical power supplies, and that each physical power supply has a temperature sensor associated with  
 947 it. The two temperature sensors are both referred to as "Temperature 1" because their  
 948 sensorInstanceNumber is defined relative to the power supply that is being monitored.



949

950

**Figure 10 – Entity identification relationship between PDRs**

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

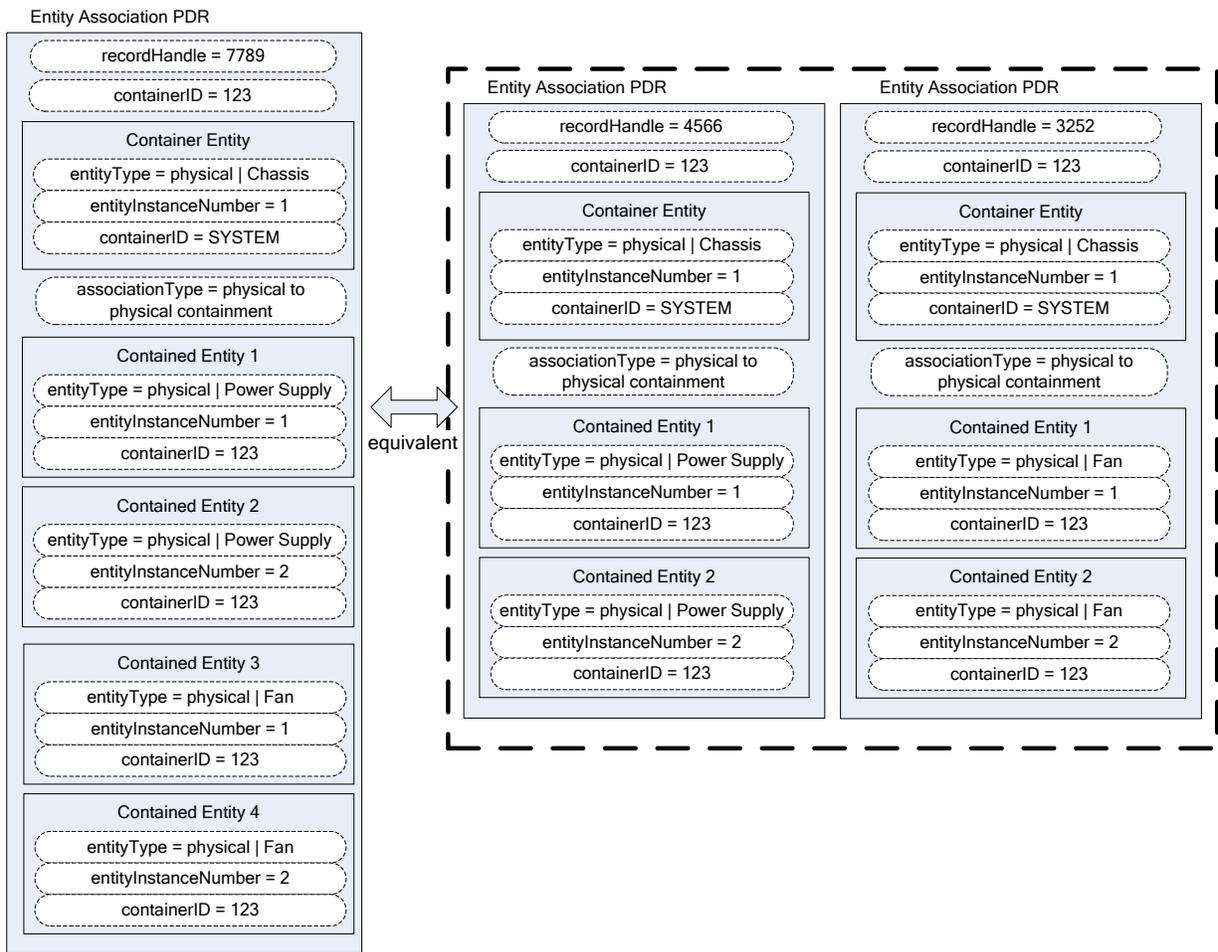
955 **11.3 Linked Entity Association PDRs**

956 Certain types of PDRs can be linked together using an Internal Association to form the equivalent of a  
 957 single joint PDR. In Figure 11, the two Entity Association PDRs on the right are implicitly linked together  
 958 by sharing the same containerID value. (Note that in Figure 11, the linked PDRs are also required to have  
 959 the same container entity information and associationType values.)

960 The two PDRs on the right and the large single PDR on the left represent exactly the same association  
 961 relationship: the container entity "physical chassis 1" contains two physical power supplies, "power supply  
 962 1" and "power supply 2", and two physical fans, "fan 1" and "fan 2".

963 It is a choice of the implementation whether a single PDR or multiple PDRs are used to represent a  
 964 containment association. Some implementations might want to use multiple records to make it easier to

965 develop and maintain the records. For example, if a new physical entity is added for the chassis, it might  
 966 be more convenient to create a new PDR and link it into the existing containment PDRs for a chassis  
 967 rather than extending an existing containment PDR.



968

969

Figure 11 – Linked Entity Association PDRs

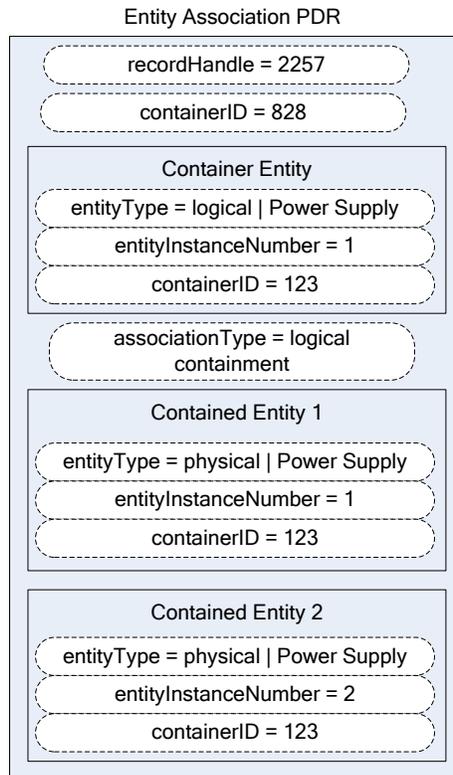
970 **11.4 Logical containment associations**

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

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

978 The example PDR in Figure 12 shows a logical power supply 1 that contains physical power supply 1 and  
 979 a physical power supply 2. In this example, the containerIDs in the enclosed Entity Identification  
 980 Information do not reference the containerID of this overall PDR, but instead reference a container entity  
 981 from a different PDR. This follows from the previous example where containerID 123 corresponds to  
 982 physical chassis 1. The explanation for this is provided in 11.5.

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



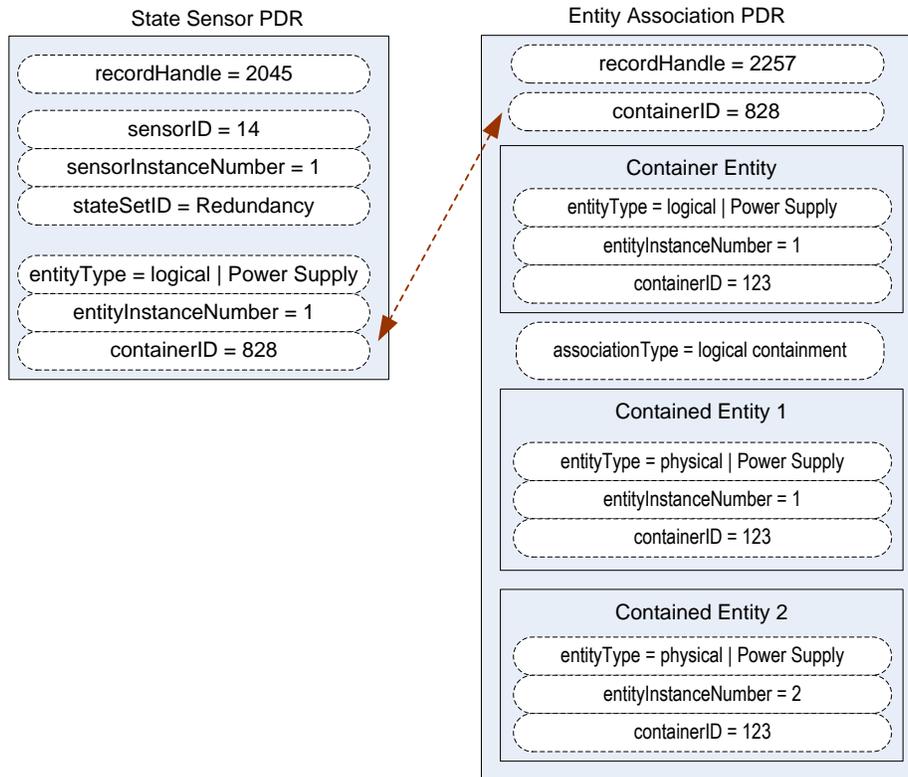
985

986

**Figure 12 – Logical Containment PDR**

987 **11.5 Sensor/effector associations with logical entities**

988 Sensors and effectors can be associated with logical entities in the same way that they can be associated  
 989 with physical entities. Figure 13 shows a state sensor that provides redundancy status and that has a  
 990 sensor-to-entity association to logical power supply 1. Note that containerID 123 follows from the previous  
 991 example where containerID 123 corresponds to physical chassis 1.



992

993

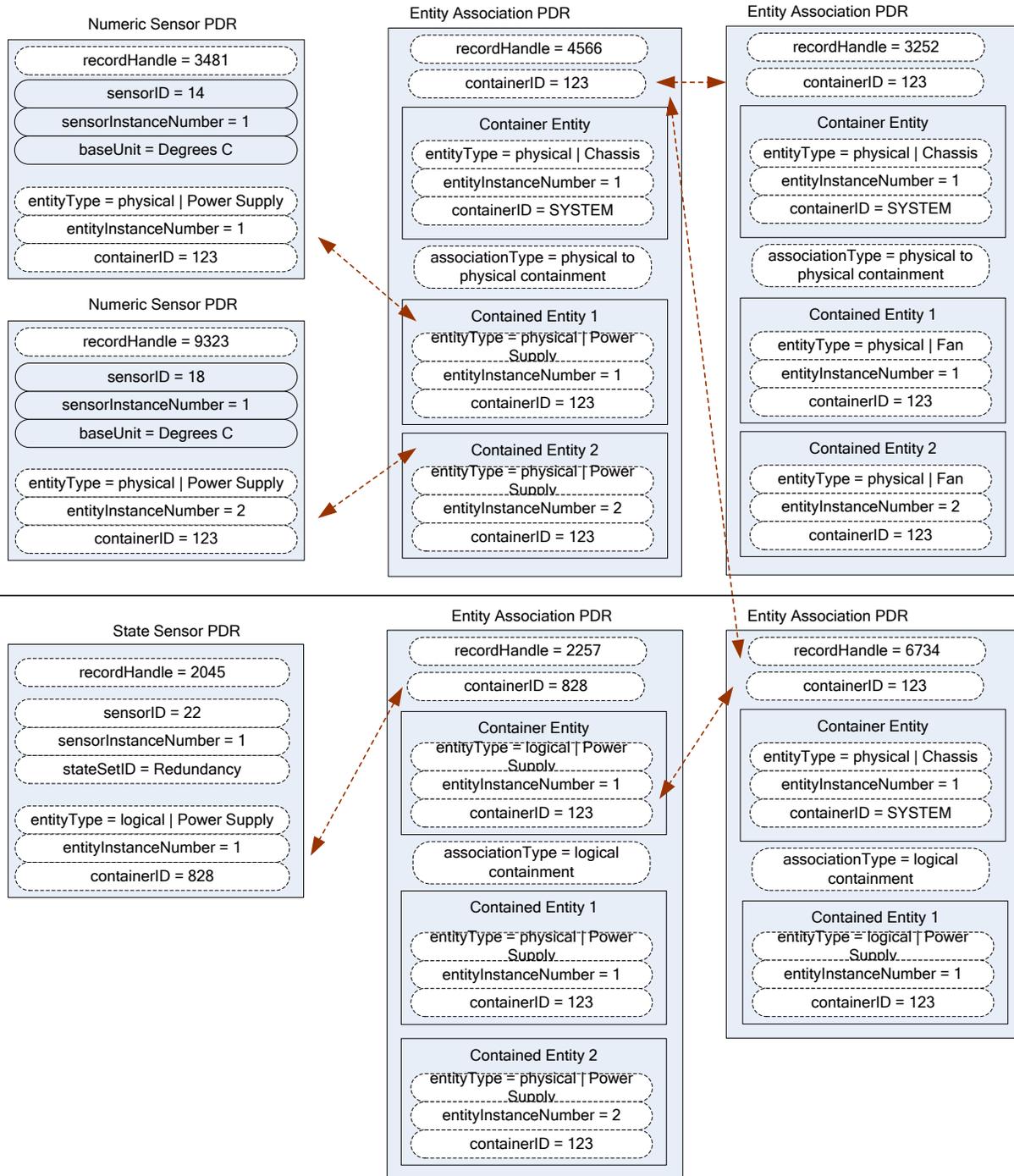
**Figure 13 – Sensor/effector to logical entity association**

994 **11.6 Merged entity associations**

995 Figure 14 presents a merged example that illustrates the different aspects and types of entity  
 996 associations that were introduced in previous subclauses 11.1 through 11.5. The PDRs in the top portion  
 997 of Figure 14 represent sensors and physical-to-physical containment associations. The lower half of  
 998 Figure 14 has PDRs that are related to the sensor and containment associations that define a logical  
 999 power supply. Together, these PDRs model a system that is represented in the block diagram shown in  
 1000 Figure 15.

1001 The Entity Association PDR that defines the contained entities for logical power supply 1 uses 123 as the  
 1002 containerID in the Entity Identification Information for the contained physical power supplies rather than  
 1003 828, the containerID for the logical association, for the following reasons:

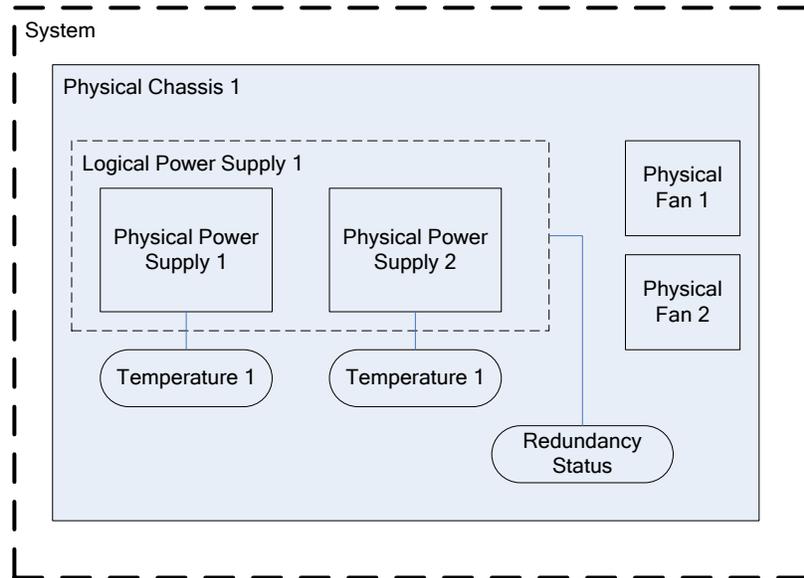
- 1004
- 1005
- An entity that is contained in both physical and logical containment associations should use the containerID that corresponds to a physical containment association.
- 1006
- The Entity Identification Information values for a given entity must be the same for all references to the entity within the PDRs. A given entity cannot be identified using different container IDs in different associations.
- 1007
- 1008



1009

1010

Figure 14 – Merged Entity Association PDR example



1011

1012

**Figure 15 – Block diagram for Merged Entity Association PDR example**

1013 **11.7 Separation of logical and physical associations**

1014 Logical associations may be thought of as something that is layered on top of the physical association  
 1015 hierarchy. The previous example identifies container entity 123 (which corresponds to Physical Chassis  
 1016 1) as the container entity for both physical and logical association PDRs. The types of associations are  
 1017 handled through separate PDRs, which separates the types of associations and helps avoid confusion  
 1018 when a given entity is part of more than one association.

1019 Figure 15 highlights this by showing the physical-to-physical association PDRs in the upper part of the  
 1020 figure and the logical containment PDRs in the lower part.

1021 **11.8 Designing association PDRs for monitoring and control**

1022 Following is one method for creating or designing PDRs for a simple system:

- 1023 1) Identify the physical entities and assign them Entity Identification Information values:
- 1024 a) Identify the topmost physical container entities and give them the containerID for "system".
- 1025 b) Assign each remaining physical entity a different containerID value using whatever
- 1026 approach works best for the implementation. (For example, containerID values could be
- 1027 assigned sequentially starting from 1, or 1000 if it necessary to have a value that is more
- 1028 readily distinguishable as a being a containerID.)
- 1029 2) Create Entity Association PDRs for the physical-to-physical containment associations.
- 1030 3) Create the Sensor PDR, Effector PDR, or other PDRs that are associated with the physical
- 1031 entities, and set the Entity Identification Information based on the containment PDRs that were
- 1032 created earlier.

- 1033 4) Create the PDRs for any logical entities and set the containerID value for the containing entity to  
1034 the containerID for the appropriate physical container entities.
- 1035 5) Create the Sensor PDR, Effector PDR, or other PDRs that reference those logical entities.

## 1036 11.9 Terminus associations

1037 Many PDRs that are related to monitoring and control include a value called the PLDM Terminus Handle.  
1038 This is an opaque value that is used solely within the PDRs in a given repository as a means of identifying  
1039 the records that are associated with a particular terminus. The Terminus ID (TID) is a value that is used  
1040 with PLDM messaging as a way to identify a particular terminus. A PDR called the PLDM Terminus  
1041 Locator PDR is used to bind the PLDM Terminus Handle and the TID for a given terminus.

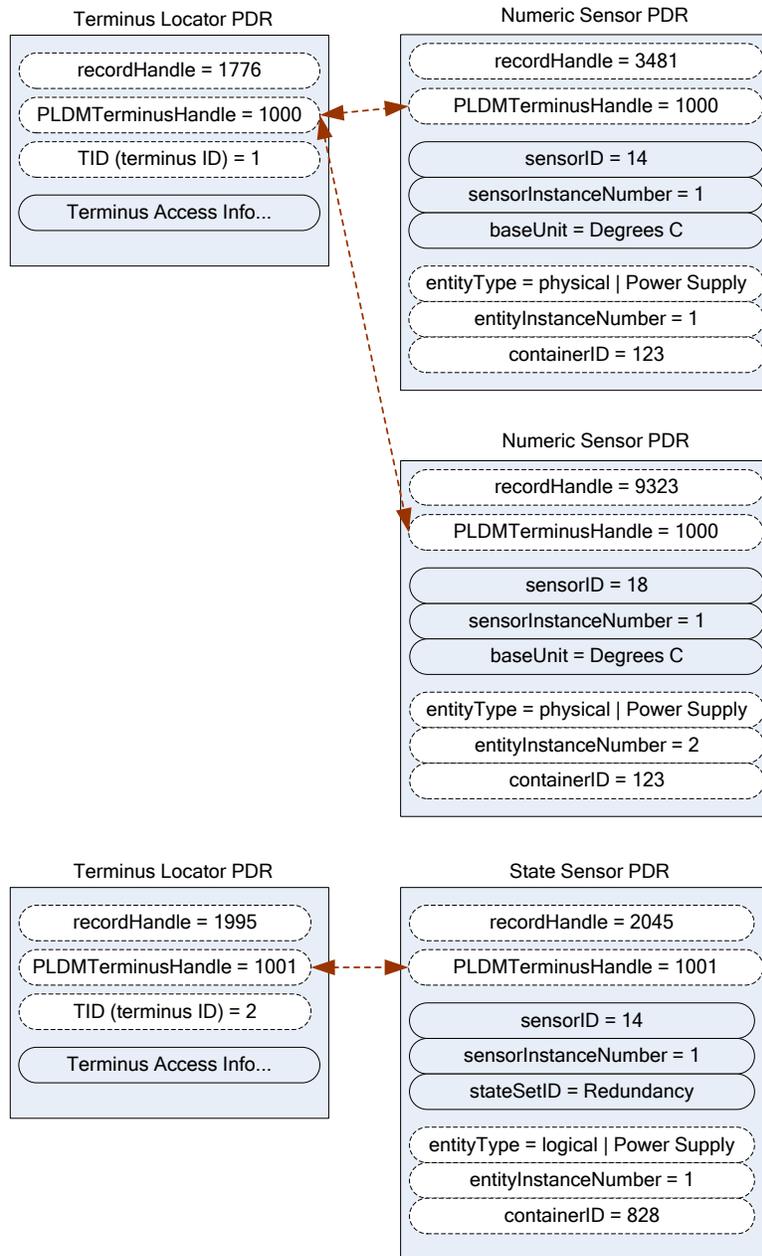
1042 An overview of PLDM Terminus Handles and TIDs is given in 12.1. Figure 16 provides an illustration of  
1043 the relationship of the PLDM Terminus Handle and TID and how they are used within the PDRs.

1044 The association of entities with sensors and effectors is independent of the terminus that provides access  
1045 to the sensor or effector. Sensors and effectors are associated with the entity that is being monitored or  
1046 controlled rather than the entity that is providing the PLDM terminus that is used to access the sensor or  
1047 effector. For example, if a system board entity has a voltage sensor and a temperature sensor, the  
1048 voltage sensor could be provided through one terminus and the temperature sensor through a different  
1049 terminus. Both sensors would be associated with the same system board entity, however.

1050 Because Entity Association PDRs may have content in them that has associations with more than one  
1051 terminus, the PLDM Terminus Handle is used to identify which terminus *provided* the PDR rather than  
1052 which terminus *is associated with* the PDR. For example, this information can be used to identify when  
1053 PDR information has been provided by an add-in card so that the PDRs can be updated if the add-in card  
1054 is removed. In many applications, such as mapping PLDM to CIM, the PLDM Terminus Handle  
1055 information in an Entity Association PDR can be ignored.

1056 Figure 16 also shows how the PLDMTerminusHandle field is used to identify which sensor PDRs are  
1057 accessed through a particular terminus. The example shows two different termini providing sensors for  
1058 the system. The terminus with TID 1 is bound to PLDMTerminusHandle 1000 using the Terminus Locator  
1059 PDR with recordHandle 1776; the terminus with TID 2 is bound to PLDM Terminus Handle 1001 using the  
1060 Terminus Locator PDR with recordHandle 1995.

1061 PLDMTerminusHandle 1000 is associated with the PDRs for two numeric temperature sensors that are  
1062 then associated with physical power supplies 1 and 2. PLDMTerminusHandle 1001 is associated with a  
1063 single redundancy state sensor that is associated with logical power supply 1. Figure 17 shows a block  
1064 diagram of these relationships. Note that while this example shows different termini monitoring different  
1065 entities, different termini can also provide sensors that monitor a common entity. For example, one  
1066 terminus could provide voltage sensors for a processor while another terminus could provide a  
1067 temperature sensor for the same processor.



1068

1069

**Figure 16 – TID and PLDM Terminus Handle associations**

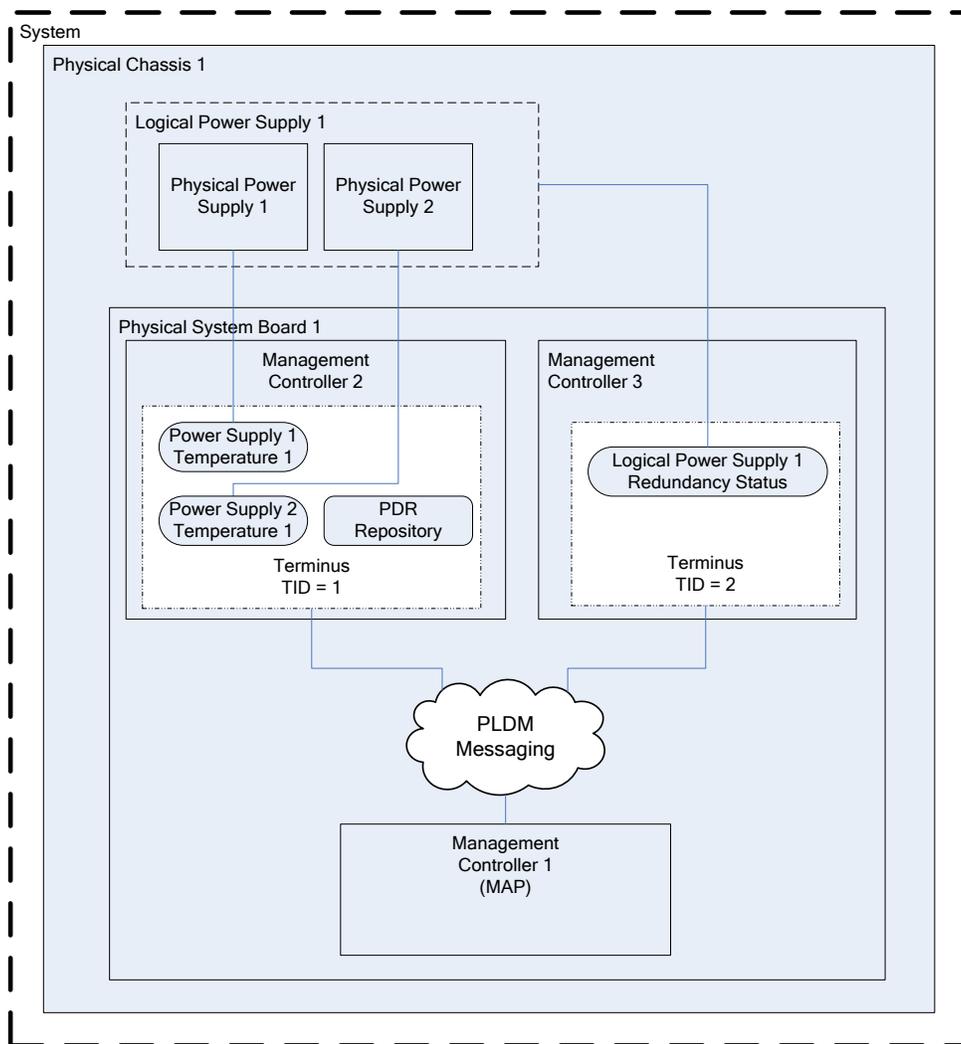
1070 Figure 17 shows a block diagram representation of a hypothetical system that is consistent with the  
 1071 terminus-to-sensor associations shown in Figure 16.

1072 The example contains three management controllers. Management Controller 3 implements a PLDM  
 1073 terminus that includes a PLDM State Sensor that provides the redundancy status of logical power supply  
 1074 1. Management Controller 2 implements a PLDM terminus that supports PLDM access to temperature  
 1075 sensors for physical power supplies 1 and 2. Management Controller 2 also holds the Primary PDR  
 1076 Repository for the system. Management Controller 1 represents a management controller or some other  
 1077 party that is accessing the PLDM subsystem. Management Controller 1 gets its view of the PLDM

1078 subsystem by accessing the PDRs in the Primary PDR Repository provided by Management Controller 2.  
 1079 Although this example shows one terminus per management controller, more than one terminus can be  
 1080 implemented in a management controller.

1081 The PLDM Messaging cloud represents PLDM messaging connectivity between these three controllers.  
 1082 In an actual implementation, this connectivity would be accomplished using a transport protocol and  
 1083 physical medium that supports PLDM messaging, such as MCTP over SMBus/I<sup>2</sup>C.

1084 The example PDRs in Figure 16 are a subset of the PDRs that would be needed to represent the system  
 1085 shown in Figure 17. For example, in addition to the Terminus Locator and Sensor PDRs, Entity  
 1086 Association PDRs would identify that physical chassis 1 contains physical power supplies 1 and 2, logical  
 1087 power supply 1, and a physical system board 1; that system board 1 contains Management Controllers 1,  
 1088 2, and 3; and so on.



1089

1090

Figure 17 – Block diagram of Terminus to Sensor associations

## 1091 11.10 Interrupt associations

1092 Platform interrupts represent logical or physical signals that may be monitored or controlled by PLDM,  
1093 such as NMI, IRQs, software interrupts, and so on. PLDM State Sensors and PLDM State Effecters can  
1094 be used to monitor or control platform interrupts.

### 1095 11.10.1 Interrupt Association PDR

1096 PLDM includes a type of Association PDR called an Interrupt Association PDR that can be used to  
1097 identify the relationship between one or more interrupt source entities and the target entity for a platform  
1098 interrupt. The Interrupt Association PDR also identifies which sensor or effector is associated with the  
1099 source entity. (Because a given target may receive interrupts from multiple sources, the sensor or effector  
1100 is typically associated with the source entity rather than the target entity.)

1101 Two kinds of interrupts can be monitored by a state sensor:

- 1102 • **Received** interrupt associations identify when an interrupt target entity has received an interrupt  
1103 from an interrupt source entity.
- 1104 • **Requested** interrupt associations identify when an interrupt source has issued an interrupt  
1105 request to an interrupt target entity.

1106 Received interrupts and requested interrupts have different state sets. Thus, received and requested  
1107 interrupts are differentiated by the state set that is used with the sensor. Effecters will typically use only  
1108 the state sets for requested interrupts.

### 1109 11.10.2 Interrupt Association example

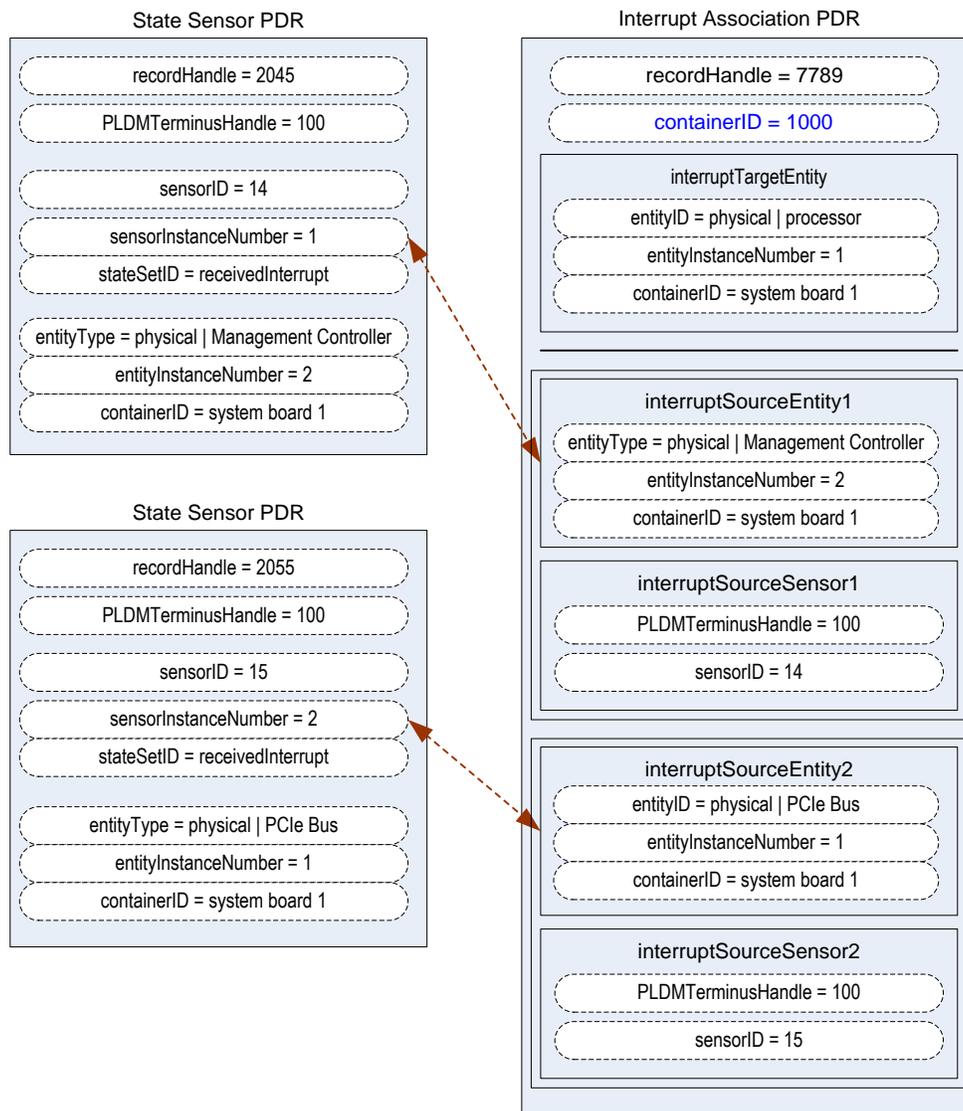
1110 This clause presents an example of using an Interrupt Association PDR. In this example, processor 1 is  
1111 the interrupt target entity that is associated with PCIe Bus 1 and Management Controller 2 as potential  
1112 interrupt source entities. Management Controller 1 provides the implementation of two sensors that report  
1113 whether interrupts have been received from those sources.

1114 For this example, assume that each state sensor detected that an interrupt occurred and subsequently  
1115 generated an event message on that state change. The event message itself indicates only that "Sensor  
1116 14 in TID 2 has entered state x". The PDRs are used to interpret this information as follows:

- 1117 1) The TID that is received in the event message is used to locate the PLDM Terminus Locator  
1118 record for the terminus. From this, the PLDMTerminusHandle is obtained.
- 1119 2) The PLDMTerminusHandle and sensorID value are used to locate the State Sensor PDR for the  
1120 sensor that triggered the event message. This PDR indicates that the stateSetID equals the  
1121 "Interrupt" state set. The state set definition indicates that the value "x" means "received  
1122 interrupt detected".
- 1123 3) The Entity Identification Information in the State Sensor PDR indicates that the interrupt is  
1124 associated with Management Controller 1, which implies that Management Controller 1 is the  
1125 source entity for the interrupt.
- 1126 4) At this point, the combination of the information in the event message and the state sensor PDR  
1127 yields the following interpretation of the event message:
  - 1128 – "Sensor 14 in TID 2 has detected that an interrupt has been received from Management  
1129 Controller 1".
- 1130 5) This information does not identify the target of the interrupt, however. To identify the target, the  
1131 PLDMTerminusHandle and sensorID are used to locate the Interrupt Association PDR that  
1132 identifies the target.

1133 The format of the Interrupt Association PDR in Figure 18 is similar to that of the containment association  
1134 PDRs shown earlier. The main difference is that sensorID information is provided in conjunction with the

1135 Entity Identification Information for the interrupt source entities. This additional information is required  
 1136 because a given source entity may be the source of more than one interrupt. The sensorID information  
 1137 provides the mechanism for differentiating different interrupts from the same interrupt source entity.



1138

1139 **Figure 18 – Received interrupt association example**

1140 **12 PLDM terminus**

1141 A PLDM terminus is the point of communication termination for PLDM messages and the PLDM functions  
 1142 associated with those messages. A terminus must be uniquely identifiable so that PLDM PDRs can  
 1143 associate semantic information with it. Additionally, a terminus must be identifiable when it generates

1144 asynchronous messages, such as event messages. This identification is accomplished through a value  
1145 called the Terminus ID (TID).

## 1146 **12.1 TIDs, PLDM Terminus Handles, and Terminus Locator PDRs**

1147 The TID is primarily used in PLDM messages to identify which terminus generated an asynchronous  
1148 message, such as an event message. The PLDM Terminus Handle is a value that is used within a PDR  
1149 Repository to identify PDRs that are associated with a particular terminus. Thus, the PLDM Terminus  
1150 Handle is defined only within the scope of a particular PDR Repository. A PDR called the Terminus  
1151 Locator PDR is used to associate a TID with a Terminus Handle. The Terminus Locator PDR also  
1152 includes information that describes how the terminus is accessed using PLDM messaging.

## 1153 **12.2 Requirements for unique TIDs**

1154 The assignment of unique TIDs to termini is required in the following situations:

- 1155 • Unique TIDs are required for implementations that use PDRs for describing sensors, effecters,  
1156 and associations within and among termini.
- 1157 • Unique TIDs are required when an implementation exposes a PLDM Event Log in order to  
1158 discriminate events from different termini when reading the log.

## 1159 **12.3 Terminus messaging requirements**

1160 PLDM termini that meet this specification must implement PLDM Request (command) and Response  
1161 messages per [DSP0240](#). Additionally, a Management Controller that implements the Event Receiver  
1162 function must be able to accept and process at least one Event Message request while it is processing  
1163 other (non-Event Message) requests. Similarly, a device that generates Event Messages must be able to  
1164 accept an incoming request while it is waiting for the response for the event message.

1165 It is recommended that a terminus can accept and track requests from multiple requesters if the terminus  
1166 is used in an implementation where it is likely to receive simultaneous requests from multiple parties.

## 1167 **12.4 Terminus Locator PDRs**

1168 The Terminus Locator PDR forms the association between a TID and PLDM Terminus Handle for a  
1169 terminus. The Terminus Locator PDR thus binds a given terminus and the semantic information that is  
1170 provided through the PDRs for the terminus. Figure 19 illustrates the relationship between a TID and  
1171 PLDM Terminus Handle.

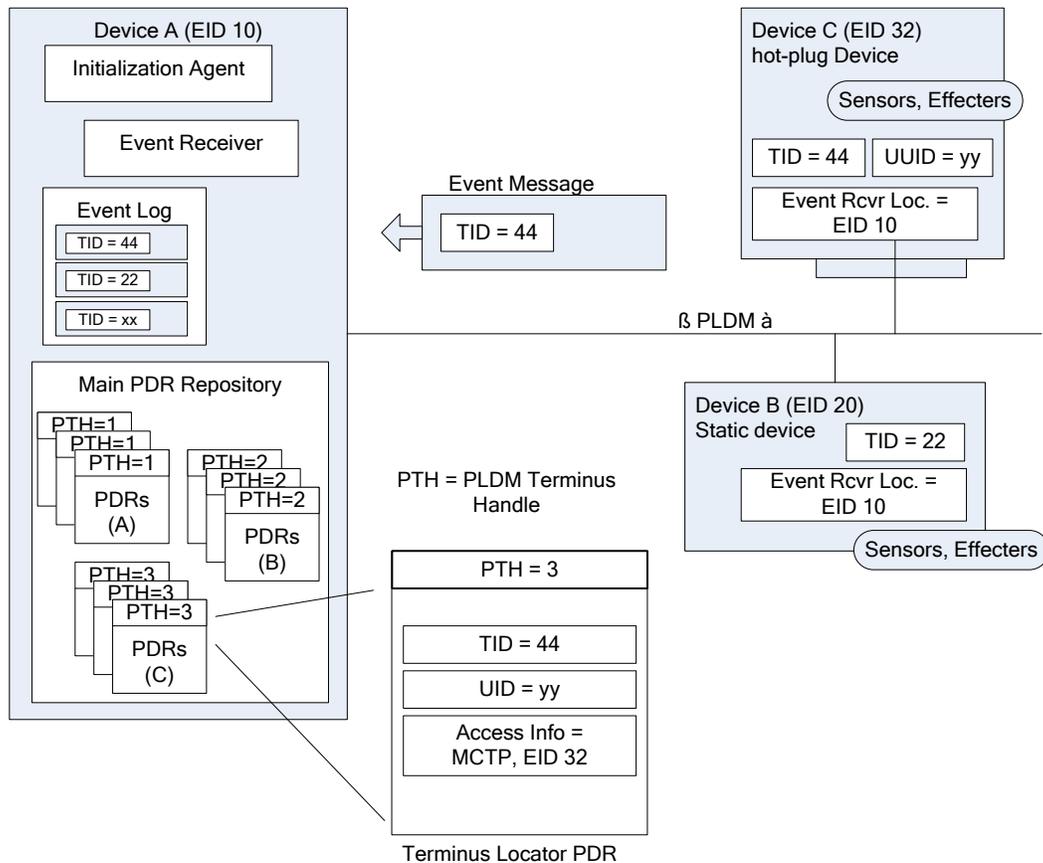
1172 The Terminus Locator PDR also provides additional information about a terminus, such as how it can be  
1173 accessed through PLDM messages (hence the name "Terminus Locator"), and whether the terminus and  
1174 set of PDRs associated with that terminus should be considered present.

1175 If the terminus has a UID or UUID, the Terminus Locator PDR may also hold a copy of the UID/UUID  
1176 value. This value provides an additional mechanism to help verify that the PDRs associated with the  
1177 terminus are correct for the particular terminus instance.

1178 The relationship between the PDRs and PLDM Messaging to and from a given terminus is identified using  
1179 the following data in the Terminus Locator PDR. (This information is expressed using multiple fields within  
1180 the actual record format.)

- 1181 • The PLDM Terminus Handle is used to identify PDRs that are associated to a particular  
1182 terminus. It is used only within the scope of a particular PDR Repository.
- 1183 • The TID identifies a terminus for PLDM messaging, particularly for identifying messages that  
1184 come from a given terminus. A PLDM Terminus Locator PDR associates the TID with the PLDM  
1185 Terminus Handle that is used for accessing the PDRs that are associated with the terminus.

- 1186 • The Terminus Access Info consists of a list of protocols and additional information, such as  
1187 addressing, which enables a party to send PLDM messages to the terminus.



1188

1189 **Figure 19 – Example of TID and PLDM Terminus Handle relationships**

1190 **12.5 Enumerating termini**

1191 A party that accesses the Primary PDR Repository can use the PDRs to enumerate the termini by listing  
1192 and examining the Terminus Locator PDRs.

1193 **12.5.1 General**

1194 To support alternative platform configurations and hot-plug devices, the PDR Repository may have PDRs  
1195 in it for termini that might not be present. This enables the PDR Repository to hold a superset of  
1196 information for the possible termini that might be installed in the system. This helps enable  
1197 implementations that support different configurations of termini using a preconfigured, static set of PDRs.

1198 To support this, the Terminus Locator PDR contains a field that indicates whether the record itself is valid.  
1199 A terminus may also have a state sensor associated with it that reports whether the terminus is present  
1200 and available for use (described in 12.5.3).

1201 The following rules apply to using Terminus Locator PDRs for enumerating termini. When it is stated that  
1202 a terminus should be ignored, it is not an error condition. It means that the status of the terminus is  
1203 unknown and from a PLDM point-of-view should be treated as if it did not exist at all.

1204 • A terminus must have a Terminus Locator PDR that is marked as valid in order to be  
1205 considered present. Only one Terminus Locator PDR is allowed to be valid at a time for a given  
1206 PLDM Terminus Handle within a PDR Repository. It is an error condition if multiple Terminus  
1207 Locator PDRs exist and are simultaneously marked as valid for a given PLDM Terminus  
1208 Handle.

1209 • If the terminus has a sensor associated with it that reports Terminus State, the sensor must  
1210 indicate that the terminus is present. Otherwise, the terminus and its associated PDRs should  
1211 be ignored.

1212 • If the terminus has a sensor associated with it that reports Terminus State and the Terminus  
1213 State information cannot be accessed because the operationalState of the sensor is not  
1214 "enabled", the terminus and its associated PDRs should be ignored.

### 1215 **12.5.2 Unlisted or absent termini**

1216 PDRs for a particular terminus should be ignored under the following conditions:

- 1217 • The PDR does not have an associated Terminus Locator PDR.
- 1218 • The PDR is related to a terminus that has an associated Terminus Locator PDR that is marked  
1219 invalid or is not present based on a presence sensor.

1220 References to termini (for example, PLDM Terminus Handles) should be ignored under the following  
1221 conditions:

- 1222 • The reference does not have an associated Terminus Locator PDR.
- 1223 • The reference is associated with a Terminus Locator PDR that is marked invalid or is not  
1224 present based on a presence sensor.

1225 These conditions do not apply to OEM or vendor-defined PDRs.

### 1226 **12.5.3 Terminus presence using Terminus State Sensors**

1227 In some implementations, termini may need to be added or removed as devices are added to or removed  
1228 from the platform or as platform configurations are changed. This can be handled by updating the validity  
1229 field in the Terminus Locator PDRs or by updating the PDRs to add or remove Terminus Locator PDRs.  
1230 Correspondingly, other PDRs that are associated with the terminus may also be updated, added, or  
1231 removed. Updating PDRs may not be warranted in some implementations, such as when the  
1232 implementation would have otherwise been able to use a static configuration of PDRs.

1233 A more dynamic way of indicating terminus presence is to associate a terminus with a "Terminus State  
1234 Sensor". A Terminus State Sensor is a type of PLDM Composite State Sensor that is associated with a  
1235 logical entity of type "PLDM Terminus" using a sensor to entity association. The sensor returns state set  
1236 enumerations for "Presence status" and "Operational status". A Terminus State Sensor may be  
1237 implemented as a sensor at the terminus itself, or it may be implemented as a sensor under another  
1238 terminus.

## 1239 **13 PLDM events**

1240 PLDM events are primarily related to changes of PLDM sensor states or states that are related to the  
1241 operation of PLDM or the PLDM subsystem itself.

1242 NOTE PLDM events are not the same as CIM indications. There will typically not be a one-to-one correspondence  
 1243 between PLDM events and CIM indications. In some cases, a PLDM event may trigger a MAP to generate  
 1244 indications or entries in a CIM record log, while in other cases a PLDM event may be used solely to update  
 1245 CIM properties to eliminate or reduce polling by the MAP, or to report information about the internal health or  
 1246 operation of the PLDM subsystem that is not exposed through CIM.

1247 **13.1 PLDM Event Messages**

1248 PLDM Event Messages are PLDM monitoring and control messages that are used by a PLDM terminus to  
 1249 asynchronously report PLDM events to a central party called the PLDM Event Receiver.

1250 **13.2 PLDM Event Receiver**

1251 The destination for event messages within PLDM is called the Event Receiver. The Event Receiver  
 1252 function is implemented by a PLDM terminus within the platform management subsystem. Multiple termini  
 1253 can send Event Messages to the Event Receiver function. The SetEventReceiver command is used to  
 1254 give the location of the Event Receiver function to termini that generate event messages.

1255 A PLDM subsystem implementation can have only one PLDM Event Receiver function enabled at a given  
 1256 time. It is expected that typical implementations will always assign the same Event Receiver location.  
 1257 However, the location of the Event Receiver function is allowed to be changed during PLDM subsystem  
 1258 operation. For example, some implementations may do this to support a failover of the Event Receiver  
 1259 function, or to migrate it to a management controller that is hot plugged into the system, and so forth.

1260 **13.3 PLDM Event Logging**

1261 PLDM Event Logging defines an interface through which event messages that have been received at the  
 1262 Event Receiver can be saved in an area of storage called the PLDM Event Log for later retrieval. Event  
 1263 logging includes mechanisms for storing and time-stamping event records, determining characteristics of  
 1264 the log (such as its capacity), and reading and clearing the contents of the log.

1265 Additionally, "virtual" PLDM Event Messages may be internally generated within the terminus that is  
 1266 providing the PLDM Event Log function and directly logged without appearing as PLDM Event Messages  
 1267 on any external interface.

1268 A PLDM subsystem shall contain only one PLDM Event Log function.

1269 Additional information about event logging is provided in clause 23.

1270 **13.4 PLDM Event Log clearing policies**

1271 The PLDM Event Log can use different policies for automatically clearing entries from the log (Table 5).  
 1272 The active policy is configured through the SetPLDMEventLogPolicy command. Refer to the specification  
 1273 of this command for policy support requirements.

1274 **Table 5 – 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 utilize any parameters.
FIFO	When the log is full, the oldest <i>N</i> entries are automatically deleted when the next entry is received.  This policy uses a single parameter, <i>N</i> . <i>N</i> may be a fixed or configurable parameter, depending on the implementation. An implementation can also express <i>N</i> as a percentage of the log ( <i>N</i> Percentage) instead of as an integral number of entries.

Policy	Description
Clear on Age	<p>When the log has filled past a threshold number of entries, <math>M</math>, the age of the first <math>N</math> entries is checked to see if they have been in the log for more than a given age interval. If the <math>M</math>th entry is older than the age interval, the first <math>N</math> entries are automatically cleared from the log. If the log is less than <math>M</math> entries full, entries are retained indefinitely, regardless of their age.</p> <p>This policy uses three parameters: Age, <math>N</math>, and <math>M</math>. The Age interval, the number of automatically cleared entries, <math>N</math>, and the threshold value, <math>M</math>, may be fixed or configurable parameters, depending on the implementation. The policy may also be implemented with <math>N</math> and <math>M</math> given as percentages of the log (<math>M</math>Percentage and <math>N</math>Percentage) instead of an integral number of entries.</p>

1275 **13.5 Oldest and newest log entries**

1276 Unless otherwise specified, when the terms *old*, *older*, *oldest*, *new*, *newer*, and *newest* are used to refer  
 1277 to PLDM Event Log entries, the terms refer to the time that the event was entered into the log rather than  
 1278 the time stamp of the entry. This is because the setting of the log time stamp clock might be changed  
 1279 during system operation, making it possible for temporally newer log entries to have time stamps that  
 1280 refer to an older time than temporally older entries.

1281 **13.6 Event Receiver Location**

1282 The information that is used by a given terminus to send messages to the Event Receiver function (such  
 1283 as addressing) is referred to as the Event Receiver Location information. Event Receiver Location  
 1284 information is transport dependent; for example, for MCTP the information would consist of the EID  
 1285 (MCTP Endpoint ID) of the Event Receiver. Additionally, the Event Receiver Location information may  
 1286 vary on a per-terminus basis, depending on the requirements of the transport and medium. The PLDM  
 1287 Transport binding specifications define how the Event Receiver Location is set for a particular transport  
 1288 and medium.

1289 PLDM supports a SetEventReceiver command that enables the Event Receiver Location information to  
 1290 be delivered to termini that generate event messages. This approach provides the following  
 1291 characteristics:

- 1292 • It eliminates the need to specify a well-known address for the Event Receiver function for each  
 1293 different medium and transport.
- 1294 • It supports assigning the Event Receiver function to a different location, which could be used to  
 1295 – support failover of the Event Receiver function to another device  
 1296 – enable the Event Receiver function to be handled by an alternative device that gets added  
 1297 into the system  
 1298 – support a situation in which the Event Receiver function is on a medium where its address  
 1299 changes during PLDM operation
- 1300 • It provides a mechanism that helps synchronize the generation of event messages with the  
 1301 availability of the Event Receiver function.

1302 **13.7 PLDM Event Log entry formats**

1303 Table 6 shows the general format that is used for all PLDM Event Log entries.

1304

**Table 6 – PLDM Event Log entry format**

Byte	Type	Field
0	enum8	<b>entryType</b> value: { PLDMPlatformEvent, OEMTimestampedEntry, OEMEntry }
1	uint8	<b>entryDataLength</b> The size in bytes of the entryData field.
variable	–	<b>entryData</b> Data for the entry, dependent on the entryType. If entryType = PLDMPlatformEvent, the entryData format is given in Table 7. If entryType = OEMTimestampedEntry, the entryData format is given in Table 8. If entryType = OEMEntry, the entryData format is given in Table 9.

1305 **13.8 PLDM Platform Event Entry Data format**

1306 Table 7 specifies the format used for the entryData field in PLDM Event Log entries that use the  
1307 PLDMPlatformEvent value for the entryType field.

1308

**Table 7 – Platform Event Entry Data format**

Byte	Type	Field
0	sint8	<b>entryTimestampUTCOffset</b> The UTC offset for the log entry time stamp in increments of 1/2 hour special value: 0xFF = unspecified
1:5	uint40	<b>entryTimestampSeconds</b> 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	<b>entryTimestamp100s</b> 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	–	<b>eventData</b> The eventData format is the same as the format for the request parameters of the PlatformEventMessage command (see Table 14).

1309 **13.9 OEM Timestamped Event Entry Data format**

1310 Table 8 specifies the format used for the entryData field in PLDM Event Log entries that use the  
 1311 OEMTimestampedEntry value for the entryType field.

1312 **Table 8 – OEM Timestamped Event Entry Data format**

Byte	Type	Field
0:3	uint32	<b>vendorIANA</b> The IANA Enterprise Number for the vendor that is defining the OEMData. The list of Enterprise Numbers can be found at <a href="http://www.iana.org/protocols/">www.iana.org/protocols/</a> . special value: 0 = unspecified.
4	sint8	<b>entryTimestampUTCOffset</b> The UTC offset for the log entry time stamp in increments of 1/2 hour special value: 0xFF = unspecified
5	uint40	<b>entryTimestampSeconds</b> 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	<b>entryTimestamp100s</b> 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	<b>OEMData</b> 0 to 32 bytes of OEM-specific data that is specified by the vendor identified by vendorIANA

1313 **13.10 OEM Event Entry Data format**

1314 Table 9 specifies the format used for the entryData field in PLDM Event Log entries that use the  
 1315 OEMEntry value for the entryType field. The format is similar to the OEM Timestamped Event Entry Data  
 1316 format (shown in Table 8), except that it does not include PLDM-defined time stamp fields.

1317 **Table 9 – OEM Event Entry Data format**

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

1318 **14 Discovery Agent**

1319 The Discovery Agent function is responsible for discovering termini, assigning them unique TID values,  
 1320 and assigning them the address of the Event Receiver function.

1321 If the implementation is maintaining a Primary PDR Repository, the Discovery Agent may also be required  
1322 to automatically create or update PDRs to support devices such as hot-plug devices that may be  
1323 dynamically added or removed from the system. This includes the following actions:

- 1324       • creating records such as Terminus Locator PDRs
- 1325       • extracting Device PDR information and merging it into the Primary PDR Repository
- 1326       • updating associating records to link Device PDR information into the overall context of the  
1327 platform management subsystem

1328 Any OEM PDRs in the Device PDR information that are identified to be copied to the Primary PDR  
1329 Repository are also added to the Primary PDR Repository by the Discovery Agent.

### 1330 **14.1 Assignment of TIDs and Event Receiver location**

1331 Following are the support requirements for assignment of TIDs and the launching of the Initialization  
1332 Agent by a Discovery Agent within a PLDM implementation:

- 1333       • All termini must support the SetTID command.
- 1334       • All termini that generate PLDM Event Messages shall support the SetEventReceiver command.  
1335 Termini that do not generate PLDM Event Messages are not required to support the  
1336 SetEventReceiver command.
- 1337       • The Discovery Agent function is responsible for discovering termini and assigning them unique  
1338 TID values. (A default TID setting may be pre-configured for a PLDM terminus if the terminus is  
1339 statically configured into the platform. This setting must be able to be overridden using the  
1340 SetTID command.)
- 1341       • The Initialization Agent function is responsible for initializing PLDM sensors and effecters and  
1342 setting Event Receiver location information into the termini. (A default Event Receiver setting  
1343 may be pre-configured for a PLDM terminus if the terminus is statically configured into the  
1344 platform. This setting must be able to be overridden using the SetEventReceiver command.)  
1345 The Initialization Agent function is described in more detail in clause 15.
- 1346       • When PDRs are used, the Initialization Agent is also responsible for maintaining corresponding  
1347 Terminus Locator PDR information.
- 1348       • A terminus must have its Event Receiver information set before it can begin to issue PLDM  
1349 Event Messages.
- 1350       • A terminus that has standby power should retain its TID and Event Receiver settings. When the  
1351 terminus comes back online, it can use that information for event messaging without requiring  
1352 Event Receiver re-initialization.
- 1353       • A terminus should retain its TID and Event Receiver settings during a given PLDM subsystem  
1354 operation.
- 1355       • Termini that are to be rediscovered (that is, termini that are not statically configured into the  
1356 system and may lose PLDM communication temporarily, which might occur in different platform  
1357 power states) must have a separate unique and persistent ID that can be associated with the  
1358 terminus. For example, if a terminus is hot-plug, it should have a universally unique ID (UUID).
- 1359       • TIDs are not required to persist or remain constant across PLDM subsystem restarts, unless the  
1360 system is using PDRs or exposes a PLDM Event Log. In such cases, TIDs must be persistently  
1361 stored by the termini or reassigned to the same value by the Discovery Agent function.
- 1362       • A MAP or other entity that is accessing a PLDM subsystem should not cache TIDs because  
1363 TIDs might change if the PLDM subsystem is reset or reinitialized.

- 1364
- 1365
- Termini on hot-plug cards must have a UUID or be associated with a terminus on the same card that has a UUID.
- 1366
- 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.
- 1367
- 1368

## 1369 **14.2 UUIDs for devices in hot-plug or add-in card applications**

1370 If the device is intended to be used on an add-in or hot-plug card, it may be required to support a  
1371 universally unique ID (UUID) depending on higher-level system requirements or initiatives. In general,  
1372 add-in cards that plug into standardized I/O connections and are used in multiple vendor systems, such  
1373 as PCIe add-in cards, are required to use UUIDs so that multiple instances of the same card can be  
1374 detected.

## 1375 **14.3 UID implementation**

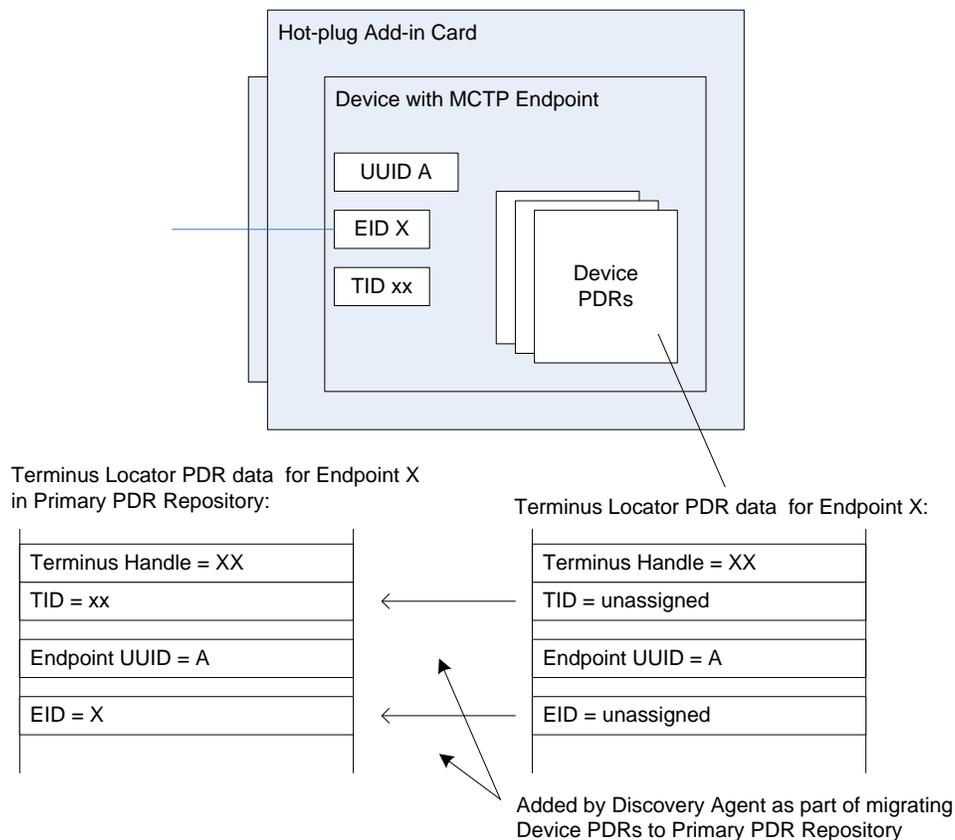
1376 If a terminus is required to have a unique ID (UID), how the UID is implemented depends on the  
1377 component and how the device manufacturer intends the device to be used in a system. For example, it  
1378 is the device manufacturer's choice whether the entire UID must be configured by the system integrator  
1379 after purchasing the device, or a number of pre-configured UIDs in the device are selectable by a pin or  
1380 non-volatile configuration selection, or the UID is permanently embedded in the device. Typically, each  
1381 device will have fuses, PROM, EPROM/EEPROM, or some other non-volatile mechanism for holding the  
1382 unique ID that is configured either during device manufacture or when the device is integrated into a  
1383 system.

## 1384 **14.4 More than one terminus in a device**

1385 The Terminus Locator PDR contains a containerEntity field that can be used to identify the entity that  
1386 contains the terminus. This field provides the mechanism to identify when multiple termini are within the  
1387 same device or are located within the same entity.

## 1388 **14.5 Examples of PDR and UUID use with add-in cards**

1389 Figure 20 and Figure 21 present examples of how Device PDRs, UUIDs, and Terminus Locator PDRs  
1390 work together to identify PLDM termini on add-in cards, such as hot-plug add-in cards, that may be  
1391 dynamically inserted or removed during PLDM subsystem operation. Both examples illustrate MCTP-  
1392 based implementations. However, the approach may be extrapolated to other transport types.



1393

1394

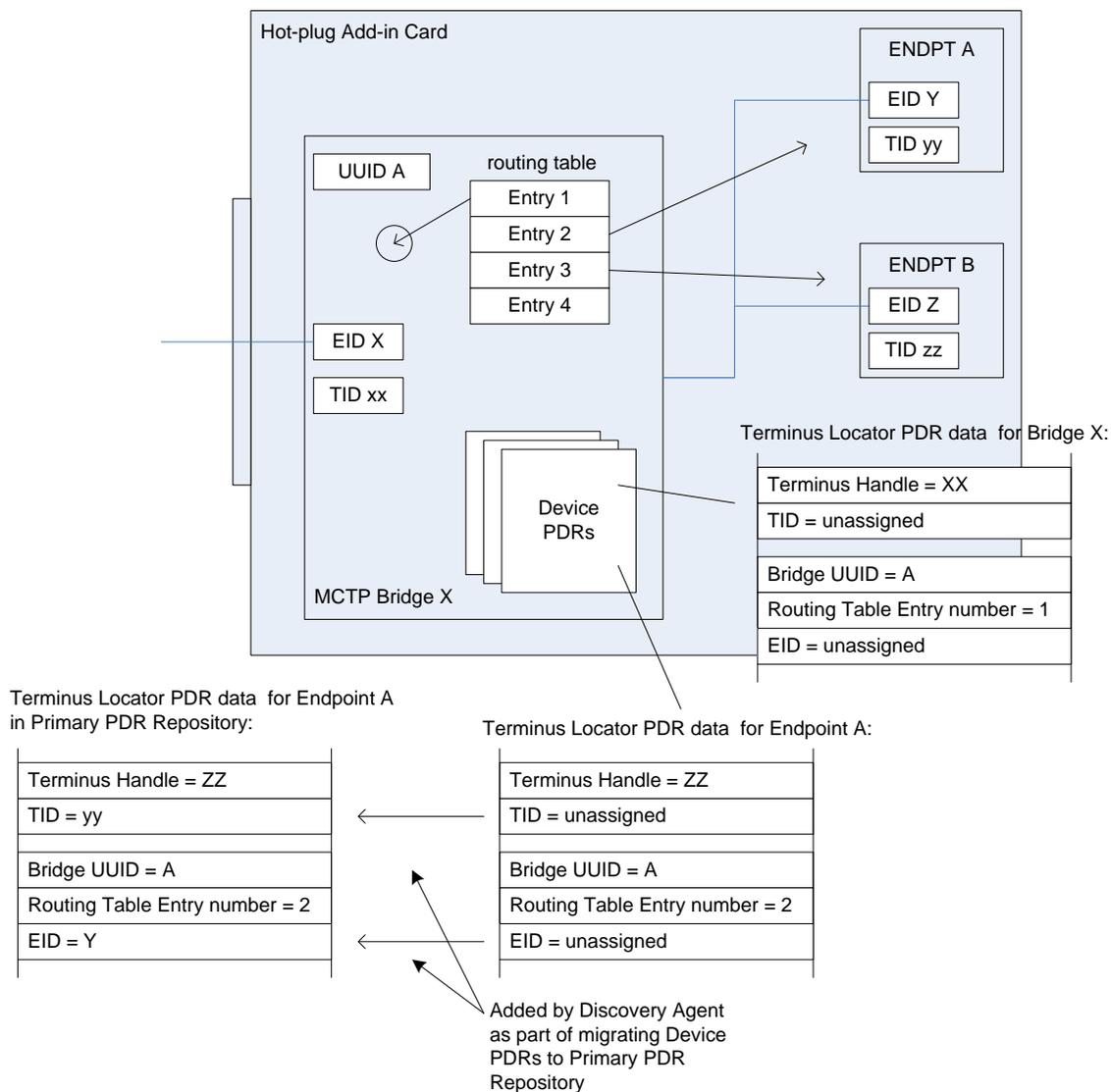
**Figure 20 – Hot-plug add-in card with single PLDM terminus**

1395 Figure 20 shows an add-in card that has a single PLDM terminus that is accessed through a single MCTP  
 1396 endpoint. The terminus is persistently and uniquely identified within the PLDM subsystem by a UUID that  
 1397 is associated with the endpoint and the terminus. This UUID is recorded in a partially filled-in Terminus  
 1398 Locator PDR that is part of the Device PDRs that are provided by the add-in card. The UUID can also be  
 1399 read by issuing a GetTerminusUID command to the terminus. The Device PDRs also report the presence  
 1400 of and semantic information about sensors, effectors, and other functions on the add-in card.

1401 The Terminus Locator PDR from the Device PDRs returns "unassigned" values for the Endpoint ID (EID)  
 1402 and Terminus ID (TID) fields because those values are unavailable before the card has been discovered  
 1403 and initialized by MCTP and the PLDM Discovery Agent within the PLDM subsystem. It also eliminates  
 1404 the need for the terminus to update those Device PDRs whenever TID or EID values are assigned or  
 1405 changed. The Discovery Agent sets the TID for the terminus and adds the EID and TID values to the  
 1406 Terminus Locator Record PDRs when they are integrated into the Primary PDR Repository. The  
 1407 Discovery Agent then synthesizes other PDRs as necessary to link the add-in card into the overall  
 1408 semantic information of the PLDM subsystem. For example, the Discovery Agent may create association  
 1409 PDRs that associate the add-in card with a particular bus and connector within the system.

1410 The Discovery Agent is also responsible for keeping those records up-to-date if EID assignments change  
 1411 during PLDM subsystem operation and for deleting or invalidating the PDRs that are associated with the  
 1412 card and its termini if it detects that the card has been removed.

1413 Figure 21 shows an add-in card that has several MCTP endpoints, each with its own PLDM terminus.  
 1414 One terminus is within an MCTP Bridge device that provides the Device PDRs for all the termini on the  
 1415 card. Additionally, the MCTP Bridge provides a UUID that identifies the overall card for MCTP. All MCTP  
 1416 endpoints are defined relative to MCTP Bridge function based on the position of their routing information  
 1417 in the routing table.



1418

1419 **Figure 21 – Hot-plug add-in card with multiple PLDM termini**

1420 In Figure 21, the MCTP Bridge itself is associated with the first routing table entry, Endpoint A is  
 1421 associated with the second entry, and Endpoint B is associated with the third entry. The Device PDRs  
 1422 hold Terminus Locator PDRs for each terminus that is on the add-in card. These PDRs uniquely identify  
 1423 each terminus using two pieces of information: the UUID of the MCTP Bridge and the position of a routing  
 1424 table entry that is associated with the terminus. The routing table entry positions must not change during

1425 PLDM subsystem operation. This approach eliminates the need for Endpoints A and B to have their own  
1426 support for UUIDs.

## 1427 **15 Initialization Agent**

1428 This clause describes the role and operation of the Initialization Agent function in a PLDM subsystem that  
1429 uses PDRs.

### 1430 **15.1 General**

1431 PLDM sensors are not required to completely self-initialize and enable themselves upon PLDM  
1432 subsystem startup or upon power state changes of the device that is hosting the sensor. Thus, low-cost  
1433 devices are not required to have non-volatile configuration resources. Additionally, the mechanism  
1434 provides options for overriding default configurations of sensors and event generation.

1435 The Initialization Agent is a function that initializes message generation and sensor configuration as  
1436 described by Sensor Initialization PDRs. The Initialization Agent function normally runs whenever the  
1437 platform management subsystem is first powered up, upon system Hard and Soft Resets, and on certain  
1438 other transitions. Fields in the Sensor Initialization PDRs indicate the system transitions on which a given  
1439 sensor is initialized.

1440 The Initialization Agent is also responsible for setting the Event Receiver Location information and  
1441 enabling event message generation.

1442 The Sensor Initialization PDRs hold information that describes the default threshold values, states, and  
1443 event generation settings for sensors that are initialized by the Initialization Agent function. Sensor  
1444 Initialization PDRs are required only for sensors that are initialized by the Initialization Agent. Sensors that  
1445 are self-initializing or are initialized through some mechanism that is outside the PLDM specifications do  
1446 not need Sensor Initialization PDRs.

1447 The Initialization Agent function thus eliminates the need for all sensors to retain their own non-volatile  
1448 storage for their default settings, and also provides a mechanism to retrigger any events that may have  
1449 been transmitted before the Event Receiver function was ready to accept them.

1450 Only one Initialization Agent function is supported within a given PLDM subsystem. The Initialization  
1451 Agent shall be implemented behind the same terminus that provides the Primary PDR Repository for the  
1452 PLDM subsystem.

### 1453 **15.2 PLDM and power state interaction**

1454 The Initialization Agent may need to re-initialize certain sensors or termini as the result of a change of  
1455 system power state. An implementation should avoid requiring the Initialization Agent to execute because  
1456 of low-latency power state transitions, such as transitions between ACPI S0 and S1, or S1 and S2 states.  
1457 The implementation should instead ensure that termini retain their settings across low-latency power state  
1458 transitions.

1459 The Sensor Initialization PDRs include a field that tells the Initialization Agent upon which system  
1460 transitions a given sensor should be initialized.

### 1461 **15.3 RunInitAgent command**

1462 PLDM does not specify a particular mechanism for an implementation to use to detect when to run the  
1463 Initialization Agent function. For example, it does not specify how a management controller would detect a  
1464 system hard reset or power-up transition. In some implementations, it will be useful to have another  
1465 management controller, system firmware, or another entity decide that the Initialization Agent should run.  
1466 For example, system firmware may decide that the Initialization Agent should be run after a BIOS update.

1467 To enable this, PLDM defines a RunInitAgent command that can be used to launch the Initialization Agent  
 1468 “on demand.” The command includes a parameter that can select a subset of Sensor Initialization PDRs  
 1469 to be used.

1470 **15.4 Recommended Initialization Agent steps**

1471 The following presents an outline of the steps for an Initialization Agent in a system implementation that  
 1472 includes Initialization PDRs.

- 1473 1) Stop the Event Receiver function from accepting events received from any interface but the system  
 1474 (host) interface.
- 1475 2) Scan the PDR Repository for Terminus Locator PDRs. Collect a list of valid termini that require  
 1476 initialization. (A field in the Terminus Locator PDR indicates whether any sensors/effecters in the  
 1477 terminus require initialization, and, if so, whether event messaging should be enabled after the  
 1478 controller has been initialized.)
- 1479 3) For each terminus in the list, perform the following actions:
  - 1480 a) Turn off Event Generation by using the SetEventReceiver command. If a terminus does not  
 1481 respond to the SetEventReceiver command, take that terminus off the list.
  - 1482 b) Use the GetTID command to determine whether the terminus has a TID. If so, leave that value  
 1483 unchanged unless it is already assigned to another terminus. If not, use the SetTID command to  
 1484 assign a TID to the terminus.
  - 1485 c) Scan the PDR Repository for Initialization PDRs (for example, numeric sensor initialization  
 1486 PDRs or state sensor initialization PDRs) that are associated for the terminus. For each PDR  
 1487 that is found, perform the following actions:
    - 1488 – Set the sensor type, sensor thresholds, and hysteresis as directed by the PDR using the  
 1489 SetSensorThresholds and SetSensorHysteresis commands.
    - 1490 – Use the appropriate enabling command (for example, SetNumericSensor Enables if the  
 1491 sensor is a numeric sensor) to enable scanning and event generation per the PDR.
- 1492 4) Enable the Event Receiver function to accept event messages.
- 1493 5) For each terminus with a Terminus Locator PDR, enable event message generation using the  
 1494 SetEventReceiver command or leave it disabled (A field in the Management Controller Device  
 1495 Locator record indicates whether event messaging should be enabled after the controller has been  
 1496 initialized.)

1497 **16 Terminus and event commands**

1498 This clause describes the commands that are used by PLDM termini that implement PLDM monitoring  
 1499 and control as defined in this specification. The command numbers for the PLDM messages are given in  
 1500 clause 30.

1501 If a PLDM terminus is implemented to provide access to any of the capabilities of this specification, the  
 1502 Mandatory/Conditional (M/C) requirements shown in Table 10 apply.

1503 **Table 10 – Terminus commands**

Command	M/C	Reference
SetTID (see <a href="#">DSP0240</a> )	M	See 16.1.
GetTID (see <a href="#">DSP0240</a> )	M	See 16.2.
GetTerminusUID	C <sup>[1]</sup>	See 16.3.

Command	M/C	Reference
SetEventReceiver	C <sup>[2][3]</sup>	See 16.4.
GetEventReceiver	C <sup>[2]</sup>	See 16.5.
PlatformEventMessage	C <sup>[2][4]</sup>	See 16.6.

- 1504 <sup>[1]</sup> See 16.3.
- 1505 <sup>[2]</sup> Mandatory for termini that generate PLDM Event Messages.
- 1506 <sup>[3]</sup> Sending the SetEventReceiver command is Mandatory for termini that implement the
- 1507 Initialization Agent function.
- 1508 <sup>[4]</sup> Accepting the PlatformEventMessage is Mandatory for termini that implement the Event
- 1509 Receiver function.

1510 **16.1 SetTID command**

1511 The SetTID command is used to set the TID for a PLDM terminus. This command is typically used by the  
 1512 PLDM Discovery Agent function. This command is defined in [DSP0240](#).

1513 **16.2 GetTID command**

1514 The GetTID command is used to retrieve the present TID setting for a PLDM terminus. This command is  
 1515 defined in [DSP0240](#).

1516 **16.3 GetTerminusUID command**

1517 The GetTerminusUID command is used to obtain a unique ID for the terminus when it is necessary to  
 1518 differentiate between different instances of identical devices that hold the terminus (such as two otherwise  
 1519 identical add-in cards), or when it is necessary to track a particular terminus that may be “relocated,” such  
 1520 as a terminus on an add-in card that is moved from one slot to another.

1521 The GetTerminusUID command shall be supported by a terminus when the terminus is on a hot-  
 1522 pluggable or other add-in card where the platform management subsystem implementation is expected to  
 1523 discover and automatically adopt PLDM capabilities in the terminus (such as sensors) without requiring  
 1524 separate configuration steps to be taken outside of PLDM. See 14.3 and 14.2 for more information.

1525 If more than one terminus is on the same card, only the terminus that provides PDRs for the add-in card  
 1526 is required to support the GetTerminusUID command. Table 11 describes the format of the command.

1527 **Table 11 – GetTerminusUID command format**

Type	Request data
–	none
Type	Response data
enum8	<b>completionCode</b> value: { PLDM_BASE_CODES }
UUID	<b>UUIDValue</b>

1528 **16.4 SetEventReceiver command**

1529 The SetEventReceiver command is used to set the address of the Event Receiver into a terminus that  
 1530 generates event messages. It is also used to globally enable or disable whether event messages are  
 1531 generated from the terminus. Table 12 describes the format of the command.

1532 **Table 12 – SetEventReceiver command format**

Type	Request data
enum8	<p><b>eventMessageGlobalEnable</b></p> <p>This value is used to enable or disable event message generation from the terminus.</p> <p>value: {</p> <p style="padding-left: 40px;">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.</p> <p style="padding-left: 40px;">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.</p> <p style="padding-left: 40px;">// Globally enabling event generation causes all sensors and effecters within the terminus to                      // reassess their event state. The sensors and effecters will generate event messages if                      // their present state does not match their default initialization state.</p> <p style="padding-left: 40px;">}</p>
enum8	<p><b>transportProtocolType</b></p> <p>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 <a href="#">DSP0245</a>. 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.</p>
varies	<p><b>eventReceiverAddressInfo</b></p> <p>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.</p> <p>The format of this field is defined in the PLDM-to-Transport binding specification identified by the transportProtocolType field.</p> <p>If the transportProtocolType value from <a href="#">DSP0245</a> 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.</p>
Type	Response data
enum8	<p><b>completionCode</b></p> <p>value: { PLDM_BASE_CODES, INVALID_PROTOCOL_TYPE=0x80 }</p>

1533 **16.5 GetEventReceiver command**

1534 The GetEventReceiver command is used to verify the values that were set into an Event Generator using  
 1535 the SetEventReceiver command. Table 13 describes the format of the command.

1536 **Table 13 – GetEventReceiver command format**

Type	Request data
–	none
Type	Response data
enum8	<b>completionCode</b> value: { PLDM_BASE_CODES }
enum8	<b>transportProtocolType</b> This value indicates the transportProtocolType that the terminus uses for its eventReceiverAddress and the format of the eventReceiverAddress field. This value is defined in <a href="#">DSP0245</a> .
varies	<b>eventReceiverAddress</b> 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 <a href="#">DSP0245</a> 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.

1537 **16.6 PlatformEventMessage command**

1538 PLDM Event Messages are sent as PLDM request messages to the Event Receiver using the  
 1539 PlatformEventMessage command. Because PLDM requests have associated responses, this approach  
 1540 provides a positive acknowledgement that the event message was received. Table 14 describes the  
 1541 format of the command.

1542 **Table 14 – PlatformEventMessage command format**

Type	Request data
uint8	<b>formatVersion</b> Version of the event format (the format and definition of the following bytes): 0x01 for this format.
uint8	<b>TID</b> Terminus ID for the terminus that originated the event message
enum8	<b>eventClass</b> value: { sensorEvent, // Events that are issued for events that are related to PLDM numeric and // state sensors. See Table 15 for the eventData format for this eventClass. effecterEvent, // See Table 16 for the eventData format for this eventClass. }
var	<b>eventData</b> Event data based on the eventClass
Type	Response data
enum8	<b>completionCode</b> value: { PLDM_BASE_CODES, UNSUPPORTED_EVENT_FORMAT_VERSION = 0x81 }
enum8	<b>status</b> 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. }

1543 **16.7 eventData format for sensorEvent**

1544 Table 15 defines the format of the eventData field in PLDM Event Messages for the sensorEvent class.  
 1545 This field includes event data for PLDM state sensor and numeric sensor events, and for events related to  
 1546 changes of the sensor's operational state.

1547 **Table 15 – sensorEvent class eventData format**

Type	Request data
uint16	<p><b>sensorID</b></p> <p>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.</p>
enum8	<p><b>sensorEventClass</b></p> <p>value: {</p> <p style="padding-left: 40px;">sensorOpState, // Events from a PLDM state or numeric sensor that are related to                      // changes of the sensor's operational state</p> <p style="padding-left: 40px;">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</p> <p style="padding-left: 40px;">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.</p> <p>}</p>
<i>For sensorEventClass = stateSensorState</i>	
uint8	<p><b>sensorOffset</b></p> <p>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</p> <p>value: 0x00 to 0x07</p>
enum8	<p><b>eventState</b></p> <p>The event state value from the state change that triggered the event message.                      See Table 31 for the definition of eventState.</p>
enum8	<p><b>previousEventState</b></p> <p>The event state value for the state from which the present event state was entered.                      See Table 31 for the definition of eventState.</p> <p>special value: This value shall be set to the same value as eventState if the previous event state 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.</p>
<i>For sensorEventClass = numericSensorState</i>	
enum8	<p><b>eventState</b></p> <p>The eventState value from the state change that triggered the event message.                      See Table 20 for the enumeration values of eventState.</p>

Type	Request data
enum8	<p><b>previousEventState</b></p> <p>The eventState value for the state from which the present state was entered.</p> <p>See Table 20 for the enumeration values of eventState.</p> <p>special value: This value shall be set to the same value as eventState if the previous event state 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).</p>
enum8	<p><b>sensorDataSize</b></p> <p>The bit width and format of reading and threshold values that the sensor returns</p> <p>value: { uint8, sint8, uint16, sint16, uint32, sint32 }</p>
uint8   sint8   uint 16   sint16   sint32   uint32	<p><b>presentReading</b></p> <p>The present value indicated by the sensor. The sensorDataSize field returns an enumeration that indicates the number of bits used to return the value.</p>
<i>For sensorEventClass = sensorOpState</i>	
enum8	<p><b>presentOpState</b></p> <p>The sensorOperationalState value from the state change that triggered the event message.</p> <p>See Table 20 for the enumeration values of sensorOperationalState.</p>
enum8	<p><b>previousOpState</b></p> <p>The sensorOperationalState value for the state from which the present state was entered.</p> <p>See Table 20 for the enumeration values of sensorOperationalState.</p> <p>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.</p>

1548 **16.8 eventData format for effectorEvent**

1549 Table 16 defines the format of the eventData field in PLDM Event Messages for the effectorEvent class.  
 1550 This field supports events for changes of the effector's operational state.

1551 **Table 16 – effectorEvent class eventData format**

Type	Request data
uint16	<p><b>effectorID</b></p> <p>The effectorID is the value that is used in PDRs and PLDM effector access commands to identify and access a particular effector within a terminus.</p>
enum8	<p><b>effectorEventClass</b></p> <p>value: {</p> <p style="padding-left: 40px;">effectorOpState // Events from a PLDM state or numeric effector that are related to</p> <p style="padding-left: 40px;">// changes of the effector's operational state</p> <p>}</p>

Type	Request data
<i>For effecterEventClass = effecterOpState</i>	
enum8	<p><b>presentOpState</b></p> <p>The effecterOperationalState value from the state change that triggered the event message.</p>
enum8	<p><b>previousOpState</b></p> <p>The effecterOperationalState value for the state from which the present state was entered.</p> <p>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.</p>

## 1552 17 PLDM Numeric Sensors

1553 This clause provides information that describes the characteristics and operation of PLDM Numeric  
1554 Sensors.

### 1555 17.1 Sensor readings, data sizes

1556 PLDM Numeric Sensors can return a present reading value. The value is returned as a binary integer.  
1557 The size of this integer and whether it is signed can vary on a per-sensor basis. The PLDM  
1558 GetSensorReading command includes a parameter in its response that indicates the format used for  
1559 returning the reading. The same format is used for any thresholds and hysteresis values that are used for  
1560 request or response parameters. Additionally, the data size is supported in PDR information for the  
1561 sensor.

### 1562 17.2 Units and reading conversion

1563 The sensor commands do not intrinsically identify what type of unit, such as volts, amps, or RPM, is used  
1564 for the sensor's present reading value. Additionally, the value may require scaling to convert the value to  
1565 normalized units, such as millivolts (mV), nanoseconds, and so on.

1566 For example, microcontrollers commonly incorporate an 8-bit analog-to-digital (A/D) converter. If the  
1567 converter is monitoring a signal where the 0x00 value of the conversion corresponds to 0 volts and a  
1568 0xFF reading corresponds to 4.00 volts, each count of the converter corresponds to a value of  $4.0/255 \approx$   
1569  $15.686274$  mV per count. Converting a particular reading from counts into volts requires multiplying the  
1570 reading by a conversion factor. A reasonable guideline is that the conversion factor should be accurate to  
1571 at least 4 times the resolution of the converter. In this case, the resolution of the converter is 1 part in 255,  
1572 which would require the accuracy of the conversion factor to be to better than 1 part in 1020, which  
1573 rounds up to four significant digits, or 15.69 mV per count.

1574 To avoid the need for a floating point format for sensor readings and the need for multibyte multiplications  
1575 and divisions in simple devices, PLDM readings are returned as "raw" integers that are converted to  
1576 normalized units by the consumer of the reading data by using a specified conversion formula and  
1577 sensor-specific conversion factors. The consumer of the PLDM sensor reading data will be a device  
1578 serving a role such as a MAP that has more resources for doing mathematical operations. This approach  
1579 avoids burdening simple devices with the conversion task.

1580 The conversion formula is specified in 27.7. The conversion factors must be provided by the vendor or  
1581 designer of the particular sensor implementation. The PDR for a numeric sensor supports returning  
1582 conversion factors and the type of units (volts, amps, and so on) used for a particular numeric sensor.

### 1583 **17.3 Reading-only or threshold-based numeric sensors**

1584 A particular instance of a PLDM Numeric Sensor can return just a numeric reading or a numeric reading  
1585 *and* a threshold-based status. These sensors are referred to as "reading-only" or "threshold-based"  
1586 numeric sensors.

### 1587 **17.4 Readable and settable thresholds**

1588 A given instance of a PLDM Numeric Sensor may have thresholds that are readable through the  
1589 GetSensorThresholds command or that are settable through the SetSensorThresholds command. The  
1590 PDR information can indicate whether a particular numeric sensor uses thresholds and, if so, which  
1591 thresholds are supported and whether they are settable.

### 1592 **17.5 Update / polling intervals and states updates**

1593 A sensor may periodically collect internal readings and status (that is, it may poll for updates) and  
1594 respond to a GetSensorReading request with the last collected values, or it may collect the values "on  
1595 demand" upon receiving the request.

1596 An updateInterval value in the PDR for the sensor provides a way for the requester to determine the  
1597 maximum time from when a sensor was re-armed or accessed to when the subsequent eventState or  
1598 reading update should have occurred.

1599 For a sensor that polls for updates, the updateInterval corresponds to the nominal polling interval,  $\pm 50\%$ .  
1600 (The  $\pm 50\%$  variation is to accommodate manufacturing variations between devices implementing sensors  
1601 and variations in firmware-based polling intervals.) There is no requirement for a sensor's polling interval  
1602 to be synchronized (restarted) when a re-arm occurs. A sensor is also allowed to take as long as two  
1603 polling intervals before updating its state following a re-arm (one interval to recognize the re-arm, and one  
1604 interval to collect and apply the updated state).

1605 For a sensor that updates "on demand," the updateInterval indicates the maximum time,  $\pm 50\%$ , from  
1606 receiving a GetSensorReading command to when a reading and status update should occur. If the sensor  
1607 can update itself within the PLDM Request-to-response time (refer to [DSP0240](#)), either an updateInterval  
1608 value of 0 or the actual update interval may be used in the PDR.

1609 If the updateInterval for a given sensor is longer than the PLDM Request-to-response time, the  
1610 updateInterval must be specified and the sensorOperationalStatus must be returned as "initializing" while  
1611 the sensor is performing its initial state assessment after being enabled or re-armed.

1612 Because a sensor is allowed to take up to two polling intervals to update after a re-arm, and because the  
1613 variation is allowed to be  $\pm 50\%$ , it may take as long as three nominal polling intervals (two nominal  
1614 intervals times 1.5) plus a PLDM Request-to-response time before the effect of a re-arm is realized.

### 1615 **17.6 Thresholds, Present State, and Event State**

1616 PLDM Numeric Sensors that are threshold-based have associated thresholds against which the reading  
1617 is compared.

#### 1618 **17.6.1 Threshold severity levels**

1619 Each threshold is associated with a severity that is related to how far the threshold is from the normal  
1620 range of the sensor. Unless otherwise specified, the severity level is generally based on the view that a  
1621 sensor is monitoring parameters that are associated with a physical entity. Table 17 describes the  
1622 threshold severity levels.

1623

**Table 17 – Threshold severity levels**

Severity level	Description
<b>warning</b>	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.
<b>critical</b>	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.
<b>fatal</b>	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.

1624 **17.6.2 Upper and lower thresholds**

1625 A given threshold for a PLDM Numeric Sensor can either be an upper or a lower threshold. Upper  
 1626 thresholds are for tracking events that become more severe as the reading becomes more positive  
 1627 numerically. Lower thresholds are for events that become more severe as the reading becomes more  
 1628 negative numerically.

1629 PLDM has three upper thresholds: upper warning, upper critical, and upper fatal. Similarly, PLDM has  
 1630 three lower thresholds: lower warning, lower critical, and lower fatal. By convention, these thresholds  
 1631 occur in the following order: lower fatal, lower critical, lower warning, upper warning, upper critical, and  
 1632 upper fatal. Lower fatal corresponds to the most negative threshold value, and upper fatal corresponds to  
 1633 the most positive threshold value. This order is illustrated in Figure 22 on page 65.

1634 A sensor is not required to implement all thresholds. For example, a sensor that monitors for an over-  
 1635 voltage condition may implement only an upper critical threshold. A sensor that is monitoring a low-RPM  
 1636 condition may implement only lower warning and lower critical thresholds. A temperature sensor may  
 1637 implement both upper and lower thresholds so that it can track both over-temperature and under-  
 1638 temperature conditions.

1639 **17.6.3 Present State**

1640 A PLDM Numeric Sensor that uses thresholds returns a presentState value that is based on a simple  
 1641 numeric comparison of the present reading against the sensor to the thresholds and returns the threshold  
 1642 range with which the reading is associated. The presentState value is updated solely based on a numeric  
 1643 comparison of the present reading to the thresholds. For upper thresholds, the presentState value is  
 1644 based on whether the present reading is greater than or equal to the threshold value. For lower  
 1645 thresholds, the presentState value is based on whether the present reading is less than or equal to the  
 1646 threshold value. For example, if the presentState value is greater than or equal to the value for upper  
 1647 critical threshold but is less than the value for upper fatal threshold, the presentState value will be  
 1648 UpperCritical.

#### 1649 **17.6.4 Event State**

1650 The eventState field of a PLDM Numeric Sensor is updated based on transitions between the different  
1651 monitored states of the sensor. Unlike presentState, the eventState value includes the effect of the  
1652 hysteresis setting. If the hysteresis value for the sensor is equal to one count of the reading, the  
1653 eventState and presentState values will be the same. Otherwise, the eventState setting may vary from  
1654 the presentState due to the effect of hysteresis. See 17.9 for more information about hysteresis and its  
1655 relationship to eventState.

1656 The eventState behavior is also affected by whether the sensor implementation is manual- or auto-rearm  
1657 (see 17.7).

#### 1658 **17.7 Manual re-arm and auto re-arm sensors**

1659 The event state tracking for a sensor can be either auto re-arm or manual re-arm. An auto re-arm sensor  
1660 updates its eventState automatically whenever the sensor detects that a state transition has occurred.

1661 A manual re-arm sensor retains the most severe event state transition that it has detected since the time  
1662 the sensor was initialized or since the last time the eventState value was explicitly cleared (using the  
1663 rearm operation in the GetSensorReading command). If a new state is assessed that has the same  
1664 criticality as the previous state, the most recently assessed value shall be returned. For example, if the  
1665 previous value was upperCritical and the presentState value is lowerCritical, then upperCritical shall be  
1666 returned.

1667 Thus, auto re-arm sensors automatically update their status on *any* detected state transition, while  
1668 manual re-arm sensors automatically update their eventState value only on detecting a worsening  
1669 (increasing severity) transition (or upon a transition to a different state of equivalent severity as the  
1670 previous state).

1671 Re-arming of numeric sensors is done through the GetSensorReading command. Re-arming causes the  
1672 sensor to internally enter its "initializing" operating state until it next updates its presentState and  
1673 eventState. (This update may happen so quickly that the temporary entry into the initializing state is never  
1674 reflected in the sensorOperationalState parameter of the GetSensorReading command.)

#### 1675 **17.8 Event message generation**

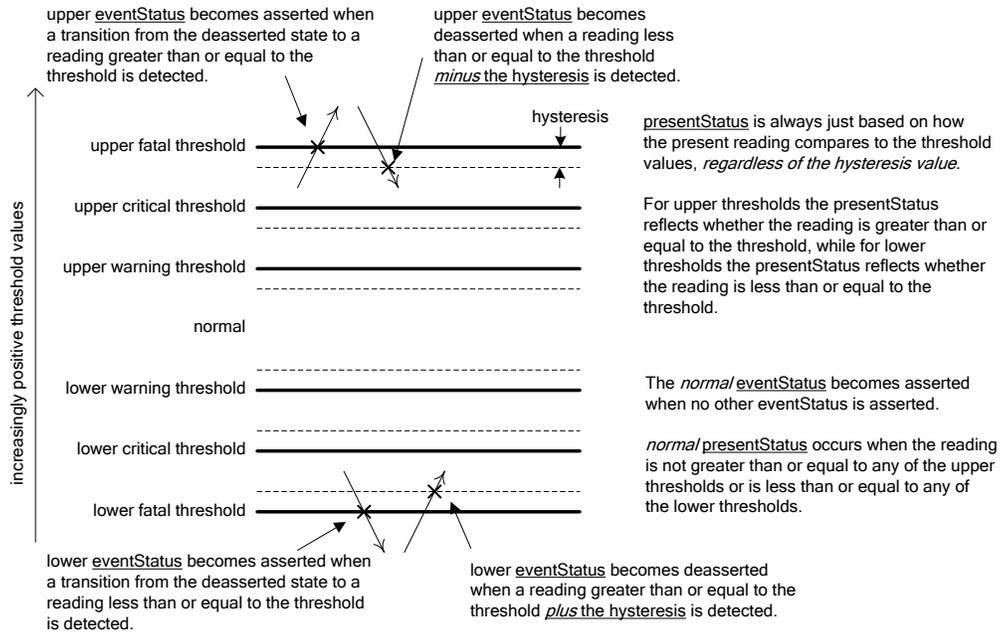
1676 A PLDM Numeric Sensor that supports and is enabled to generate event messages shall generate them  
1677 whenever an Event State (eventState) change is detected. To detect changes in the Event State, the  
1678 sensor implementation must do periodic polling or incorporate some other asynchronous mechanism,  
1679 such as the occurrence of an interrupt, which causes the sensor to obtain a new reading, the eventState  
1680 to update and an event message to be generated.

#### 1681 **17.9 Threshold values and hysteresis**

1682 Threshold settings for PLDM Numeric Sensors are required to be ordered from numerically most negative  
1683 to most positive in the following order: lower fatal, lower critical, lower warning, upper warning, upper  
1684 critical, upper fatal. The hysteresis value is always subtracted from the "upper" thresholds and added to  
1685 the "lower" thresholds.

1686 Thus, hysteresis is always applied on the transition from a more severe state to a less severe state. For  
1687 example, assume that a sensor has a hysteresis value of 2, has an upper critical threshold set to 80, and  
1688 is presently in the "upper warning" state. The sensor will transition to the "upper critical" state when it  
1689 detects that the reading value reaches a value that is greater than or equal to the threshold setting of 80.  
1690 The sensor is now in the "upper critical" state. To return to the "upper warning" state, the reading has to  
1691 drop to 78 (80 minus the hysteresis value of 2).

1692 Figure 22 helps further describe and illustrate the relationships between thresholds, hysteresis,  
 1693 eventState, and presentState for numeric sensors.



1694

1695

**Figure 22 – Numeric sensor threshold and hysteresis relationships**

1696 **18 PLDM Numeric Sensor commands**

1697 This clause describes the commands for accessing PLDM Numeric Sensors per this specification. The  
 1698 command numbers for the PLDM messages are given in clause 30.

1699 If PLDM numeric sensors are implemented, the Mandatory/Optional/Conditional (M/O/C) requirements  
 1700 shown in Table 18 apply.

1701 **Table 18 – Numeric Sensor commands**

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

1702 <sup>[1]</sup> The GetSensorThresholds command is required if the SetSensorThresholds command is implemented. Otherwise,  
 1703 the command is optional.

1704 <sup>[2]</sup> The GetSensorHysteresis command is required if the SetSensorHysteresis command is implemented. Otherwise,  
 1705 the command is optional.

1706 <sup>[3]</sup> The InitNumericSensor command is required if the sensor requires initialization following any one of the conditions  
 1707 identified in the initConditions field of the PLDM Numeric Sensor Initialization PDR.

1708 **18.1 SetNumericSensorEnable command**

1709 The SetNumericSensorEnable command is used to set the operating state of the sensor itself and  
 1710 whether the sensor generates event messages. Changing this state affects only the operation of the  
 1711 sensor; it has no effect on the operational state of the entity or parameter that is being monitored. Event  
 1712 message generation is optional for a sensor. Table 19 describes the format of the command.

1713 **Table 19 – SetNumericSensorEnable command format**

Type	Request data
uint16	<b>sensorID</b> A handle that is used to identify and access the sensor special values: 0x0000, 0xFFFF = reserved
enum8	<b>sensorOperationalState</b> 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	<p><b>sensorEventMessageEnable</b></p> <p>This value is used to enable or disable event message generation from the sensor.</p> <p>value: { noChange, disableEvents, enableEvents, enableOpEventsOnly, enableStateEventsOnly}</p> <p>noChange means do not alter the present setting. Use noChange when the sensor does not support event message generation.</p>
Type	Response data
enum8	<p><b>completionCode</b></p> <p>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. }</p>

1714 **18.2 GetSensorReading command**

1715 The GetSensorReading command is used to get the present reading and threshold event state values  
 1716 from a numeric sensor, as well as the operating state of the sensor itself. Table 20 describes the format of  
 1717 the command. NOTE: The Numeric Sensor PDR sensorID type, in clause 28.4 Numeric Sensor PDR has  
 1718 been changed in version 1.1.1 of this specification from uint8 to uint16 to be consistent with  
 1719 GetSensorReading command.

1720 **Table 20 – GetSensorReading command format**

Type	Request data
uint16	<p><b>sensorID</b></p> <p>A handle that is used to identify and access the sensor</p> <p>special values: 0x0000, 0xFFFF reserved</p>
bool8	<p><b>rearmEventState</b></p> <p>true = manually re-arm EventState after responding to this request</p> <p>Re-arming causes the sensor to enter the "initializing" state until it updates its presentState and eventState.</p> <p>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 eventState.</p> <p>If event messages are enabled, the status update shall also cause the sensor to issue a corresponding assertion event message based on the eventState that it assesses. This includes generating an event message for the "normal" state.</p> <p>false = no manual re-arm</p>

1721

Type	Response data
enum8	<p><b>completionCode</b></p> <p>value: { PLDM_BASE_CODES, INVALID_SENSOR_ID = 0x80, REARM_UNAVAILABLE_IN_PRESENT_STATE = 0x81 }</p>
enum8	<p><b>sensorDataSize</b></p> <p>The bit width and format of reading and threshold values that the sensor returns</p> <p>value: { uint8, sint8, uint16, sint16, uint32, sint32 }</p>
enum8	<p><b>sensorOperationalState</b></p> <p>The state of the sensor itself</p> <p>value: { enabled, disabled, unavailable, statusUnknown, failed, initializing, shuttingDown, inTest }</p> <p>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.</p> <p>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.</p> <p>disabled            The sensor is disabled from returning presentReading and event state values. This state is settable through the SetNumericSensorEnable command.</p> <p>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.</p> <p>statusUnknown    The sensor cannot presently return valid state or reading information for the monitored entity.</p> <p>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.</p> <p>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 eventState values shall be ignored while the sensor is in this state.</p> <p>shuttingDown     The sensor is transitioning to the disabled, failed, or unavailable states.</p> <p>inTest             The sensor is presently undergoing testing.</p> <p>NOTE: The operation of sensor testing and the mechanisms for sensor testing are outside the scope of this specification.</p>
enum8	<p><b>sensorEventMessageEnable</b></p> <p>value: { noEventGeneration, eventsDisabled, eventsEnabled, opEventsOnlyEnabled, stateEventsOnlyEnabled }</p>

Type	Response data
enum8	<p><b>presentState</b></p> <p>The most recently assessed state value monitored by the sensor. Refer to 17.5 for additional information on how presentState is assessed.</p> <p>If the sensorOperationalState is set to enabled the sensor must return a value other than "Unknown" for the presentState.</p> <p>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 17.6 for important information about how presentState and eventState are generated.</p> <p>value: { Unknown, Normal, Warning, Critical, Fatal, LowerWarning, LowerCritical, LowerFatal, UpperWarning, UpperCritical, UpperFatal }</p>
enum8	<p><b>previousState</b></p> <p>The state that the presentState was entered from. This must be different from the present state (with the exception that there may be conditions where both the presentState and previousState are returned as "Unknown").</p> <p>The previousState is updated whenever the presentState is assessed as different from the previously assessed value for presentState. Refer to 17.5 for additional information on how presentState is assessed.</p> <p>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".</p> <p>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 17.6 for important information about how presentState and eventState are generated.</p> <p>value: { Unknown, Normal, Warning, Critical, Fatal, LowerWarning, LowerCritical, LowerFatal, UpperWarning, UpperCritical, UpperFatal }</p>
enum8	<p><b>eventState</b></p> <p>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.</p> <p>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".</p> <p>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 17.6 for additional information about how presentState and eventState are generated.</p> <p>value: { Unknown, Normal, Warning, Critical, Fatal, LowerWarning, LowerCritical, LowerFatal, UpperWarning, UpperCritical, UpperFatal }</p>
uint8   sint8   uint16   sint16   sint32   uint32	<p><b>presentReading</b></p> <p>The present value indicated by the sensor</p> <p>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 sensorOperationalState value of the sensor is Unavailable.</p>

1722 **18.3 GetSensorThresholds command**

1723 The GetSensorThresholds command is used to get the present threshold settings for a PLDM Numeric  
 1724 Sensor. Table 21 describes the format of the command.

1725 **Table 21 – GetSensorThresholds command format**

Type	Request data
uint16	<b>sensorID</b> A handle that is used to identify and access the sensor special values: 0x0000, 0xFFFF = reserved
Type	Response data
enum8	<b>completionCode</b> value: { PLDM_BASE_CODES, INVALID_SENSOR_ID = 0x80 }
enum8	<b>sensorDataSize</b> 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.
<i>For sensorDataSize = uint8 or sint8</i>	
uint8   sint8	<b>upperThresholdWarning</b>
uint8   sint8	<b>upperThresholdCritical</b>
uint8   sint8	<b>upperThresholdFatal</b>
uint8   sint8	<b>lowerThresholdWarning</b>
uint8   sint8	<b>lowerThresholdCritical</b>
uint8   sint8	<b>lowerThresholdFatal</b>
<i>For sensorDataSize = uint16 or sint16</i>	
uint16   sint16	<b>upperThresholdWarning</b>
uint16   sint16	<b>upperThresholdCritical</b>
uint16   sint16	<b>upperThresholdFatal</b>
uint16   sint16	<b>lowerThresholdWarning</b>
uint16   sint16	<b>lowerThresholdCritical</b>
uint16   sint16	<b>lowerThresholdFatal</b>

Type	Response data
<i>For sensorDataSize = uint32 or sint32</i>	
uint32   sint32	<b>upperThresholdWarning</b>
uint32   sint32	<b>upperThresholdCritical</b>
uint32   sint32	<b>upperThresholdFatal</b>
uint32   sint32	<b>lowerThresholdWarning</b>
uint32   sint32	<b>lowerThresholdCritical</b>
uint32   sint32	<b>lowerThresholdFatal</b>

1726 **18.4 SetSensorThresholds command**

1727 The SetSensorThresholds command is used to set the thresholds of a PLDM Numeric Sensor. Values for  
 1728 all threshold parameters must be provided. However, if a particular threshold is not supported by the  
 1729 sensor, the value passed in the corresponding parameter is ignored. To avoid unintended event  
 1730 transitions, it is recommended that the sensor be disabled while changing threshold settings.

1731 Threshold values may be volatile or non-volatile. The level of volatility is reflected in the PDR for the  
 1732 sensor.

1733 Table 22 describes the format of the command.

1734 **Table 22 – SetSensorThresholds command format**

Type	Request data
uint16	<b>sensorID</b> A handle that is used to identify and access the sensor special values: 0x0000, 0xFFFF = reserved
enum8	<b>sensorDataSize</b> 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.
<i>For sensorDataSize = uint8 or sint8</i>	
uint8   sint8	<b>upperThresholdWarning</b>
uint8   sint8	<b>upperThresholdCritical</b>
uint8   sint8	<b>upperThresholdFatal</b>
uint8   sint8	<b>lowerThresholdWarning</b>
uint8   sint8	<b>lowerThresholdCritical</b>
uint8   sint8	<b>lowerThresholdFatal</b>

Type	Request data
<i>For sensorDataSize = uint16 or sint16</i>	
uint16   sint16	<b>upperThresholdWarning</b>
uint16   sint16	<b>upperThresholdCritical</b>
uint16   sint16	<b>upperThresholdFatal</b>
uint16   sint16	<b>lowerThresholdWarning</b>
uint16   sint16	<b>lowerThresholdCritical</b>
uint16   sint16	<b>lowerThresholdFatal</b>
<i>For sensorDataSize = uint32 or sint32</i>	
uint32   sint32	<b>upperThresholdWarning</b>
uint32   sint32	<b>upperThresholdCritical</b>
uint32   sint32	<b>upperThresholdFatal</b>
uint32   sint32	<b>lowerThresholdWarning</b>
uint32   sint32	<b>lowerThresholdCritical</b>
uint32   sint32	<b>lowerThresholdFatal</b>
Type	Response data
enum8	<b>completionCode</b> value: { PLDM_BASE_CODES, INVALID_SENSOR_ID=0x80 }

1735 **18.5 RestoreSensorThresholds command**

1736 The RestoreSensorThresholds command restores default thresholds for the device. Table 23 describes  
1737 the format of the command.

1738 **Table 23 – RestoreSensorThresholds command format**

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

1739 **18.6 GetSensorHysteresis command**

1740 The GetSensorHysteresis command is used to read the present hysteresis setting for a PLDM Numeric  
1741 Sensor. The hysteresis value uses the same units, data size, and conversion factors that are specified for  
1742 the reading from the sensor. Table 24 describes the format of the command.

1743

**Table 24 – GetSensorHysteresis command format**

Type	Request data
uint16	<b>sensorID</b> A handle that is used to identify and access the sensor special values: 0x0000, 0xFFFF = reserved
Type	Response data
enum8	<b>completionCode</b> value: { PLDM_BASE_CODES, INVALID_SENSOR_ID=0x80 }
enum8	<b>sensorDataSize</b> The bit width of the hysteresis value that is being returned value: { uint8, sint8, uint16, sint16, uint32, sint32 }
<i>For sensorDataSize = uint8 or sint8</i>	
uint8   sint8	<b>hysteresis value</b>
<i>For sensorDataSize = uint16 or sint16</i>	
uint16   sint16	<b>hysteresis value</b>
<i>For sensorDataSize = uint32 or sint32</i>	
uint32   sint32	<b>hysteresis value</b>

1744 **18.7 SetSensorHysteresis command**

1745 The SetSensorHysteresis command is used to set the present hysteresis setting for a PLDM Numeric  
 1746 Sensor. The hysteresis value uses the same units, data size, and conversion factors that are specified for  
 1747 the reading from the sensor. It is recommended that the sensor be disabled while changing the hysteresis  
 1748 setting. Table 25 describes the format of the command.

1749

**Table 25 – SetSensorHysteresis command format**

Type	Request data
uint16	<p><b>sensorID</b></p> <p>A handle that is used to identify and access the sensor</p> <p>special values: 0x0000, 0xFFFF = reserved</p>
enum8	<p><b>sensorDataSize</b></p> <p>The bit width and format for the following hysteresis value that is being set into the sensor value: { uint8, sint8, uint16, sint16, uint32, sint32 }</p> <p>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.</p>
<i>For sensorDataSize = uint8 or sint8</i>	
uint8   sint8	<b>hysteresis value</b>
<i>For sensorDataSize = uint16 or sint16</i>	
uint16   sint16	<b>hysteresis value</b>
<i>For sensorDataSize = uint32 or sint32</i>	
uint32   sint32	<b>hysteresis value</b>
Type	Response data
enum8	<p><b>completionCode</b></p> <p>value: { PLDM_BASE_CODES, INVALID_SENSOR_ID=0x80 }</p>

1750 **18.8 InitNumericSensor command**

1751 The InitNumericSensor command is typically used by the Initialization Agent function (see clause 15) to  
 1752 initialize PLDM Numeric Sensors. The command may also be used as an interface for “virtual sensors,”  
 1753 which do not actually poll and update their own state but instead rely on another management controller  
 1754 or system software to set their state.

1755 Implementations should avoid virtual sensors that require initialization by the Initialization Agent function.  
 1756 Conflicts could occur if the sensor needs to be accessed by the Initialization Agent function at the same  
 1757 time it is being accessed as a virtual sensor. Typically, however, a virtual sensor would not require  
 1758 initialization by the Initialization Agent function.

1759 Table 26 describes the format of the command.

1760

**Table 26 – InitNumericSensor command format**

Type	Request data
uint16	<p><b>sensorID</b></p> <p>A handle that is used to identify and access the sensor special values: 0x0000, 0xFFFF = reserved</p>
enum8	<p><b>sensorOperationalState</b></p> <p>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.</p> <p>This parameter is applied to the sensor <i>after</i> all other fields (sensorPresentState, eventMsgEnable, and numericReadingSetting) have been applied to the sensor.</p> <p>value: { enabled, disabled, unavailable }</p>
enum8	<p><b>sensorPresentState</b></p> <p>The expected present state of the numeric sensor. See the description of the presentState field in Table 20.</p>
enum8	<p><b>eventMsgEnable</b></p> <p>This value is used to enable or disable event message generation from the sensor.</p> <p>value: {     enableEventMessages,     disableEventMessages,     noChange=0xFF // Do not alter the present event enable setting. }</p>
bool8	<p><b>setNumericReading</b></p> <p>value: { false, true }</p> <p>True directs the receiver to accept the following numericReadingSetting.</p>
var	<p><b>numericReadingSetting</b></p> <p>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.</p>
Type	Response data
enum8	<p><b>completionCode</b></p> <p>value: { PLDM_BASE_CODES, INVALID_SENSOR_ID=0x80 }</p>

1761

## 19 PLDM State Sensors

1762

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](#).

1763

1764

1765

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.

1766

1767

1768 This specification also includes support for the definition of vendor-specific state sets using the OEM  
 1769 State Set PDR. (See 28.10 for more information.)

1770 **20 PLDM State Sensor commands**

1771 This clause describes the commands for accessing PLDM State Sensors per this specification. The  
 1772 command numbers for the PLDM messages are given in clause 30.

1773 If PLDM State Sensors are implemented, the Mandatory/Conditional (M/C) requirements shown in Table  
 1774 27 apply.

1775 **Table 27 – State Sensor commands**

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

1776 <sup>[1]</sup> Required for sensors that are to be initialized through the Initialization Agent function.

1777 **20.1 SetStateSensorEnables command**

1778 The SetStateSensorEnables command is used to set enable or disable sensor operation and event  
 1779 message generation for sensors within a PLDM Composite State Sensor. Event message generation is  
 1780 optional for a sensor. Table 28 describes the format of the command.

1781 **Table 28 – SetStateSensorEnables command format**

Type	Request data
uint16	<b>sensorID</b> A handle that is used to identify and access the sensor special values: 0x0000, 0xFFFF = reserved
uint8	<b>compositeSensorCount</b> The number of individual sets of sensor information that this command accesses. Up to eight sets of state sensor information (accessed as sensor offsets 0 through 7) can be accessed through a given sensorID within a PLDM terminus. value: 0x01 to 0x08
opField xN	<b>opFields</b> 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 29.
Type	Response data
enum8	<b>completionCode</b> value: { PLDM_BASE_CODES, INVALID_SENSOR_ID=0x80, EVENT_GENERATION_NOT_SUPPORTED = 0x82 }

1782

**Table 29 – SetStateSensorEnables opField format**

Type	Description
enum8	<p><b>sensorOperationalState</b></p> <p>The expected state of the sensor</p> <p>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.</p> <p>value: { enabled, disabled, unavailable }</p>
enum8	<p><b>eventMessageEnable</b></p> <p>This value is used to enable or disable event message generation from the sensor.</p> <p>value: { noChange, disableEvents, enableEvents, enableOpEventsOnly, enableStateEventsOnly }</p> <p>noChange means do not alter the present setting. Use noChange when the sensor does not support event message generation.</p> <p>NOTE: Event message generation is optional for a sensor.</p>

1783 **20.2 GetStateSensorReadings command**

1784 The GetStateSensorReadings command can return readings for multiple state sensors (a PLDM State  
1785 Sensor that returns more than one set of state information is called a composite state sensor).

1786 State information is returned as a sequence of one to N "stateField" structures. The first stateField  
1787 structure is referred to as the structure for the sensor at offset 0, second is for the sensor at offset 1, and  
1788 so on.

1789 The same number of stateField structures must be returned and in the same sequence during platform  
1790 management subsystem operation, regardless of the operational status of the sensors.

1791 Table 30 describes the format of the command.

1792

**Table 30 – GetStateSensorReadings command format**

Type	Request data
uint16	<p><b>sensorID</b></p> <p>A handle that is used to identify and access the simple or composite sensor</p> <p>special values: 0x00, 0xFFFF = reserved</p>
bitfield8	<p><b>sensorRearm</b></p> <p>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 offset 0) and bit [7] corresponds to the eighth sensor (sensor offset 7), sequentially.</p> <p>For each bit position [n] from n = 0 to compositeSensorCount-1, the bit setting operates as follows:</p> <p>0b = do not re-arm sensor [n]+1 1b = re-arm sensor [n]+1</p>
uint8	<p><b>reserved</b></p> <p>value: 0x00</p>

Type	Response data
enum8	<b>completionCode</b> value: { PLDM_BASE_CODES, INVALID_SENSOR_ID=0x80 }
unit8	<b>compositeSensorCount</b> The number of individual sets of sensor information that this command accesses. Up to eight sets of state sensor information (accessed as sensor offsets 0 through 7) can be accessed through a given sensorID within a PLDM terminus. value: 0x01 to 0x08
stateField xN	<b>stateFields</b> Each stateField is an instance of a stateField structure that is used to return the present operational state setting and the present state and event state for a particular set of sensor information contained within the state sensor. The stateField structure is defined in Table 31.

1793

**Table 31 – GetStateSensorReadings stateField format**

Type	Description
enum8	<b>sensorOperationalState</b> The state of the sensor itself See Table 20 for the enumeration values of sensorOperationalState.
enum8	<b>presentState</b> 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	<b>previousState</b> The state that the presentState was entered from. This must be different from the present state (with the exception that there may be conditions where both the presentState and previousState are returned as "Unknown"). The previousState is updated whenever the presentState is assessed as different from the previously assessed value for presentState. Refer to 17.5 for additional information on how presentState is assessed. 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	<b>eventState</b> 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.

1794 **20.3 InitStateSensor command**

1795 The InitStateSensor command is typically used by the Initialization Agent function (see clause 15) to  
1796 initialize PLDM State Sensors. The command may also be used as an interface for virtual sensors, which  
1797 do not actually poll and update their own state but instead rely on another management controller or  
1798 system software to set their state.

1799 Implementations should avoid virtual sensors that require initialization by the Initialization Agent function.  
1800 Conflicts could occur if the sensor needs to be accessed by the Initialization Agent function at same time  
1801 it is being accessed as a virtual sensor. Typically, however, a virtual sensor would not require initialization  
1802 by the Initialization Agent function.

1803 Table 32 describes the format of the command.

1804 **Table 32 – InitStateSensor command format**

Type	Request data
uint16	<p><b>sensorID</b></p> <p>A handle that is used to identify and access the sensor</p> <p>special values: 0x0000, 0xFFFF = reserved</p>
unit8	<p><b>compositeSensorCount</b></p> <p>The number of individual sets of sensor information that this command accesses. Up to eight sets of state sensor information (accessed as sensor offsets 0 through 7) can be accessed through a given sensorID within a PLDM terminus.</p> <p>value: 0x01 to 0x08</p>
initField xN	<p>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 33.</p>
Type	Response data
enum8	<p><b>completionCode</b></p> <p>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 }</p>

1805 **Table 33 – InitStateSensor initField format**

Type	Description
enum8	<p><b>sensorOperationalState</b></p> <p>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.</p> <p>This parameter is applied to the sensor after all other fields (sensorPresentState and eventMsgEnable) have been applied to the sensor.</p> <p>value: { enabled, disabled, unavailable }</p>
enum8	<p><b>sensorPresentState</b></p> <p>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.</p> <p>value: { dependent on sensor State Set }</p>
enum8	<p><b>eventMsgEnable</b></p> <p>This value is used to enable or disable event message generation from the sensor.</p> <p>value: { enableEvents, disableEvents, noChange=0xFF }</p> <p>noChange means do not alter the present setting.</p>

## 1806 **21 PLDM effecters**

1807 PLDM effecters provide a general mechanism for controlling or configuring a state or numeric setting of  
1808 an entity. PLDM effecters are similar to PLDM sensors, except that entity state and numeric setting values  
1809 are written into an effector rather than read from it.

1810 PLDM commands are specified for writing the state or numeric setting to an effector. Effecters are  
1811 identified by and accessed using an EffectorID that is unique for each effector within a given terminus.  
1812 Corresponding PDRs provide basic semantic information for effecters, such as what type of states or  
1813 numeric units the effector accepts, what terminus and EffectorID value are used to access the effector,  
1814 which entity the effector is associated with, and so on.

### 1815 **21.1 PLDM State Effecters**

1816 PLDM State Effecters provide a regular command structure for setting state information in order to  
1817 change the state of an entity. Effecters use the same PLDM State Sets definitions as PLDM State  
1818 Sensors, but instead of using the state set information to interpret the value that is read from a sensor,  
1819 the state sets are used to define the value to write to an effector. Like PLDM Composite State Sensors,  
1820 PLDM State Effecters can be implemented and accessed as composite state effecters where a single  
1821 EffectorID is used to access a set of state effecters. This enables multiple states to be set using a single  
1822 command and to share a single PDR that provides the basic information for the effecters.

### 1823 **21.2 PLDM Numeric Effecters**

1824 PLDM Numeric Effecters provide a regular command structure for setting a numeric value for a  
1825 controllable parameter of an entity. Numeric effecters use the same definition of units as the units for  
1826 readings returned by numeric sensors (see 27.2). For example, a numeric effector could be used to set a  
1827 value for revolutions per second.

### 1828 **21.3 Effector semantics**

1829 An effector has a meaning or use that is associated with what an effector does or is used for. This will be  
1830 referred to as the "effector semantic", or just the "semantic."

1831 Although PLDM effecters provide a straightforward mechanism for setting a state or numeric value for an  
1832 entity, conveying the semantic of how that state or numeric value affects the entity, or how the setting  
1833 should be used, is not always straightforward.

1834 Suppose a numeric effector is defined for setting a fan speed. A PDR for the numeric effector can readily  
1835 indicate that the effector is for "Physical Fan 1", and that "Fan 1" is contained by Processor 1. The PDR  
1836 can also indicate that the units for the setting are "RPM". However, this does not convey what the RPM is  
1837 actually doing. For example, is the RPM a speed limit or a target speed?

1838 Additionally, other information may be necessary for understanding how the effector is to be used. If a fan  
1839 speed needs to be set because one or more temperatures have become too high, how does the user of  
1840 PLDM know which temperatures are associated with the fan, and what RPM value should be set for a  
1841 particular temperature?

1842 The information required to describe the meaning and use of an effector can vary significantly depending  
1843 on how generic or specific the use is to the platform implementation. The level of generality of effector  
1844 semantics in PLDM is categorized as shown in Table 34.

1845

**Table 34 – Categories for effector 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 effector PDRs, is sufficient to convey the semantic for the effector. For example, the state set for System Power State when combined with "System" as the containerID identifies an effector 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 effector, but the effector use can be indicated by providing a single "Semantic ID" value that identifies a predefined semantic for the effector. For example, a Semantic ID could be defined for "System Power Down with Delay" where the definition specifies that the effector accepts a time value that identifies a delay from 1 to 60 seconds and triggers a system power down after that delay when the effector 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 effector PDR information and the Semantic ID are not sufficient to identify the semantic of the effector, but the semantic can be communicated when the Semantic ID is used with other PDRs. For example, an effector 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 effector 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 effector's setting.
External Information Required	The effector semantic may not be described using the mechanisms offered by this specification. In some cases, use of the effector may require access to information that is not provided through PDRs—for example, an effector where the user (such as a MAP) requires access to SMBIOS data to understand how the effector should be used. In other cases, the effector semantic may have a private or proprietary where the effector 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.

1846 The most generic and efficient use of effectors comes when they fall into the state sets or units only  
 1847 category and use standard state set or units definitions. The second most generic and efficient use of  
 1848 effectors is when they use a standard defined Semantic ID. Thus, if new standard effector semantics  
 1849 need to be defined, it should be first examined whether a new state set or units definition should be  
 1850 added to the specifications, or whether a new Semantic ID should be added.

1851 **21.4 PLDM and OEM effector semantic IDs**

1852 Effector Semantic ID values are specified in [DSP0249](#). A range of values is reserved for definition by the  
 1853 DMTF PLDM specifications and another range of values is available for OEM (vendor defined) effector  
 1854 semantics. When the OEM range is used, the semantic is identified and optionally named using an OEM  
 1855 Effector Semantic PDR. The use of the OEM Effector Semantic PDR is similar to how OEM units, entities,  
 1856 and state sets are defined within the PDRs.

1857 **22 PLDM effector commands**

1858 This clause describes the commands for accessing PLDM effectors per this specification. The command  
 1859 numbers for the PLDM messages are given in clause 30.

1860 If PLDM Numeric Effectors or PLDM State Effectors are implemented, the Mandatory (M) requirements  
 1861 shown in Table 35 apply.

1862

**Table 35 – State and Numeric Effector commands**

Command	M	Reference
SetNumericEffectorEnable	M <sup>[1]</sup>	See 22.1.
SetNumericEffectorValue	M <sup>[1]</sup>	See 22.2.
GetNumericEffectorValue	M <sup>[1]</sup>	See 22.3.
SetStateEffectorEnables	M <sup>[2]</sup>	See 22.4.
SetStateEffectorStates	M <sup>[2]</sup>	See 22.5.
GetStateEffectorStates	M <sup>[2]</sup>	See 22.6.

1863

<sup>[1]</sup> Required if one of more numeric effectors are implemented

1864

<sup>[2]</sup> Required if one or more state effectors are implemented

1865 **22.1 SetNumericEffectorEnable command**

1866 The SetNumericEffectorEnable command is used to enable or disable effector operation. A disabled  
 1867 effector cannot have its state updated. An effector may have a default state that it automatically returns to  
 1868 when it is disabled. An effector may also be able to be returned to its default state through the  
 1869 SetStateNumericEffectorValue command. The PLDM Numeric Effector PDR can describe a numeric  
 1870 effector and whether it has a default state. NOTE: The Numeric Effector PDR effectorID type, in clause  
 1871 28.11 Numeric Effector PDR has been changed in version 1.1.1 of this specification from uint8 to uint16  
 1872 to be consistent with SetNumericEffectorEnable command.

1873 Table 36 describes the format of this command.

1874

**Table 36 – SetNumericEffectorEnable command format**

Type	Request data
uint16	<b>effectorID</b> A handle that is used to identify and access the effector special values: 0x0000, 0xFFFF = reserved
enum8	<b>effectorOperationalState</b> The expected state of the effector. This enumeration is a subset of the operational state values that are returned by the GetStateEffectorStates command. Refer to the GetStateEffectorStates command for the definition of the values in this enumeration. value: { enabled, disabled = 2, unavailable }
Type	Response data
enum8	<b>completionCode</b> value: { PLDM_BASE_CODES, INVALID_EFFECTER_ID=0x80 }

1875 **22.2 SetNumericEffectorValue command**

1876 The SetNumericEffectorValue command is used to set the value for a PLDM Numeric Effector. Table 37  
1877 describes the format of this command.

1878

**Table 37 – SetNumericEffectorValue command format**

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

1879 **22.3 GetNumericEffectorValue command**

1880 The GetNumericEffectorValue command is used to return the present numeric setting of a PLDM Numeric  
 1881 Effector. Table 38 describes the format of this command.

1882 **Table 38 – GetNumericEffectorValue command format**

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

Type	Response data
uint8   sint8   uint16   sint16   sint32   uint32	<p><b>pendingValue</b></p> <p>The pending numeric value setting of the effector. The effectorDataSize field indicates the number of bits used for this field.</p>
uint8   sint8   uint16   sint16   sint32   uint32	<p><b>presentValue</b></p> <p>The present numeric value setting of the effector. The effectorDataSize indicates the number of bits used for this field.</p>

1883 **22.4 SetStateEffectorEnables command**

1884 The SetStateEffectorEnables command is used to enable or disable effector operation. A disabled  
 1885 effector cannot have its state updated. An effector may have a default state that it automatically returns to  
 1886 when it is disabled. An effector may also be able to be returned to its default state through the  
 1887 SetStateEffectorStates command. The PLDM State Effector PDR describes a state effector and whether  
 1888 it has a default state. Table 39 describes the format of this command.

1889 **Table 39 – SetStateEffectorEnables command format**

Type	Request data
uint16	<p><b>effectorID</b></p> <p>A handle that is used to identify and access the effector</p> <p>special values: 0x0000, 0xFFFF = reserved</p>
uint8	<p><b>compositeEffectorCount</b></p> <p>The number of individual sets of state effector information that are accessed by this command. Up to eight sets of effector information (accessed as effector offsets 0 through 7) can be accessed through a given effectorID within a PLDM terminus.</p> <p>value: 0x01 to 0x08</p>
opField xN	<p><b>opFields</b></p> <p>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 effector. The opField structure is defined in Table 40.</p>
Type	Response data
enum8	<p><b>completionCode</b></p> <p>value: { PLDM_BASE_CODES, INVALID_EFFECTER_ID=0x80 }</p>

1890

**Table 40 – SetStateEffectorEnables opField format**

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

1891 **22.5 SetStateEffectorStates command**

1892 The SetStateEffectorStates command is used to set the state of one or more effectors within a PLDM  
 1893 State Effector. Table 41 describes the format of this command.

1894

**Table 41 – SetStateEffectorStates command format**

Type	Request data
uint16	<p><b>effectorID</b></p> <p>A handle that is used to identify and access the effector</p> <p>special values: 0x0000, 0xFFFF = reserved</p>
unit8	<p><b>compositeEffectorCount</b></p> <p>The number of individual sets of effector information that are accessed by this command. Up to eight sets of state effector information (accessed as effector offsets 0 through 7) can be accessed through a given effectorID within a PLDM terminus.</p> <p>value: 0x01 to 0x08</p>
stateField xN	<p>Each stateField is an instance of a stateField structure that is used to set the requested state for a particular effector within the state effector. The stateField structure is defined in Table 42.</p>
Type	Response data
enum8	<p><b>completionCode</b></p> <p>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. }</p>

1895

**Table 42 – SetStateEffectorStates stateField format**

Type	Description
enum8	<p><b>setRequest</b></p> <p>value: {</p> <p style="padding-left: 40px;">noChange, // Do not request a change of the state of this effector.</p> <p style="padding-left: 40px;">requestSet // Request the effector state to be set to the state given by the following // effectorState value.</p> <p>}</p>
enum8	<p><b>effectorState</b></p> <p>The expected state of the effector. The state values come from the particular state set used for the implementation of the effector.</p> <p>value: { dependent on effector state set }</p>

1896

**22.6 GetStateEffectorStates command**

1897  
1898

The GetStateEffectorStates command is used to get the present state of an effector. Table 43 describes the format of this command.

1899

**Table 43 – GetStateEffectorStates command format**

Type	Request data
uint16	<p><b>effectorID</b></p> <p>A handle that is used to identify and access the simple or composite effector</p> <p>special values: 0x0000, 0xFFFF = reserved</p>
Type	Response data
enum8	<p><b>completionCode</b></p> <p>value: { PLDM_BASE_CODES, INVALID_EFFECTER_ID=0x80 }</p>
unit8	<p><b>compositeEffectorCount</b></p> <p>The number of individual sets of effector information that are accessed by this command. Up to eight sets of state effector information (accessed as effector offsets 0 through 7) can be accessed through a given effectorID within a PLDM terminus.</p> <p>value: 0x01 to 0x08</p>
stateField xN	<p><b>stateFields</b></p> <p>Each stateField is an instance of a stateField structure that is used to return the present operational state setting and the present state for a particular effector contained within the state effector. The stateField structure is defined in Table 44.</p>

1900

**Table 44 – GetStateEffectorStates stateField format**

Type	Description
enum8	<p><b>effectorOperationalState</b></p> <p>The state of the effector itself</p> <p>See Table 38 for the enumeration values of effectorOperationalState.</p>
enum8	<p><b>pendingState</b></p> <p>If the value of effectorOperationalState 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 effector. The effector implementation should return the "Unknown" state value whenever the effectorOperationalState 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 effectorOperationalState is anything except enabled-updatePending or enabled-noUpdatePending.</p> <p>value: { dependent on effector state set on which the effector implementation is based }</p>
enum8	<p><b>presentState</b></p> <p>The present state of the effector. The effector implementation should return the "Unknown" state value whenever the value of effectorOperationalState 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 effectorOperationalState is anything except enabled-updatePending or enabled-noUpdatePending.</p> <p>value: { dependent on the state set used for the effector implementation }</p>

1901 **23 PLDM Event Log commands**

1902 This clause describes the commands for accessing a PLDM Event Log per this specification. The  
 1903 command numbers for the PLDM messages are given in clause 30.

1904 The PLDM Event Log is typically accessed through the same PLDM terminus as the Event Receiver.  
 1905 However, this is not mandatory. The PDRs include information that describes which terminus is used to  
 1906 access the PLDM Event Log.

1907 If a PLDM Event Log is implemented, the Mandatory/Optional/Conditional (M/O/C) requirements shown in  
 1908 Table 45 apply.

1909

**Table 45 – 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 <sup>[1]</sup>	See 23.8.
FindPLDMEventLogEntry	O	See 23.9

1910

<sup>[1]</sup> Required if the PLDMEventLog implementation supports configurable policy parameters

1911 **23.1 GetPLDMEventLogInfo command**

1912 The GetPLDMEventLogInfo command returns basic information about the PLDM Event Log, such as its  
 1913 operational status, percentage used, and time stamps for the most recent add and erase actions. Table  
 1914 46 describes the format of the command.

1915 **Table 46 – GetPLDMEventLogInfo command format**

Type	Request data
–	none
Type	Response data
enum8	<b>completionCode</b> value: { PLDM_BASE_CODES }
enum8	<b>logOperationalStatus</b> 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	<b>activeLogClearingPolicy</b> 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	<p><b>entryCount</b></p> <p>number of entries presently in the Event Log</p>
uint8	<p><b>storagePercentUsed</b></p> <p>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</p> <p>value: 0 to 100</p> <p>special value: 0xFF = unspecified</p>
uint8	<p><b>percentWear</b></p> <p>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.</p> <p>value: 0x00 to 0x064 = wear in %</p> <p>special value: 0xFF = unspecified</p>
<p><b>mostRecentAddTimestamp</b></p> <p>The following three fields return the timestamp of the most recent addition or change to the log.</p> <p>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.</p> <p>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.</p>	
sint8	<p><b>mostRecentAddTimestampUTCOffset</b></p> <p>The UTC offset for the log entry timestamp in increments of 1/2 hour.</p> <p>special value: 0xFF = unspecified</p>
uint40	<p><b>mostRecentAddTimestampSeconds</b></p> <p>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.</p>
uint8	<p><b>mostRecentAddTimestamp100s</b></p> <p>This value provides a number of 1/100ths of a second added to <b>entryTimestampSeconds</b>.</p> <p>value: 0 to 99.</p> <p>special value: 0xFF = unspecified. This value is used if the implementation timestamps entries to no finer than a one second resolution.</p>
<p><b>mostRecentEraseTimestamp</b></p> <p>The following three fields return the most recent time that entries were deleted from the log or the log was cleared.</p> <p>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.</p> <p>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 if the timestamp has never been initialized, or if the timestamp has been lost due to an error or firmware update condition.</p>	

Type	Response data
sint8	<b>mostRecentEraseTimestampUTCOffset</b> The UTC offset for the log entry timestamp in increments of 1/2 hour. special value: 0xFF = unspecified
uint40	<b>mostRecentEraseTimestampSeconds</b> 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	<b>mostRecentEraseTimestamp100s</b> This value provides a number of 1/100ths of a second added to <b>entryTimestampSeconds</b> . 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.

1916 **23.2 EnablePLDMEventLogging command**

1917 The EnablePLDMEventLogging command is used to enable or disable the PLDM Event log from logging  
 1918 events. The log can be accessed and cleared while in the disabled state unless the logOperationalStatus  
 1919 is “failed”, in which case logging may not be able to be enabled. Table 47 describes the format of the  
 1920 command.

1921 **Table 47 – EnablePLDMEventLogging command format**

Type	Request data
enum8	<b>enableLogging</b> value: { disableLogging,       // Disable accepting events into the log. enableLogging        // Enable logging events. }
Type	Response data
enum8	<b>completionCode</b> value: { PLDM_BASE_CODES }
enum8	<b>logOperationalStatus</b> value: { See the definition of logOperationalStatus field for the GetPLDMEventLogInfo command (Table 46). }

1922 **23.3 ClearPLDMEventLog command**

1923 The ClearPLDMEventLog command is used to clear the contents of the PLDM Event Log. The execution  
 1924 of this command does not affect whether logging is enabled or disabled. Depending on the subsystem  
 1925 and its implementation, it is possible that events may be received or be in the process of being received  
 1926 during the terminus' execution of this command. If event logging is enabled, a terminus should continue to  
 1927 accept events while it is processing this command. It is recognized that in some implementations clearing  
 1928 the log device may take a significant amount of time. The number of events that an implementation may  
 1929 support queuing up while the log is being cleared is implementation dependent. Table 48 describes the  
 1930 format of this command.

1931

**Table 48 – ClearPLDMEventLog command format**

Type	Request data
–	none
Type	Response data
enum8	<b>completionCode</b> value: { PLDM_BASE_CODES }
enum8	<b>logOperationalStatus</b> 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 49). }

1932

### 23.4 GetPLDMEventLogTimestamp command

1933

The GetPLDMEventLogTimestamp command returns a snapshot of the present PLDM Event Log

1934

Timestamp time. Table 49 describes the format of this command.

1935

**Table 49 – GetPLDMEventLogTimestamp command format**

Type	Request data
–	none
Type	Response data
enum8	<b>completionCode</b> value: { PLDM_BASE_CODES }
sint8	<b>entryTimestampUTCOffset</b> The UTC offset for the log entry time stamp in increments of 1/2 hour special value: 0xFF = unspecified
uint40	<b>entryTimestampSeconds</b> 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	<b>entryTimestamp100s</b> This value provides a number of 1/100 of a second that is added to <b>entryTimestampSeconds</b> . 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.

1936 **23.5 SetPLDMEventLogTimestamp command**

1937 The SetPLDMEventLogTimestamp command can be used to set the PLDM Event Log Timestamp time.

1938 Some implementations may not implement the ability to set the time stamp to 1/100 of a second  
 1939 resolution and will round the time up or down to match the resolution that it supports. Therefore, the time  
 1940 stamp value in the response may vary from what was submitted because of rounding. The returned value  
 1941 may also vary due to delays in command response processing within the terminus.

1942 Implementations are required to support a 1 second or finer resolution for the time stamp. Table 50  
 1943 describes the format of this command.

1944 **Table 50 – SetPLDMEventLogTimestamp command format**

Type	Request data
sint8	<p><b>entryTimestampUTCOffset</b></p> <p>The UTC offset for the log entry time stamp in increments of 1/2 hour                      special value: 0xFF = unspecified</p>
uint40	<p><b>entryTimestampSeconds</b></p> <p>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).</p>
uint8	<p><b>entryTimestamp100s</b></p> <p>This value provides a number of 1/100 of a second that is added to <b>entryTimestampSeconds</b>.                      value: 0 to 99</p> <p>This value is ignored if the implementation only timestamps entries to a one second resolution.</p>
enum8	<p><b>logUpdateEvent</b></p> <p>value: {</p> <ul style="list-style-type: none"> <li>noEvent,</li> <li>logEvent // automatically logs a time stamp change event if the new time stamp clock                      // value is accepted. See <a href="#">DSP0249</a> for the state set definition for time                      // stamp change events.</li> </ul> <p>}</p>

1945

Type	Response data
enum8	<b>completionCode</b> value: { PLDM_BASE_CODES }
sint8	<b>entryTimestampUTCOffset</b> The UTC offset for the log entry time stamp in increments of 1/2 hour special value: 0xFF = unspecified
uint40	<b>entryTimestampSeconds</b> 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	<b>entryTimestamp100s</b> This value provides a number of 1/100 of a second that is added to <b>entryTimestampSeconds</b> . 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	<b>timestampResolution</b> 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)

1946 **23.6 ReadPLDMEventLog command**

1947 The ReadPLDMEventLog command can be used iteratively to read all or part of the entries in the PLDM  
1948 Event Log. Entries are returned one at a time. The data for one or more entries may be requested. Table  
1949 51 describes the format of this command.

1950 To use the command to start reading from the first entry in the log:

- 1951 • Set entryID to 0 and transferOperationFlag to GetFirstPart.
- 1952 • Issue the command to get the first portion of data for the first entry in the log.
- 1953 • Take the nextEntryID and nextTransferOperationFlag data from the response and use it as the  
1954 entryID and transferOperationFlag for the next request.
- 1955 • Repeat this until the desired number of entries have been read or the end of the log has been  
1956 reached.

1957 The FindPLDMEventLogEntry command can be used to get the entryID for an entry that is at an offset  
1958 into the log, or that has a timestamp that is older or newer than a given value. This entryID can then be  
1959 used in the ReadPLDMEventLog command, along with setting transferOperationFlag = GetFirstPart, to  
1960 begin reading the log starting with the found entry.

**Table 51 – ReadPLDMEventLog command format**

Type	Request data
uint32	<p><b>entryID</b></p> <p>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.</p> <p>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, <b>MC1</b>. (See clause 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.</p> <p>value: Set to 0x00000000 and transferOperationFlag = GetFirstPart to start reading from the first (oldest) entry in the log;</p>
enum8	<p><b>transferOperationFlag</b></p> <p>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.</p> <p>Possible values: {GetNextPart=0x00, GetFirstPart=0x01}</p>
Type	Response data
enum8	<p><b>completionCode</b></p> <p>Possible values:</p> <p>{ PLDM_BASE_CODES,  INVALID_TRANSFER_OPERATION_FLAG=0x81, INVALID_ENTRY_ID=0x82, }</p>
uint32	<p><b>nextEntryID</b></p> <p>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 <i>portion</i> 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.</p> <p>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.</p>

Type	Request data
enum8	<p><b>splitEntry</b></p> <p>value: {</p> <p>full, // All of the data for the entry is provided in the entryData field.</p> <p>firstFragment, // The eventData for the entry is split across ReadPLDMEventLogmessages. // The entryData field holds the first portion of the data for the entry.</p> <p>middleFragment, // The eventData for the entry is split across ReadPLDMEventLogmessages. // The entryData field holds a middle portion of the data for the entry.</p> <p>lastFragment // The eventData for the entry is split across ReadPLDMEventLogmessages. // The entryData field holds the last portion of the data for the entry.</p> <p>}</p>
–	<p><b>PLDMEventLogData</b></p> <p>The data or partial data for the requested PLDM Event Log entry. Entries are transferred starting from the oldest to the newest.</p>
<i>If splitEntry = lastFragment</i>	
uint8	<p><b>transferCRC</b></p> <p>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 6 using the polynomial <math>x^8 + x^2 + x^1 + 1</math> (This is the same polynomial used in the MCTP over SMBus/I<sup>2</sup>C transport binding specification). The CRC is calculated from most-significant bit to least-significant bit on bytes in the order that they are received. This field is only present when splitEntry = lastFragment.</p>

1962

**Table 52 – PLDMEventLogData format**

Type	Field
uint8	<p><b>transferredDataSize</b></p> <p>If splitEntry = full, then dataSize = number of bytes of entryData for the entire entry.</p> <p>If splitEntry = firstFragment, middleFragment, or lastFragment, then dataSize = number of bytes of entryData for the portion that is being transferred.</p>
–	<p><b>transferredEntryData</b></p> <p>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.</p>

1963

**23.7 GetPLDMEventLogPolicyInfo command**

1964

The GetPLDMEventLogPolicyInfo command returns details about the different log clearing policies that are supported for the particular PLDM Event Log implementation. Table 53 describes the format of this command.

1965

1966

Table 53 – GetPLDMEventLogPolicyInfo command format

Type	Request data
enum8	<p><b>logClearingPolicy</b></p> <p>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.</p> <p>value: { fillAndStop, FIFO, clearOnAge }</p>
Type	Response data
enum8	<p><b>completionCode</b></p> <p>value: { PLDM_BASE_CODES }</p>
bitfield8	<p><b>configurableParameterSupport</b></p> <p>This information and the following fields are specific to the logClearingPolicy that was selected in the request.</p> <p>[7:5] – reserved</p> <p>[4:3] – 00b = M and MPercentage are not configurable.                      01b = M is configurable                      10b = MPercentage is configurable.                      11b = reserved</p> <p>[2:1] – 00b = N and NPercentage are not configurable.                      01b = N is configurable.                      10b = NPercentage is configurable.                      11b = reserved</p> <p>[0] – 1b = Age is configurable.</p>
uint32	<p><b>NMin</b></p> <p>The smallest number that the implementation accepts or uses as a value for N for the given logClearingPolicy (see 13.4).</p> <p>special value: Return 0x00000000 if the policy implementation uses NPercentage instead of N, or if the policy does not use an N value.</p>
uint32	<p><b>NMax</b></p> <p>The largest number that the implementation accepts or uses as a value for N for the given logClearingPolicy (see 13.4).</p> <p>special value: Return 0x00000000 if the policy implementation uses NPercentage instead of N, or if the policy does not use an N value.</p>
Type	Response data
uint8	<p><b>NPercentageMin</b></p> <p>The smallest number that the implementation accepts or uses as a value for NPercentage for the given logClearingPolicy (see 13.4).</p> <p>value: 1 to 100; all other values = reserved</p> <p>special value: Return 0x00 if the policy implementation uses N instead of NPercentage, or if the policy does not use an NPercentage value.</p>

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

1968

1969 **23.8 SetPLDMEventLogPolicy command**

1970 The SetPLDMEventLogPolicy command is used to select and configure the PLDM Event Log clearing  
 1971 policies. Table 54 describes the format of the command.

1972 **Table 54 – SetPLDMEventLogPolicy command format**

Type	Request data
enum8	<p><b>selectedLogClearingPolicy</b></p> <p>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.</p> <p>value: { fillAndStop, FIFO, clearOnAge }</p>
enum8	<p><b>setOperation</b></p> <p>value: {</p> <p>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.</p> <p>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.</p> <p>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.</p> <p>}</p>
uint32	<p><b>N</b></p> <p>The number of entries that will be automatically cleared for the given selectedLogClearingPolicy. See 13.4 for a description of the log clearing policies.</p> <p>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.</p>
uint8	<p><b>NPercentage</b></p> <p>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.</p> <p>value: 1 to 100; all other values = reserved</p> <p>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.</p>
uint32	<p><b>M</b></p> <p>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.</p> <p>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.</p>

Type	Request data
uint8	<p><b>MPercentage</b></p> <p>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.</p> <p>value: 1 to 100; all other values = reserved</p> <p>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.</p>
uint32	<p><b>age</b></p> <p>This parameter sets the age interval in seconds for the given selectedLogClearingPolicy. See 13.4 for a description of the log clearing policies.</p> <p>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.</p>
Type	Response data
enum8	<p><b>completionCode</b></p> <p>value: { PLDM_BASE_CODES }</p>

1973 **23.9 FindPLDMEventLogEntry command**

1974 This command can be used to obtain the Entry ID value for the first entry in the Event Log that meets the  
 1975 identified search parameter. This value can then be used in the ReadPLDMEventLog command to start  
 1976 reading the log from that entry onward. The search parameters support finding the first entry that is newer  
 1977 or older than a specified timestamp value, or the entry that corresponds to a particular offset from the  
 1978 start or the present end of the log. Table 55 describes the format of this command.

1979 **Table 55 – FindPLDMEventLogEntry command format**

Type	Request data
enum8	<p><b>searchType</b></p> <p>value: {newerThan, olderThan, offsetFromStart, offsetFromEnd}</p>
uint32	<p><b>startingPoint</b></p> <p>The EntryID for the log entry or the offset from which searching will start. Searches include the entry at the identified starting point.</p> <p>The search always occurs in the direction from the start of the log (first entries) to the end of the log (last entries).</p> <p>If searchType = newerThan or olderThan:</p> <p style="padding-left: 40px;">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.</p> <p>If searchType = offsetFromStart:</p> <p style="padding-left: 40px;">The value identifies the Nth entry from the start of the log. For example, if starting point = 10 the search will start with the 10<sup>th</sup> entry at the beginning of the log. An error completionCode shall be returned if the value exceeds the number of entries in the log.</p> <p>If searchType = offsetFromEnd:</p> <p style="padding-left: 40px;">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 91<sup>st</sup> entry. An error completionCode shall be returned if the value exceeds the number of entries in the log.</p>
<p><b>compareTimestamp</b></p> <p><i>The compareTimestamp fields are only present when searchType = newerThan or olderThan.</i></p> <p><i>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.</i></p> <p><i>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.</i></p>	
sint8	<p><b>compareTimestampUTCOffset</b></p> <p>The UTC offset for the log entry timestamp in increments of 1/2 hour.</p> <p>special value: 0xFF = unspecified</p>
uint40	<p><b>compareTimestampSeconds</b></p> <p>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.</p>

Type	Request data
uint8	<p><b>compareTimestamp100s</b></p> <p>This value provides a number of 1/100ths of a second added to <b>entryTimestampSeconds</b>.</p> <p>value: 0 to 99.</p> <p>special value: 0xFF = unspecified. This value is used if the implementation timestamps entries to no finer than a one second resolution.</p>
Type	Response data
uint32	<p><b>entryID</b></p> <p>The entryID for the found log entry. This value can be used in the ReadPLDMEventLog command.</p> <p>special value: 0xFFFFFFFF = Not found. The command did not find a record matching the searchType.</p>
enum8	<p><b>completionCode</b></p> <p>value: { PLDM_BASE_CODES, INVALID_SEARCH_TYPE = 0x80 }</p>

1980 **24 PLDM State Sets**

1981 PLDM State Sets are specified enumerations for sets of state information that can be returned from  
 1982 PLDM state sensors. State sets may also be used to provide a common definition for state information  
 1983 used by other parts of PLDM.

1984 The state sets are the basis of state data that can be mapped as a data source into CIM properties that  
 1985 return state information, and also provide state information that can be used for monitoring and controlling  
 1986 the operation of PLDM itself.

1987 PLDM State Sets are defined in [DSP0249](#). This specification defines a numeric ID for each different state  
 1988 set, defines the enumeration values for the states that make up the set, and provides definitions for each  
 1989 state within the set. Because the state sets are expected to be extended over time as new CIM properties  
 1990 are defined, the state sets are maintained in a separate document to allow them to be extended without  
 1991 having to revise other PLDM specifications.

1992 **25 Platform Descriptor Records (PDRs)**

1993 PLDM can return collections of semantic and association information about the platform by using  
 1994 collections of information called Platform Descriptor Records (PDRs). This information can include  
 1995 records that return semantic information about sensors, such as their sensor resolution, tolerance,  
 1996 accuracy, and conversion factors, as well as records that return information about the associations  
 1997 between sensors and monitored entities, management controllers, effecters, and other platform  
 1998 associations or capabilities.

1999 PDRs are called descriptor records because they are mainly used to describe the subsystem, rather than  
 2000 to control it or configure it.

## 2001 **25.1 PDR Repository updates**

2002 A PDR Repository is not necessarily a static set of records. A platform that includes hot plug devices or  
2003 supports field updates may have its PDRs change over time as devices are added or removed. Even if  
2004 the implementation of a particular platform management subsystem is static, the PDRs must still be  
2005 generated and installed so that they represent the semantic information and relationships of the particular  
2006 platform implementation.

2007 PLDM does not specify the mechanisms by which PDRs get generated, installed, or updated. This was  
2008 done intentionally to allow the vendor of the PDR Repository devices to create update or configuration  
2009 utilities that are appropriate for the particular implementation. PLDM does, however, specify how the  
2010 information is accessed and used.

## 2011 **25.2 Internal storage and organization of PDRs**

2012 The PLDM specifications do not place any requirements on how PDRs are internally stored or organized  
2013 within the device or devices that implement the PDR Repository. PDRs may be compressed, stored with  
2014 additional pointers, sorted, cross indexed, split, replicated, and so on, as long as the information meets  
2015 the byte order and formats specified for the PDR commands. The byte order and formats for PDRs are  
2016 specified in tables for the different PDR types in clause 28.

## 2017 **25.3 PDR types**

2018 PDRs are identified by a PDR Type value that is given in a field in the header for each different PDR.  
2019 PDR types include type values for records that identify PDRs for PLDM numeric and state sensors,  
2020 records that direct sensor initialization, records that describe PLDM effecters, and so on. The PDR Type  
2021 values are given in Table 65.

## 2022 **25.4 PDR record handles**

2023 All PDRs are assigned an opaque numeric value called the recordHandle. This value is used for  
2024 accessing individual PDRs within the PDR Repository. Additional information about recordHandles and  
2025 their use is provided in the specification of the GetPDR command (see 26.2).

## 2026 **25.5 Accessing PDRs**

2027 For most implementations, PDR data rarely changes. A party that uses PDR information may want to  
2028 cache certain information to reduce the need for accessing the PDR Repository. The  
2029 GetPDRRepositoryInfo command provides time stamps that can be used to identify whether any record  
2030 data in a particular PDR Repository has changed. If a change is detected the party can then update its  
2031 cached information as necessary.

## 2032 **26 PDR Repository commands**

2033 This clause describes the commands for accessing PDRs from a PDR Repository per this specification.  
2034 The command numbers for the PLDM messages are given in clause 30.

2035 If a PDR Repository is implemented, the Mandatory/Optional/Conditional (M/O/C) requirements shown in  
2036 Table 56 apply.

2037

**Table 56 – PDR Repository commands**

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

2038  
2039  
2040  
2041  
2042

<sup>[1]</sup> 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.

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

2043 **26.1 GetPDRRepositoryInfo command**

2044 The GetPDRRepositoryInfo command returns information about the size and number of records in the  
2045 PDR Repository of a particular PLDM terminus, and time stamps that indicate the last time that an update  
2046 to the repository occurred. Two time stamps are returned, one that indicates whether any PLDM standard  
2047 PDRs have changed, and another that indicates whether any OEM PDRs (if any) have changed.

2048 See 25.5 for more information about accessing PDRs. Table 57 describes the format of this command.

2049

Table 57 – GetPDRRepositoryInfo command format

Type	Request data
–	none
Type	Response data
enum8	<p><b>completionCode</b></p> <p>value: { PLDM_BASE_CODES }</p>
enum8	<p><b>repositoryState</b></p> <p>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.  }</p>
timestamp104	<p><b>updateTime</b></p> <p>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".</p>
timestamp104	<p><b>OEMUpdateTime</b></p> <p>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.</p>
uint32	<p><b>recordCount</b></p> <p>Total number of PDRs in this repository</p>
uint32	<p><b>repositorySize</b></p> <p>Size of the PDR Repository in bytes. This value provides information that can be used for helping estimate buffer size requirements when accessing PDRs.</p> <p>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.</p> <p>An implementation is allowed to round this number up to the nearest kilobyte (1024 bytes).</p>
uint32	<p><b>largestRecordSize</b></p> <p>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.</p> <p>An implementation is allowed to round this number of up to the nearest 64-byte increment.</p>
uint8	<p><b>dataTransferHandleTimeout</b></p> <p>The minimum interval, in seconds, that a dataTransferHandle value remains valid after it was delivered in the response of a GetPDR or FindPDR command.</p> <p>special values: { 0x00 = no timeout, 0x01 = default minimum timeout (<b>MC1</b>, see clause 29), 0xFF = timeout &gt;254 seconds. Any timeout values that are less than the specified default minimum timeout are illegal. }</p>

2050 **26.2 GetPDR command**

2051 The GetPDR command is used to retrieve individual PDRs from a PDR Repository. The record is  
 2052 identified by the PDR recordHandle value that is passed in the request. The command can also be used  
 2053 to dump all the PDRs within a PDR Repository.

2054 **26.2.1 GetPDR command format**

2055 Table 58 describes the format of the GetPDR command.

2056 **Table 58 – GetPDR command format**

Type	Request data
uint32	<b>recordHandle</b> 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	<b>dataTransferHandle</b> 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	<b>transferOperationFlag</b> Indicates whether this request is for the first portion of the PDR value: { GetNextPart = 0x00, GetFirstPart = 0x01}
uint16	<b>requestCount</b> 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.
uint16	<b>recordChangeNumber</b> 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	<b>completionCode</b> 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	<b>nextRecordHandle</b> 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	<b>nextDataTransferHandle</b> 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. }

Type	Request data
enum8	<p><b>transferFlag</b></p> <p>Indicates what portion of the PDR is being transferred</p> <p>value: {Start = 0x00, Middle = 0x01, End = 0x04, StartAndEnd = 0x05}</p>
uint16	<p><b>responseCount</b></p> <p>The number of recordData bytes returned in this response</p> <p>special value: { returns 0x0000 if the requestCount was 0x0000 }</p>
(var)	<p><b>recordData</b></p> <p>PDR data bytes. This field is absent if responseCount = 0x0000. The number of PDR data bytes returned in this field must match responseCount.</p>
<i>If transferFlag = End</i>	
uint8	<p><b>transferCRC</b></p> <p>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 <math>x^8 + x^2 + x^1 + 1</math> (This is the same polynomial used in the MCTP over SMBus/I<sup>2</sup>C transport binding specification). The CRC is calculated from most-significant bit to least-significant bit on bytes in the order that they are received. This field is only present when transferFlag = End.</p>

## 2057 26.2.2 Single-part and multipart transfers

2058 The data from a given PDR may be accessed using a single-part or multipart transfer. A single transfer  
 2059 occurs when the entire PDR content is delivered using a single GetPDR command response. A multipart  
 2060 transfer is required when the record data exceeds either the amount of data that the responder can return  
 2061 using a single response, or when it exceeds the amount of data that the requester can accept in a single  
 2062 response. In this case, the GetPDR command is used iteratively to retrieve the first portion of the record  
 2063 and then subsequent portions. Additional information and requirements for multipart transfers is provided  
 2064 in 26.2.7.

2065 Partial transfers from the beginning of a record are allowed. That is, a requester is not required to read  
 2066 out an entire record if only the beginning portion of the record data is of interest.

## 2067 26.2.3 PDR recordHandle

2068 The recordHandle is an opaque value that is used by the implementation of the PDR Repository to  
 2069 identify individual records and to track where the next data of a multipart transfer will come from. This  
 2070 value is obtained from the response data of a previous instance of the GetPDR command. A special  
 2071 value of 0x0000\_0000 is used to retrieve the first PDR in the repository.

2072 Some implementations may use the recordHandle as a direct offset into storage memory, others may use  
 2073 it as offset that is relative to the start of the PDR data, and others may use it as a table or list index.

## 2074 26.2.4 PDR recordHandle retention

2075 The recordHandle values that are used to access a particular PDR may change when the  
 2076 recordChangeNumber is changed. recordHandle values are also not guaranteed to endure across  
 2077 connections to the given PLDM terminus that is implementing the command. A party that needs to re-  
 2078 establish a connection to the terminus must assume that any PDR recordHandle values that it previously  
 2079 had are no longer valid. If any multipart transfers were not completed before the connection was re-  
 2080 established, those transfers must be restarted from the beginning.

**2081 26.2.5 PDR recordChangeNumber**

2082 The recordChangeNumber provides a mechanism for preventing the use of invalid PDR data if a record's  
2083 data gets updated while the record was in the process of being read out. The mechanism helps ensure  
2084 that a requester does not get the first parts from an earlier version of the record and remaining parts from  
2085 a later version of the record. The recordChangeNumber can also be used to help a requester scan and  
2086 identify which PDRs may have changed after an update to the PDR Repository has occurred.

2087 To accomplish this, the PDR recordChangeNumber that is returned in the GetPDR response is required  
2088 to change whenever the data of a PDR changes during a multipart access of the PDR. The party that is  
2089 accessing a PDR gets the recordChangeNumber when the first part of the record is returned. This  
2090 number is then used as one of the input parameters when retrieving the remaining parts of the record.

2091 The PLDM responder compares this number against the present recordChangeNumber that is associated  
2092 with the record. If there is a mismatch, the PLDM responder returns an error completionCode. The  
2093 requester can then handle the error by starting the PDR transfer over.

2094 It is recommended that an implementation update the recordChangeNumber only for records that have  
2095 changed due to an update. However, implementations may elect to update the recordChangeNumber for  
2096 some or all unchanged records. This latter approach can be used for small and simple implementations in  
2097 which PDR exits and updates are rare, but should be avoided in large implementations in which the party  
2098 that is accessing the PDR data may see significant delays due to the unnecessary re-reading and  
2099 handling of PDRs that have not actually changed.

**2100 26.2.6 PDR Repository time stamp and PDR Repository locking**

2101 The recordChangeNumber mechanism protects against inconsistent data only on a per record basis; it  
2102 does not automatically protect against inconsistencies that may occur due to individual updates of  
2103 interrelated records. For example, if record A and B are interrelated and both need synchronized updates,  
2104 it is possible that a party could access the records at a time when A has been updated but B has not. The  
2105 individual records would be correct, but their interrelationship could be incorrect.

2106 The party that is updating the PDRs can lock the repository while updates are occurring (the mechanisms  
2107 used for updating and locking the PDRs are outside this specification). In this case, commands such as  
2108 the GetPDR command will return an error completionCode indicating that the repository records are  
2109 inaccessible because an update is in progress. Update-in-progress status is also available in the  
2110 GetPDRRepositoryInfo command.

2111 A party that updates records in a PDR Repository while PLDM command handling is active must either  
2112 lock the PDRs and update the time stamp and recordChangeNumber values before making the repository  
2113 available, or it must update the time stamp and recordChangeNumber values as each individual updated  
2114 record is made available through PLDM.

2115 The PDR Repository has a time stamp that can be read using the GetPDRRepositoryInfo command. The  
2116 time stamp value is updated whenever changes are made to the repository. A party that is accessing  
2117 multiple PDRs and relying on an interrelationship between those records should check the time stamp  
2118 value after retrieving the records to verify that a repository update did not occur while the records were  
2119 being accessed.

2120 If an update has occurred while records were being read, the records should either be re-read or their  
2121 recordChangeNumber values checked to see if they have changed. Because the recordChangeNumber  
2122 is in the beginning portion of a PDR, it is not necessary to read the entire record to get the value.

**2123 26.2.7 Multipart PDR transfers**

2124 The command is intended to support multipart transfer of PDR data only in a sequential manner, starting  
2125 from the beginning of the PDR. Random access to a middle portion of a PDR is not required by  
2126 implementations, nor is it intentionally supported as an option in this specification.

2127 The dataTransferHandle value is therefore required to remain valid only for use with the next GetNextPart  
2128 operation from a given requester. Although many implementations will likely return the same data for an  
2129 identical sequence of PDR access commands regardless of the ID of the requester, an implementation  
2130 may allocate and track dataTransferHandles on a per-requester basis. The dataTransferHandle  
2131 information given to one requester might not be usable by another requester.

### 2132 **26.2.8 PDR dataTransferHandle retention**

2133 The dataTransferHandle value for a multipart transfer is required to remain valid for at least MC1 seconds  
2134 after it has been delivered in a response. After this interval, an implementation may elect to implement a  
2135 timeout and terminate the multipart transfer. To support this, an implementation would use some aspect  
2136 of the recordHandle value to track the particular multipart transfer in progress.

2137 The provisions that allow a dataTransferHandle value to become invalid or expire allow implementations  
2138 the option of temporarily queuing PDR data in memory and freeing up that memory if the record data is  
2139 no longer being accessed. The provisions eliminate the need for the recordHandle values for a given  
2140 request to remain valid indefinitely.

### 2141 **26.2.9 Multipart PDR transfer termination and timeouts**

2142 No formal release mechanism exists for multipart PDR transfers. Multipart transfers may be terminated by  
2143 the responder under the following conditions:

- 2144 • The responder implementation may restrict a given requester to having only one PDR transfer  
2145 in process at a time. If the requester starts a different transfer, the earlier multipart transfer that  
2146 was in progress may be aborted.
- 2147 • The responder implementation may terminate any multipart PDR transfer in progress following  
2148 expiration of the PDR dataTransferHandle retention interval, MC1.
- 2149 • Execution of the Initialization Agent function may terminate a multipart PDR transfer in progress.

### 2150 **26.2.10 Reuse of prior request values**

2151 Except for the first part of a PDR, an implementation is not required to support returning a previously  
2152 transferred portion of a PDR after the transfer has progressed to a later portion. For example, if the first  
2153 three portions of a PDR have been transferred, the implementation may not allow a re-transfer of the  
2154 second portion without restarting the transfer from the beginning. If an implementation does accept  
2155 request parameters that were used for reading an earlier portion of a given PDR, it must return the same  
2156 PDR data that was returned for the original request.

## 2157 **26.3 FindPDR command**

2158 The FindPDR command is provided to improve the efficiency of common types of access to a Primary  
2159 PDR Repository. The FindPDR command is primarily designed to provide operations that can assist a  
2160 MAP in using information from the PDRs to instantiate CIM objects and associations.

2161 The FindPDR command returns the PLDMHandleType and PLDMHandle values for a particular PDR or  
2162 set of PDRs, depending on the parameters that were passed in the request. The response can also  
2163 include the first portion of the PDR data. The response from the FindPDR command can then be used  
2164 with the GetPDR command to read the PDR or the remaining portions of the PDR.

2165 To reduce implementation and validation complexity, the FindPDR command does not provide a generic  
2166 search engine but supports only a limited number of different preconfigured queries that are restricted to  
2167 using particular key fields within the PDRs.

2168 For example, the FindPDR command can be used to find all the PDRs that have a particular  
2169 PLDMTerminusHandle, or Entity Association PDRs that have a common Container ID. It can also be used

2170 to find Numeric Sensor PDRs that share a particular type of monitored numeric unit, such as temperature,  
 2171 or state sensors that use a particular state set. However, the FindPDR command does not support less  
 2172 common operations such as finding records that have a particular hysteresis value setting or state  
 2173 sensors that implement a particular state from within a state set.

2174 The findParameters field holds the PDRTYPE-specific search fields. The format of findParameters is  
 2175 identified by the parameterFormatNumber that is passed in the request. The findParameters value may  
 2176 be applicable to more than one PDRTYPE. The parameterFormatNumber and PDRTYPE field in the  
 2177 request are used together to identify which PDRs should be searched. Table 60 lists the values for  
 2178 parameterFormatNumber and the PDRTYPE values that are associated with each  
 2179 parameterFormatNumber. Table 61 lists the different PDR fields that make up the findParameters value  
 2180 for each different parameterFormatNumber.

2181 If the PDRTYPE field value is set to 0, all of the PDRTYPE values that are specified for the  
 2182 parameterFormatNumber in Table 60 are searched. Otherwise, only PDRs that have the given PDRTYPE  
 2183 value are searched.

2184 For example, if PDRTYPE = 0 and parameterFormatNumber = 7, all PDRs with PDRTYPE values that are  
 2185 identified for searching with parameterFormatNumber = 7 are searched: Numeric Effector Initialization,  
 2186 State Effector Initialization, and Effector Auxiliary Names. If the PDRTYPE is set to the value for State  
 2187 Effector Initialization PDR, only State Effector Initialization PDRs are searched.

2188 The findParameters value is included in each request to eliminate the need for implementations to retain  
 2189 the findParameters value when a multi-PDR find operation is being done.

2190 Table 59 describes the format of this command.

2191

**Table 59 – FindPDR command format**

Type	Request data
uint32	<p><b>findHandle</b></p> <p>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.</p> <p>special values: { use 0x0000_0000 if the findOperation is findFirst,                      0xFFFF_FFFF = reserved. }</p> <p>NOTE: This field has the same retention specifications as the dataTransferHandle field used in the GetPDR command. See 26.2.4 for more information.</p>
enum8	<p><b>findOperationFlag</b></p> <p>Indicates whether this request is for locating the first matching PDR.</p> <p>value: { findNext = 0x00, findFirst = 0x01 }</p>
uint16	<p><b>requestCount</b></p> <p>The maximum number of record bytes requested to be returned in the response to this instance of the FindPDR command.</p> <p>NOTE: The responder may return fewer bytes than were requested.</p>
uint16	<p><b>PDRTYPE</b></p> <p>The PDRTYPE for the records to be located.</p> <p>special value: 0x0000 = match any PDRTYPE.</p>

Type	Request data
uint8	<p><b>parameterFormatNumber</b></p> <p>A number that identifies the format and number of parameters in the findParameters field. Table 61 lists the different PDR fields that make up the findParameters value for each different parameterFormatNumber.</p>
bitfield16	<p><b>wildcards</b></p> <p>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 any bit position for which a parameter is not defined.</p> <p>[15] – 1b = sixteenth parameter value in findParameters must be matched                      0b = sixteenth parameter value in findParameters is ignored</p> <p>...</p> <p>[0] – 1b = first parameter value in findParameters must be matched                      0b = first parameter value in findParameters is ignored</p>
varies	<p><b>findParameters</b></p> <p>A series of parameters that correspond to fields in the PDRs that are used for the find operation.</p> <p>Table 61 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 61, 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 60.</p> <p>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.</p>
Type	Response data
enum8	<p><b>completionCode</b></p> <p>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                      }</p>
uint32	<p><b>nextFindHandle</b></p> <p>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.</p> <p>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.}</p>

Type	Response data
uint32	<p><b>nextDataTransferHandle</b></p> <p>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.</p> <p>special value: { returns 0x0000_0000 if there is no remaining recordData beyond the recordData that is being returned in this response data. }</p>
enum8	<p><b>transferFlag</b></p> <p>Indicates what portion of the PDR is being transferred</p> <p>value: {Start = 0x00, Middle = 0x01, End = 0x04, StartAndEnd = 0x05}</p>
uint16	<p><b>responseCount</b></p> <p>The number of recordData bytes returned in this response</p> <p>special value: { returns 0x0000 if the requestCount was 0x0000 }</p>
(var)	<p><b>recordData</b></p> <p>PDR data bytes. This field is absent if responseCount = 0x0000. Otherwise, the number of PDR data bytes returned in this field must match responseCount.</p>

2192

**Table 60 – FindPDR Command Parameter Format Numbers**

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

2193 <sup>[1]</sup> The entire contents of the repository can be read by using this format along with PDRType = ANY and PLDMTerminusHandle set  
 2194 for "wildcard."

2195 <sup>[2]</sup> The PLDMTerminusHandle parameter must be set for "wildcard" when using this format to search for Event Log PDRs.

2196 <sup>[3]</sup> This search format can be used to return all PDRs that have any of the indicated "OEM" PDRType values or all PDRs that have  
 2197 any of the indicated "OEM" PDRType values and match a particular vendorIANA.

2198

Table 61 – 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			•	•	•					•					
effectorID						•	•	•							
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														•	

2199 **26.4 RunInitAgent command**

2200 The RunInitAgent command directs the terminus that provides the Primary PDR Repository to run the  
 2201 Initialization Agent function. This command can be used to trigger a re-initialization of the monitoring and  
 2202 control capabilities in the PLDM subsystem. Table 62 describes the format of the command.

2203 **Table 62 – RunInitAgent command format**

Type	Request data
bitfield8	<p><b>initConditionEmulation</b></p> <p>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.</p> <p>value: {</p> <p>    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.</p> <p>    0x01 = PLDMSubsystemPowerUp,         // Directs the Initialization Agent to take the same steps              // as it would when the PLDM subsystem becomes              // powered up.</p> <p>    0x02 = SystemHardReset,               // Directs the Initialization Agent to take the same steps              // as it would following a system hard reset.</p> <p>    0x03 = SystemWarmReset,              // Directs the Initialization Agent to take the same steps              // as it would following a system warm reset.</p> <p>    0x04 = PLDMTerminusOnline            // Directs the Initialization Agent to initialize the              // terminus that has a TID that matches the TID              // parameter in this request.</p> <p>}</p>
bitfield8	<p><b>TID</b></p> <p>Terminus ID for the terminus to be initialized when the initConditionEmulation field in this request is set to PLDMTerminusOnline.</p> <p>special value: The value in this field is ignored when the initConditionEmulation field in this request is set to any value other than PLDMTerminusOnline.</p>
Type	Response data
enum8	<p><b>completionCode</b></p> <p>value: { PLDM_BASE_CODES }</p>

2204 **27 PDR definitions**

2205 This clause describes certain important characteristic parameters that are provided within the PDRs for  
 2206 interpreting the readings and settings of sensors and effecters.

## 2207 **27.1 Sensor types**

2208 PLDM contains two basic types of sensors that are described using PDRs:

- 2209 • The PLDM Numeric Sensor is used to obtain a numeric value for a monitored parameter. The  
2210 sensor definition also optionally includes returning state information based on whether the  
2211 numeric reading has crossed one or more defined threshold levels.
- 2212 • The PLDM State Sensor/PLDM Composite State Sensor is used to obtain the present state of a  
2213 monitored parameter. The PLDM sensor access commands allow an implementation to provide  
2214 multiple sets of state information using a single access command. When this is done, the  
2215 implementation is referred to as providing a Composite State Sensor.

## 2216 **27.2 Effector types**

2217 PLDM contains two basic types of effectors that are described using PDRs:

- 2218 • The PLDM Numeric Effector is used to set a numeric value for a monitored parameter. The  
2219 effector definition also optionally includes returning state information based on whether the  
2220 numeric reading has crossed one or more defined threshold levels.
- 2221 • The PLDM State Effector/PLDM Composite State Effector is used to set the present state of a  
2222 monitored parameter. The PLDM effector access commands allow an implementation to provide  
2223 multiple sets of state information using a single access command. When this is done, the  
2224 implementation is referred to as providing a Composite State Effector.

## 2225 **27.3 State sets**

2226 State information is returned using an enumeration called a “state set.” Each state set has a different ID  
2227 number. This number is used within the PDRs to identify what particular state set a sensor or effector is  
2228 using. See clause 24 for more information.

## 2229 **27.4 Sensor and effector units**

2230 This subclause and following subclauses describe the fields that are used within PDRs to define and  
2231 describe sensor and effector units and related characteristics such as accuracy, tolerance, and resolution.

2232 The type of units that are associated with the value that a sensor returns or monitors, or that an effector  
2233 controls, such as volts or amps, is identified in the PDRs by a sensorUnits enumeration, listed in Table  
2234 63. Unless otherwise indicated, the units apply to all numeric properties of the sensor, such as the sensor  
2235 reading, threshold values, and resolution.

2236 Vendor defined units are identified by a special value for OEMUnit. A special PDR called the OEM Unit  
2237 PDR is used to define the meaning of the OEMUnit when it is used in the PDRs that describe a sensor or  
2238 effector. Refer to 28.9 for more information about how OEMUnits are used in PDRs.

2239

**Table 63 – 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

### 2240 27.4.1 Base units

2241 The base unit of measurement that is associated with the reading values returned by a PLDM Numeric  
 2242 Sensor or set into a PLDM Numeric Effector is represented by the combination of three fields from the  
 2243 PDR for the sensor: baseUnits, unitModifier, and rateUnits. These fields are interpreted according to the  
 2244 following formula:

$$2245 \quad \text{Sensor/Effector Units} = \text{baseUnit} * 10^{\text{unitModifier}} \text{rateUnit}$$

2246 For example, if baseUnits is Volts and the unitModifier is -6, the units of the values returned are  
 2247 microvolts.

2248 If the rateUnits property is set to a value other than None, the units are further qualified as rate units. In  
 2249 the preceding example, if rateUnits is set to Per Second, the values returned by the sensor are in  
 2250 microvolts/second.

### 2251 27.4.2 Auxiliary units

2252 In some cases, additional modification of the base unit of the sensor might be required. For example,  
 2253 acceleration is commonly given in units such as "meters per second per second". The PDRs include a  
 2254 provision for modifying the base units with an additional set of units called auxiliary units. Auxiliary units  
 2255 are defined by three elements: auxUnit, auxUnitModifier, and auxRateUnit. These elements are used in  
 2256 combination with the base units as follows:

$$2257 \quad \text{Sensor/Effector Units} = \text{baseUnit} * 10^{\text{unitModifier}} [\text{rel}] \text{auxUnit} * 10^{\text{auxUnitModifier}} \text{rateUnit auxRateUnit}$$

2258 [rel] is the relationship between the base unit and the auxiliary unit, as follows:

2259 rel = enum8 { dividedBy, multipliedBy }

2260 And:

2261 dividedBy implies a "/" or "per" relationship, such as "per foot"

2262 multipliedBy implies a "\*" operation, such as "foot\*lbs (foot-lbs)"

2263 auxUnit and auxRateUnit shall not be used if an equivalent definition can be made using only base units.

### 2264 27.4.3 Units for use with CIM

2265 Developers are cautioned that PLDM units may include types of units that are not presently supported by  
 2266 standard CIM objects such as CIM\_Sensor. PLDM supports additional types of units because certain  
 2267 types of sensors or effectors may be used within a platform management subsystem but are not exposed  
 2268 through CIM, or are mapped into CIM using proprietary CIM extensions. Parties developing platform  
 2269 management subsystems in which sensors are intended to be exposed as CIM objects should first verify  
 2270 which types of sensors and units are supported by CIM and the CIM profiles.

### 2271 27.4.4 OEM (Vendor-Defined) sensor units

2272 OEM (vendor-defined) sensor units are identified in PLDM sensor PDRs when the OEMUnit value from  
 2273 Table 63 is used for the baseUnit or auxUnit. The semantic information of an OEMUnit can then be  
 2274 further described using an OEM Sensor Units PDR that is associated with the particular sensor that is  
 2275 returning the OEMUnit. Multiple OEM Sensor Units PDRs can be defined if there is a need for defining  
 2276 more than one type of OEM unit. Additionally, multiple PLDM Sensor PDRs can be associated with a  
 2277 particular OEM Sensor Units PDR.

2278 **27.5 Counters**

2279 A counter is a numeric sensor that returns a value that returns a count. PLDM does not define any  
 2280 requirements on whether a counter must increment, decrement, or both, or whether it does so  
 2281 sequentially or monotonically, and so on.

2282 Many common types of counters can use predefined sensor unit values, such as Hits, Misses, Corrected  
 2283 Errors, Uncorrected Errors, and others. If no predefined unit fits, it is recommended that the auxiliary  
 2284 sensor unit (auxUnit) be designated using the predefined unit "Counts" in the PDR for the sensor, and  
 2285 that an OEM unit type is defined for the base unit.

2286 For example, if an implementation needed a counter for "widgets," it would be noted that no predefined  
 2287 sensor unit type for "widgets" exists. In this case, an OEM Unit PDR for "widgets" is created and used for  
 2288 the base unit type, and "Counts" is used as the auxUnit.

2289 Counters enable a party that accesses PDR information for the sensor to get a partial interpretation of the  
 2290 sensor semantics. Thus, although the party interpreting the sensor may not know what a widget is, it will  
 2291 know that the sensor is returning Counts of something.

2292 **27.6 Accuracy, tolerance, resolution, and offset**

2293 The PDRs for numeric sensors and effecters include fields for reporting the accuracy, tolerance, and  
 2294 resolution associated with the numeric value for the reading or setting. This subclause provides  
 2295 definitions for accuracy, tolerance, and resolution as used within this specification and information on how  
 2296 the values are calculated and used. Accuracy, tolerance, and resolution are summarized as follows:

2297 **Accuracy** An error in the reading that scales proportionally with the magnitude of the input. Typically  
 2298 given as a  $\pm$  percentage of the reading.

2299 **Tolerance** A  $\pm$  error in the reading that, unlike accuracy, does not scale with the magnitude of the  
 2300 reading. Tolerance typically comes from a combination of quantization (round off) errors  
 2301 including errors due to offsets in the measurement.

2302 **Resolution** The nominal size of the "steps" between sequential reading values.

2303 Accuracy specifies a degree of error that varies in proportion to the reading, and tolerance specifies a  
 2304 constant error. The combination of these two generally provides enough flexibility to cover a range of  
 2305 conversion errors in most linear analog-to-digital (A/D) converters.

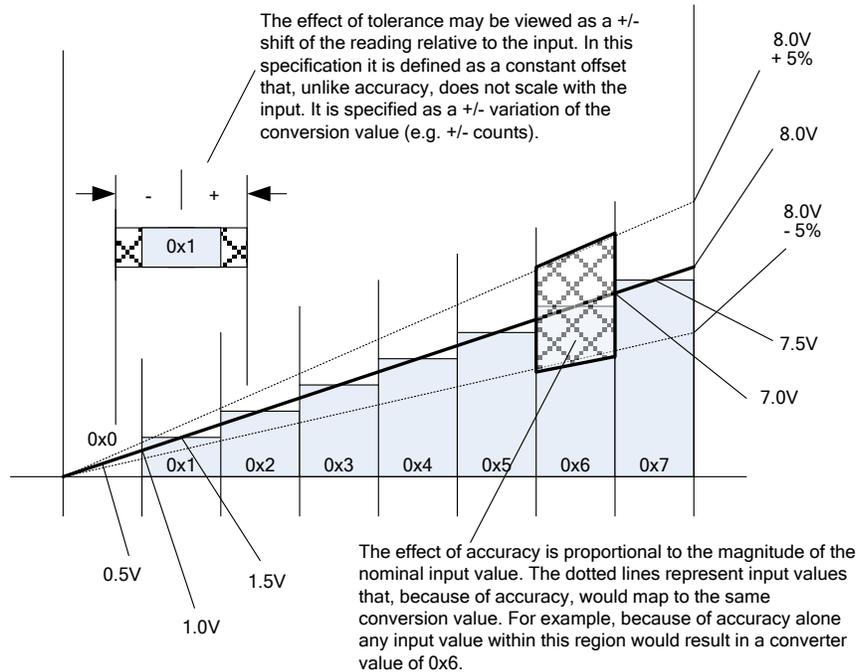
2306 Although other error types, such as non-linearity, can exist in converters, the contribution of those errors  
 2307 can be accounted for by increasing the size of the reported values for tolerance, accuracy, or both as  
 2308 necessary.

2309 **27.6.1 Additional information about numeric sensor / effector tolerance**

2310 Tolerance can be considered to be a constant portion of the quantization error in the conversion of an  
 2311 analog input to a numeric sensor. Consider a sensor where 0x00 ideally corresponds to 0.000 to 0.500 V  
 2312 and 0x01 corresponds to 0.500 V to 1.000 V. When the input is 0.500 V exactly, the sensor could either  
 2313 report 0x00 or 0x01. Now assume that the input is 0.501 V. Ideally, this would result in a value of 0x01  
 2314 from the sensor, but because of offsets in an implementation, it is possible that some implementations  
 2315 could return either a value of 0x00 or 0x01. If 0x00 is reported, the sensor is effectively returning a value  
 2316 that is -1 count from ideal. It is possible that the sensor implementation could be asymmetric with respect  
 2317 to tolerance. For example, a sensor implementation may sometimes map 0.501 V to 0x00, but would  
 2318 never map anything less than 0.500 V to 0x01. In this case, the tolerance would be +0 counts and -1  
 2319 counts. Generally, an implementation is subject to both positive and negative offsets because of  
 2320 component manufacturing variation, noise, and so on. Thus, it is common to see a tolerance of  $\pm$  1 count.

2321 **27.6.2 Examples of accuracy, tolerance, and resolution use**

2322 Figure 23 shows an example of a "3-bit" (eight step) converter. In this example, the converter is hooked  
 2323 up for monitoring a nominal signal that can vary from 0.0 V to 8.0 V. The resolution is defined as the size  
 2324 of the steps between nominal readings. The resolution is 1.0 V because there is 1.0 V difference between  
 2325 each successive reading value.



2326

2327 **Figure 23 – Accuracy, tolerance, and resolution example**

2328 In this example, the input value that corresponds to a reading of 0x0 is actually centered around 0.50 V,  
 2329 not 0.0 V. That is, the meaning of a reading of 0x0 does not mean 0.0 V, as might be expected, but  
 2330 actually means "0.5 V plus or minus 0.5 V". This represents a typical way that A/D converters are  
 2331 connected in systems. It is a common mistake to assume that a reading of zero actually corresponds to  
 2332 0.0 V.

2333 If this converter had no additional offsets or accuracy errors, the reading values would correspond to input  
 2334 values as follows:

- 2335 0x0 → 0 V to 1.0 V (0.5 V ± 0.5 V)
- 2336 0x1 → 1.0 V to 2.0 V (1.5 V ± 0.5 V)
- 2337 0x2 → 2.0 V to 3.0 V (2.5 V ± 0.5 V)
- 2338 0x3 → 3.0 V to 4.0 V (3.5 V ± 0.5 V)
- 2339 0x4 → 4.0 V to 5.0 V (4.5 V ± 0.5 V)
- 2340 0x5 → 5.0 V to 6.0 V (5.5 V ± 0.5 V)

2341           0x6 → 6.0 V to 7.0 V (6.5 V ± 0.5 V)

2342           0x7 → 7.0 V to 8.0 V (7.5 V ± 0.5 V)

2343 If these readings were converted to their corresponding nominal input voltage ( $V_{in}$ ) values, the formula  
2344 would be as follows:

2345            $V_{in}(\text{nominal}) \rightarrow (\text{resolution} * \text{reading}) + 1/2 \text{ resolution}$

2346 Note that this follows the Cartesian coordinate formula for a line:  $y = Mx + B$

2347 Now, suppose that the implementation could add a negative D.C. offset of 0.5 V to the input. Then the  
2348 center point for a reading of 0.0 V would correspond to 0.0 V, and a reading of 0x0 would correspond to a  
2349 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  $V = (\text{resolution}$   
2350  $* \text{reading}) + 0.0 \text{ V}$ . There is now no offset relative to the center of the reading value because of a D.C.  
2351 offset. If the converted negative offset of 4.0 V was connected to the input, a reading of 0x0 would now  
2352 correspond to -3.5 V ± 0.5 V and a reading of 111b would correspond to 3.5 V ± 0.5 V.

2353 It is very common for an A/D converter implementation to have a D.C. offset that needs to be accounted  
2354 for when converting a reading to the corresponding nominal input value. The party that implements the  
2355 hardware for the sensor needs to provide this offset value as well as the resolution (step size per count)  
2356 so that the basic conversion of the reading can be accomplished.

2357 After the basic conversion of the reading is done, the effects of accuracy and tolerance may need to be  
2358 taken into account. For example, if someone is depending on the reading to determine whether  
2359 something has failed, it is important to understand how much error might be in the reading so that a  
2360 failure is not falsely assessed for a healthy component.

2361 For PLDM, the effects of accuracy and tolerance are considered to be orthogonal to one another and  
2362 additive. First consider the effect of accuracy. Suppose the accuracy of the sensor is specified as ±5%.  
2363 Using that figure, a value of 001b will nominally correspond to 1.5 V ± 5%, but because of quantization  
2364 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  
2365 reading of 0x1.

2366 The next step is to factor in tolerance. The quantization within a converter is never perfect; some slight  
2367 variation always exists in the comparison points that yield a particular converter output. Instead of the  
2368 conversion ranges being evenly spaced as shown in Figure 23, some ranges may be a little wider and  
2369 others a little narrower. The effect of this is that in an actual implementation, borderline values such as  
2370 1.99 V or 2.01 V, for example, may sometimes yield a value of 0x1 and sometimes 0x2.

2371 Tolerance in PLDM is defined as an error in the quantization that is applied to all counts of the converter  
2372 equally. Because PLDM sensors are all specified as returning integer values, any errors in the reading  
2373 will always result in an integral number of counts. Thus, tolerance is specified as a +/- effect on the count.

2374 The tolerance value is typically used to account for quantization errors in A/D conversion circuitry that  
2375 occur because of effects such as D.C. voltage offsets within the circuit. For example, suppose the input to  
2376 an A/D converter that monitors voltage was shifted up by a constant amount, as would be the case if a  
2377 D.C. offset was added to the input. Per the figure, if a D.C. offset error of 0.25 V were added when  
2378 converting, the input reading 0x01 would represent a range that actually goes from 0.75 V to 1.75 V  
2379 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  
2380 cause a reading of 0x1 to be returned instead 0x0. Thus, because of this offset error, the reading would  
2381 be one count higher than it was intended to be for inputs in that range. Similarly, with the same offset, a  
2382 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  
2383 2.00 V would also result in a reading that is one count higher than intended.

2384 This does not mean that all conversions are off by one count. In this example, the reading is incorrect  
2385 only for inputs that are in the range caused by the offset. A reading of 0x1 would be correctly returned for

2386 an input of 1.5 V. The reading can thus be incorrect by 0 counts or +1 counts depending on what range  
2387 the input value is in. In this case, the tolerance would be specified as +1/-0 counts.

2388 Manufacturing variations and tolerances in A/D conversion circuitry mean that both positive and negative  
2389 offsets are possible. This is why it is typical to see a specification of  $\pm 1$  count for tolerance. In many  
2390 implementations, tolerance is specified as  $\pm 1$  count for these types of conversions. Because resolution is  
2391 given in units of 1 count, tolerance and resolution may sometimes appear to equate to the same value.  
2392 However, tolerance and resolution should not be misinterpreted as being the same thing.

2393 Lastly, in some cases PLDM Numeric Sensors will return values such as counts or other measurements  
2394 that do not use a conversion process that can introduce errors in the reading. In this case, the tolerance is  
2395 specified as  $\pm 0$  counts.

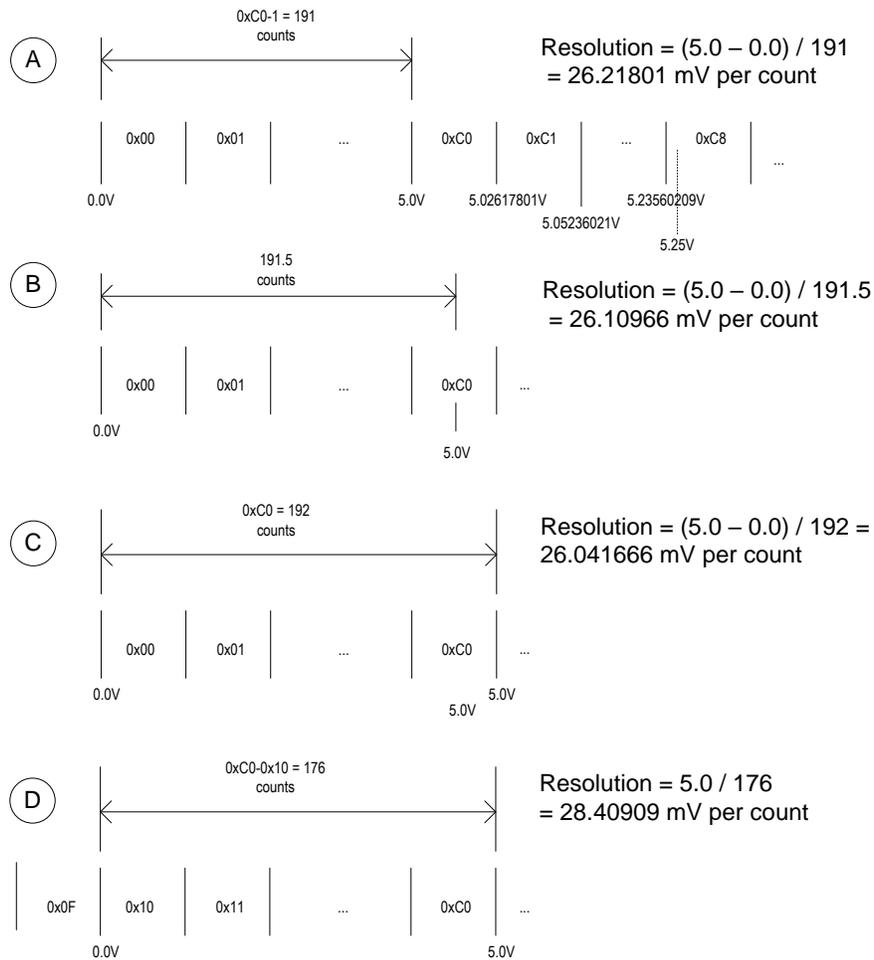
### 2396 **27.6.3 Accuracy, tolerance, and resolution relationship to thresholds**

2397 Accuracy, tolerance, and resolution must all be taken into account to generate a threshold that does not  
2398 generate a "false positive" (a false indication of a failure). For example, if accuracy, tolerance, and  
2399 resolution are not taken into account when calculating the threshold for a warning level, it is possible that  
2400 an input could be assessed as being within the warning range when the input was actually near the limit  
2401 of the normal range.

2402 A consequence of avoiding false positives is that for a particular range a value that is actually within the  
2403 intended warning range can be assessed as being within the normal range. That is, false positives are  
2404 avoided at the cost of having the possibility of 'false negatives'. However, in most implementations it is  
2405 considered better to avoid the false alarms that false positives would cause. Whether to design thresholds  
2406 to avoid false positives or false negatives is a choice of the system implementation.

2407 Because it is the more common case, the following examples describe how thresholds may be calculated  
2408 to avoid false positives.

2409 EXAMPLE: An 8-bit A/D converter monitoring a 5.0 V nominal signal where the sensor has been designed such  
2410 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  
2411 Figure 24A). Assume the converter implementation has a specified worst-case accuracy of  $\pm 4\%$ , and a tolerance of  $\pm$   
2412 1 count.



2413

2414

**Figure 24 – Figuring resolution from the design**

2415 For Figure 24A, this yields resolution, tolerance, and accuracy values as follows:

2416 Resolution

2417  $= 5.0 \text{ V} / (\text{C0h} - 1) = 26.17801 \text{ mV}$

2418 Accuracy

2419  $= \pm 4\%$  (given, from the design)

2420 Tolerance

2421  $= \pm 1 \text{ count (given)} = \pm 26.17801 \text{ mV}$

2422 Now, suppose it is necessary to calculate an upper critical threshold for the 5.0 V + 5% point (5.25 V)

2423 where this threshold will not produce "false positives" (falsely return 'critical') across the range of

2424 accuracy, tolerance, and resolution. The following example shows steps that can be used to calculate a

2425 threshold suitable for a PLDM Numeric Sensor:

2426 Step 1: Divide the target threshold value by the resolution to find how many counts correspond to  
2427 5.25 V:

2428  $5.25 \text{ V} / 26.17801 \text{ mV} = 200.55 \text{ counts}$   
2429 (which puts the 5.25 V point within the nominal range of reading 0xC8, as shown in  
2430 Figure 24A)

2431 Step 2: Factor in the tolerance:

2432 **Important:** Because tolerance is specified as an error, a "+" count for tolerance means that  
2433 the reading may be higher than it should be, and a "-" count means that the reading may be  
2434 lower than it should be. To account for these errors, the "-" tolerance value should be added  
2435 to upper thresholds, and the "+" tolerance value subtracted from lower thresholds. This is  
2436 particularly important when the plus and minus tolerance values are different from one  
2437 another.

2438  $200.55 + 1 = 201.55 \text{ counts}$

2439 Step 3: Account for the effect of accuracy:

2440  $201.55 * 1.04 = 209.612 \text{ counts}$

2441 Step 4: Round up (because an A/D converter cannot give a non-integer count)

2442  $209.612 \rightarrow 210 \text{ counts} = 0xD2$

2443 This yields a Threshold value of 210 which corresponds to 5.497 V. This shows that even though a  
2444 threshold of 5.25 V is being targeted, it is necessary to set the threshold to a value that, because of the  
2445 effects of accuracy, tolerance, and resolution, could allow the actual monitored value to be as high as  
2446 5.497 V in some implementations before a threshold match would be detected.

2447 The calculations for lower thresholds are the same, except that negative values for the accuracy,  
2448 tolerance, and resolution are used.

2449 Figure 24 illustrates what to be aware of when deriving the values for resolution from an implementation.  
2450 To get an accurate value for resolution, it is important to know whether the input values that correspond to  
2451 a particular reading are given as values that are at the point of change (quantization point) between  
2452 successive readings, are a nominal "center point" of a reading, or a combination of the two. (The  
2453 difference in the resolution value between Figure 24A and Figure 24C is almost 0.5%. This shows that a  
2454 non-trivial amount of error could be introduced if the implementer uses the wrong calculation point for its  
2455 implementation).

2456 Lastly, area D in Figure 24 shows that offsets in the implementation also need to be taken into account.  
2457 Offset adds a new first step to the threshold calculation:

2458 Step 0: Take the target threshold and subtract (or add, depending on the implementation) the D.C.  
2459 offset value before calculating the counts for the threshold.

## 2460 27.7 Numeric reading conversion formula

2461 The following formula is used with data from the Numeric Sensor PDR to convert the corresponding  
2462 PLDM Numeric Sensor's raw reading to the units specified in the Numeric Sensor PDR.

2463 **Reading Conversion formula:  $Y = (m * X + B)$**

2464 Where:

2465  $Y =$  converted reading in Units

2466  $X =$  reading from sensor

2467  $m =$  resolution from PDR in Units

2468  $B =$  offset from PDR in Units

2469 Units = sensor/effecter Units, based on the Units and auxUnits fields from the PDR for the  
2470 numeric sensor

2471 For example, a sensor with the following units, resolution, offset, and reading:

2472 Reading = 0xBF

2473 Units = Volts

2474 Resolution: 26.17801 mV

2475 Offset = -1.00 V

2476 would have the following the converted reading:

2477  $Y = (26.17801 * 10^{-3} \text{ V} * 0xBF + (-1.00 \text{ V})) = [(0.2617801 * 191) - 1.00] \text{ V} = 4.00 \text{ V}$

2478 A full interpretation of the reading should also take tolerance and accuracy into account. For example, if  
2479 the PDR indicates the following:

2480 Accuracy:  $\pm 4\%$

2481 Tolerance:  $\pm 1$  count (given)

2482 combined with the previous example, the full interpretation of the reading would be:

2483  $(4.00 \text{ V} \pm 26.17801 \text{ mV}) \pm 4\%$

2484 where  $\pm 26.17801 \text{ mV}$  corresponds to the effect of a Tolerance of  $\pm 1$  count.

### 2485 27.7.1 Rounding

2486 Some precision may often be lost in the conversion of binary to decimal. For example, the previous  
2487 conversion that was shown as 4.00 V actually calculates out to 3.99999991 V using the given value for  
2488 the resolution, but the result was rounded up to 4.00. This raises a question about how much rounding  
2489 should be applied, or how many digits of precision should be used for a converted value.

2490 The number of digits of precision for the converted value can be based on the overall size of the binary  
2491 number. For example, an eight-bit unsigned value has a range of 0 to 255, which is three decimal digits.  
2492 Thus, rounding the converted reading to three significant digits is appropriate.

2493 **27.8 Numeric effector conversion formula**

2494 A reverse process from that used to convert a sensor reading is used to generate the raw value to be set  
 2495 into a PLDM Numeric Effector. In this case, the formula is as follows:

2496 **Setting Conversion formula:**  $X = \text{Round} [(Y - B) / m]$

2497 Where:

2498 X = integer setting value for the effector

2499 Y = target setting in Units

2500 m = resolution from PDR in Units

2501 B = offset from PDR in Units

2502 Round = rounding operation to round the value in [ ] to the nearest integer value

2503 Units = sensor/effector Units, based on the Units and auxUnits fields from the Numeric Effector  
 2504 PDR

2505 **28 Platform Descriptor Record (PDR) formats**

2506 This clause defines the content and format of the PDRs that are used for supporting sensor monitoring  
 2507 and control in PLDM.

2508 **28.1 Common PDR header format**

2509 All PDRs have a common, fixed format header followed by variable length record data. The size and  
 2510 definition of the bytes within the PDR data field are specific to each PDR Type. Table 64 describes the  
 2511 format of the common PDR header.

2512 The PDR data length can vary on a per record basis. It is generally recommended that the definition of  
 2513 PDRs of a given type use a fixed length when practical.

2514 The header fields are not shown in the succeeding PDR format subclauses.

2515 **Table 64 – Common PDR header format**

Type	PDR fields
uint32	<p><b>recordHandle</b></p> <p>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.</p> <p>special value: {0x0000_0000 = reserved }</p>
uint8	<p><b>PDRHeaderVersion</b></p> <p>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.</p> <p>value: The value 0x01 shall be used as the PDRHeaderVersion for PDRs that are defined in this specification.</p>
uint8	<p><b>PDRType</b></p> <p>The type of the PDR. See 25.3 and 28.2.</p>

Type	PDR fields
uint16	<b>recordChangeNumber</b> See 26.2.3 for more information.
uint16	<b>dataLength</b> The total number of PDR data bytes following this field.

2516 **28.2 PDR type values**

2517 Table 65 lists the different types of PDRs defined in this document and the corresponding PDR Type  
 2518 values used for those PDRs. Unspecified values are reserved for future definition by this specification.

2519 **Table 65 – 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 Effector PDR	See 28.11.
10	Numeric Effector Initialization PDR	See 28.12.
11	State Effector PDR	See 28.13.
12	State Effector Initialization PDR	See 28.14.
13	Effector Auxiliary Names PDR	See 28.15.
14	Effector 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.
20	FRU Record Set PDR	See 28.22.
126	OEM Device PDR	See 28.23.
127	OEM PDR	See 28.24.

2520 **28.3 Terminus Locator PDR**

2521 The Terminus Locator PDR provides information that associates a PLDMTerminusHandle with values that  
 2522 uniquely identify the device or software that contains the PLDM terminus. Table 66 describes the format  
 2523 of this PDR.

2524 **Table 66 – Terminus Locator PDR format**

Type	Description
–	<b>commonHeader</b> See 28.1.
uint16	<b>PLDMTerminusHandle</b> A handle that identifies PDRs that belong to a particular PLDM terminus.
enum8	<b>validity</b> 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	<b>TID</b> PLDM Terminus ID. This value is used to identify asynchronous messages from a given terminus.
uint16	<b>containerID</b> The containerID for the containing entity that holds this terminus. See 9.1 for more information.
enum8	<b>terminusLocatorType</b> 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	<b>terminusLocatorValueSize</b> 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 persistent and unique for a given terminus in PLDM.
<i>terminusLocatorValue for terminusLocatorType = UID:</i>	
uint8	<b>terminusInstance</b> This field is used to differentiate between different PLDM termini if the device contains more than one PLDM terminus.

Type	Description
UUID	<p><b>deviceUID</b></p> <p>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.</p> <p>A PLDM terminus that is identified using this type of ID must support the GetTerminusUID command.</p>
<i>terminusLocatorValue for terminusLocatorType = MCTP_EID:</i>	
uint8	<p><b>EID</b></p> <p>A MCTP EID that is assigned to an MCTP Endpoint that provides the transport protocol termination for a PLDM terminus</p>
<i>terminusLocatorValue for terminusLocatorType = SMBusRelative</i>	
UUID	<p><b>UID</b></p> <p>A UID for the controller that owns the bus to which the device is connected. For more information, see the preceding description for "<i>terminusLocatorType = UID</i>".</p>
uint8	<p><b>busNumber</b></p> <p>A bus number for the bus to which the device is connected, relative to the controller that owns the bus.</p> <p>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.</p>
uint8	<p><b>slaveAddress</b></p> <p>The SMBus or I<sup>2</sup>C slave address for the device that is providing the</p> <p>[7:1] - SMBus or I<sup>2</sup>C slave address value.</p> <p>[0] - 0b.</p>
<i>terminusLocatorValue for terminusLocatorType = systemSoftware</i>	
enum8	<p><b>softwareClass</b></p> <pre>{     unspecified, other, systemFirmware, OSloader, OS, CIMprovider, otherProvider,     virtualMachineManager }</pre>
UUID	<p><b>UUID</b></p> <p>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.</p>

2525 **28.4 Numeric Sensor PDR**

2526 The Numeric Sensor PDR is primarily used to describe the semantics of a PLDM Numeric Sensor to a  
 2527 party such as a MAP. It also includes the factors that are used for converting raw sensor readings to  
 2528 normalized units. The record also identifies the Entity that is being monitored by the sensor. Table 67  
 2529 describes the format of this PDR. NOTE: The Numeric Sensor PDR sensorID type in this clause has been  
 2530 changed in version 1.1.1 of this specification from uint8 to uint16 to be consistent with GetSensorReading  
 2531 command.

2532

2533

**Table 67 – Numeric Sensor PDR format**

Type	Description
–	<b>commonHeader</b> See 28.1.
uint16	<b>PLDMTerminusHandle</b> A handle that identifies PDRs that belong to a particular PLDM terminus.
uint16	<b>sensorID</b> ID of the sensor relative to the given PLDM Terminus ID.
uint16	<b>entityType</b> The Type value for the entity that is associated with this sensor. See 9.1 for more information.
uint16	<b>entityInstanceNumber</b> The Instance Number for the entity that is associated with this sensor. See 9.1 for more information.
uint16	<b>containerID</b> The containerID for the containing entity that is associated with this sensor. See 9.1 for more information.
enum8	<b>sensorInit</b> Indicates whether the sensor requires initialization by the initializationAgent.  value: { nolnit, // 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 initialize 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	<b>sensorAuxiliaryNamesPDR</b>  true = sensor has a Sensor Auxiliary Names PDR  false = sensor does not have an associated Sensor Auxiliary Names PDR

Type	Description
enum8	<p><b>baseUnit</b></p> <p>The base unit of the reading returned by this sensor. See 27.4 for more information.</p> <p>value: { see Table 63 }</p>
sint8	<p><b>unitModifier</b></p> <p>A power-of-10 multiplier for the baseUnit. See 27.4 for more information.</p>
enum8	<p><b>rateUnit</b></p> <p>value: { None, Per MicroSecond, Per MilliSecond, Per Second, Per Minute, Per Hour, Per Day, Per Week, Per Month, Per Year }</p>
uint8	<p><b>baseOEMUnitHandle</b></p> <p>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.</p>
enum8	<p><b>auxUnit</b></p> <p>The base unit of the reading returned by this sensor. See 27.4 for more information.</p> <p>value: { see Table 63 }</p>
sint8	<p><b>auxUnitModifier</b></p> <p>A power-of-10 multiplier for the auxUnit. See 27.4 for more information.</p>
enum8	<p><b>auxrateUnit</b></p> <p>value: { None, Per MicroSecond, Per MilliSecond, Per Second, Per Minute, Per Hour, Per Day, Per Week, Per Month, Per Year }</p>
enum8	<p><b>rel</b></p> <p>The relationship between the base unit and the auxiliary unit, as follows:</p> <p>value = { dividedBy, multipliedBy}</p> <p>dividedBy implies a "/" or "per" relationship, such as "per foot"</p> <p>multipliedBy implies a "*" operation, such as "foot*lbs (foot-lbs)"</p>
uint8	<p><b>auxOEMUnitHandle</b></p> <p>This value is used to locate the PLDM OEM Unit PDR that defines the OEMUnit if the OEMUnit value is used for the auxUnit.</p>
bool8	<p><b>isLinear</b></p> <p>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."</p> <p>value: This field is typically set to "true".</p>
enum8	<p><b>sensorDataSize</b></p> <p>The bit width and format of reading and threshold values that the sensor returns</p> <p>value: { uint8, sint8, uint16, sint16, uint32, sint32 }</p>
real32	<p><b>resolution</b></p> <p>The resolution of the sensor in Units (see 27.7).</p>

Type	Description
real32	<p><b>offset</b></p> <p>A constant value that is added in as part of the conversion process of converting a raw sensor reading to Units (see 27.7).</p>
uint16	<p><b>accuracy</b></p> <p>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.</p>
uint8	<p><b>plusTolerance</b></p> <p>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.</p> <p>See 27.6 for more information about how tolerance is defined and used.</p>
uint8	<p><b>minusTolerance</b></p> <p>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.</p> <p>See 27.6 for more information about how tolerance is defined and used.</p>
uint8   sint8   uint16   sint16   uint32   sint32	<p><b>hysteresis</b></p> <p>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.</p> <p>The size of this field is identified by sensorDataSize.</p> <p>value: 1 or greater</p> <p>special value: 0 = sensor does not use hysteresis</p>
bitfield8	<p><b>supportedThresholds</b></p> <p>For PLDM: bit field where bit position represents whether a given threshold is supported</p> <p>0x1b = threshold is supported</p> <p>0x0b = threshold is not supported</p> <p>[6:7] – reserved</p> <p>[5] – lowerThresholdFatal</p> <p>[4] – lowerThresholdCritical</p> <p>[3] – lowerThresholdWarning</p> <p>[2] – upperThresholdFatal</p> <p>[1] – upperThresholdCritical</p> <p>[0] – upperThresholdWarning</p>

Type	Description
bitfield8	<p><b>thresholdAndHysteresisVolatility</b></p> <p>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.</p> <p>special value: 00000b = non-volatile. The threshold settings retained indefinitely regardless of system state.</p> <p>[7:5] – reserved</p> <p>[4] – 1b = PLDM terminus returns to online condition</p> <p>[3] – 1b = System warm resets</p> <p>[2] – 1b = System hard resets</p> <p>[1] – 1b = PLDM subsystem power up</p> <p>[0] – 1b = Initialization Agent controller restart/update (initialize/reinitialize this sensor whenever the device that holds the Initialization Agent has been restarted or reinitialized)</p>
real32	<p><b>stateTransitionInterval</b></p> <p>How long the sensor device takes to do an enabledState change (worst case), in seconds.</p> <p>NOTE: Because this is floating point format, fractional seconds can be represented. The real32 format also supports a value for "Unknown".</p>
real32	<p><b>updateInterval</b></p> <p>Polling or update interval in seconds expressed using a floating point number (generally corresponds to the CIM PollingInterval property)</p>
uint8   sint8   uint16   sint16   uint32   sint32	<p><b>maxReadable</b></p> <p>The maximum value that the sensor may return. The size of this field is given by the sensorDataSize field in this PDR.</p> <p>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.</p>
uint8   sint8   uint16   sint16   uint32   sint32	<p><b>minReadable</b></p> <p>The minimum value that the sensor may return. The size of this field is given by the sensorDataSize field in this PDR.</p> <p>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.</p>
enum8	<p><b>rangeFieldFormat</b></p> <p>Indicates the format used for the following nominalReading, normalMax, normalMin, criticalMax, criticalMin, fatalMax, and fatalMin fields.</p> <p>value: { uint8, sint8, uint16, sint16, uint32, sint32, real32 }</p>

Type	Description
bitfield8	<p><b>rangeFieldSupport</b></p> <p>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.)</p> <ul style="list-style-type: none"> <li>[7] – reserved</li> <li>[6] – 1b = fatalLow field supported</li> <li>[5] – 1b = fatalHigh field supported</li> <li>[4] – 1b = criticalLow field supported</li> <li>[3] – 1b = criticalHigh field supported</li> <li>[2] – 1b = normalMin field supported</li> <li>[1] – 1b = normalMax field supported</li> <li>[0] – 1b = nominalValue field supported</li> </ul>
uint8   sint8   uint16   sint16   uint32   sint32   real32	<p><b>nominalValue</b></p> <p>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.</p> <p>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.</p> <p>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.</p> <p>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.</p> <p>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.</p>
uint8   sint8   uint16   sint16   uint32   sint32   real32	<p><b>normalMax</b></p> <p>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.</p>

Type	Description
uint8   sint8   uint16   sint16   uint32   sint32   real32	<b>normalMin</b>  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	<b>warningHigh</b>  A warning condition that occurs when the monitored value is <i>greater than</i> 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.
uint8   sint8   uint16   sint16   uint32   sint32   real32	<b>warningLow</b>  A warning condition that occurs when the monitored value is <i>less than or equal to</i> 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	<b>criticalHigh</b>  A critical condition that occurs when the monitored value is <i>greater than or equal to</i> 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	<b>criticalLow</b>  A critical condition that occurs when the monitored value is <i>less than</i> 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	<b>fatalHigh</b>  A fatal condition that occurs when the monitored value is <i>greater than</i> 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	<b>fatalLow</b>  A fatal condition that occurs when the monitored value is <i>less than</i> 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.

2534 **28.5 Numeric Sensor Initialization PDR**

2535 The Numeric Sensor Initialization PDR is used when a PLDM Numeric Sensor requires initialization by a  
 2536 PLDM Initialization Agent. Table 68 describes the format of this PDR.

2537 **Table 68 – Numeric Sensor Initialization PDR format**

Type	Description
–	<b>commonHeader</b> See 28.1.
uint16	<b>PLDMTerminusHandle</b> A handle that identifies PDRs that belong to a particular PLDM terminus
uint16	<b>sensorID</b> ID of the sensor relative to the given PLDM Terminus ID
bitfield8	<b>initConditions</b> 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	<b>sensorEnable</b> 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	<b>thresholdInitMask</b> 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	<b>sensorDataSize</b> The bit width of reading and threshold values that the sensor returns value: { uint8, sint8, uint16, sint16, uint32, sint32 }

Type	Description
uint8   sint8   uint16   sint16   uint32   sint32	<b>upperThresholdWarning</b> 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	<b>upperThresholdCritical</b> 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	<b>upperThresholdFatal</b> 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	<b>lowerThresholdWarning</b> 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	<b>lowerThresholdCritical</b> 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	<b>lowerThresholdFatal</b> This value is given in raw units for the sensor. The size of this field is given by the sensorDataSize field in this PDR.

2538 **28.6 State Sensor PDR**

2539 The State Sensor PDR provides the sensorID for a composite state sensor within a PLDM terminus and  
 2540 the number of sensors, and the state set and the possible state values for each sensor that is accessed  
 2541 through the given sensorID. The record also identifies the entity that is being monitored by the sensor.  
 2542 Only one set of fields exists for the entity identification information. Therefore, all sensors in this record  
 2543 must be associated with the same entity. Table 69 describes the format of this PDR.

2544 **Table 69 – State Sensor PDR format**

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

Type	Description
uint16	<p><b>containerID</b></p> <p>The containerID for the containing entity that is associated with this sensor. See 9.1 for more information.</p>
enum8	<p><b>sensorInit</b></p> <p>Indicates whether the sensor requires initialization by the initializationAgent.</p> <pre> value: { nolnit,           // 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 initialize 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.     } </pre>
bool8	<p><b>sensorAuxiliaryNamesPDR</b></p> <p>true = sensor has a Sensor Auxiliary Names PDR  false = sensor does not have an associated Sensor Auxiliary Names PDR</p>
uint8	<p><b>compositeSensorCount</b></p> <p>The number of state sensors in the terminus that are accessed under the sensorID given in this PDR</p> <p>value: 0x01 to 0x08</p>
var	<p><b>possibleStates</b></p> <p>One instance of State Sensor Possible States Fields (see Table 70) for each sensor in the PLDM State Sensor, up to sensorCount.</p>

2545

**Table 70 – State Sensor possible states fields format**

Type	Description
uint16	<p><b>stateSetID</b></p> <p>A numeric value that identifies the PLDM State Set that is used with this sensor</p>
uint8	<p><b>possibleStatesSize</b></p> <p>The number of bytes (M) in the following possibleStates bitfield</p> <p>value: 0x01 to 0x20</p> <p>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.</p>

Type	Description
bitfield8 x M	<p><b>possibleStates [subset of the State Set that is supported]</b></p> <p>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.</p> <p>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.</p> <p>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.</p> <p>[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.</p> <p>...</p> <p>[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.</p> <p>[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.</p> <p>[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.</p>

2546 **28.7 State Sensor Initialization PDR**

2547 The State Sensor Initialization PDR contains values that direct the Initialization Agent's initialization of a  
 2548 particular PLDM Single or Composite State Sensor. This action includes enabling or disabling PLDM  
 2549 Event Message generation for individual sensors within the PLDM Composite State Sensor and directing  
 2550 whether a particular sensor will assess an event if the initialization state value does not match the present  
 2551 state of the sensor.

2552 The PDR always has eight state values (stateValue0 through stateValue7). Dummy values must be used  
 2553 (0x00 is recommended) if the implementation does not have a sensor that corresponds to a particular  
 2554 offset. Table 71 describes the format of the PDR.

2555 **Table 71 – State Sensor Initialization PDR format**

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

Type	Description
bitfield8	<p><b>initConditions</b></p> <p>Identifies under which conditions the Initialization Agent must initialize or reinitialize these sensors</p> <p>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.</p> <p>[7:5] – reserved</p> <p>[4] – 1b = PLDM terminus returns to online condition</p> <p>[3] – 1b = System warm resets</p> <p>[2] – 1b = System hard resets</p> <p>[1] – 1b = PLDM subsystem power up</p> <p>[0] – 1b = Initialization Agent controller restart/update (initialize/reinitialize this sensor whenever the device that holds the Initialization Agent has been restarted or reinitialized)</p>
enum8	<p><b>sensorEnable</b></p> <p>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.</p> <p>special value: {0xFF = do not set the sensorOperationalStates}</p>
bitfield8	<p><b>sensorInitMask</b></p> <p>Identifies which sensors within the composite state sensor require initialization</p> <p>[7] – 1b = state sensor at offset 7 requires initialization 0b = state sensor at offset 7 does not require initialization</p> <p>[6] – 1b = state sensor at offset 6 requires initialization 0b = state sensor at offset 6 does not require initialization</p> <p>...</p> <p>[2] – 1b = state sensor at offset 2 requires initialization 0b = state sensor at offset 2 does not require initialization</p> <p>[1] – 1b = state sensor at offset 1 requires initialization 0b = state sensor at offset 1 does not require initialization</p> <p>[0] – 1b = state sensor at offset 0 requires initialization 0b = state sensor at offset 0 does not require initialization</p>

Type	Description
bitfield8	<p><b>sensorOpStateEventEnableMask</b></p> <p>Identifies which sensors within the composite state sensor should have their operational state event message generation enabled after initialization</p> <p>[7] – 1b = enable event message generator for state sensor at offset 7 0b = disable event message generator for state sensor at offset 7</p> <p>[6] – 1b = enable event message generator for state sensor at offset 6 0b = disable event message generator for state sensor at offset 6</p> <p>...</p> <p>[2] – 1b = enable event message generator for state sensor at offset 2 0b = disable event message generator for state sensor at offset 2</p> <p>[1] – 1b = enable event message generator for state sensor at offset 1 0b = disable event message generator for state sensor at offset 1</p> <p>[0] – 1b = enable event message generator for state sensor at offset 0 0b = disable event message generator for state sensor at offset 0</p>
bitfield8	<p><b>sensorStateEventEnableMask</b></p> <p>Identifies which sensors within the composite state sensor should have their state event message generation enabled after initialization</p> <p>[7] – 1b = enable event message generator for state sensor at offset 7 0b = disable event message generator for state sensor at offset 7</p> <p>[6] – 1b = enable event message generator for state sensor at offset 6 0b = disable event message generator for state sensor at offset 6</p> <p>...</p> <p>[2] – 1b = enable event message generator for state sensor at offset 2 0b = disable event message generator for state sensor at offset 2</p> <p>[1] – 1b = enable event message generator for state sensor at offset 1 0b = disable event message generator for state sensor at offset 1</p> <p>[0] – 1b = enable event message generator for state sensor at offset 0 0b = disable event message generator for state sensor at offset 0</p>
bitfield8	<p><b>sensorEventRearm</b></p> <p>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</p> <p>sensorEventRearm value:</p> <p>1b = trigger an event if the initialization stateValue does not match the present state 0b = accept the initialization stateValue as the present state</p> <p>[7] – sensorEventRearm value for the state sensor at offset 7</p> <p>[6] – sensorEventRearm value for the state sensor at offset 6</p> <p>...</p> <p>[2] – sensorEventRearm value for the state sensor at offset 2</p> <p>[1] – sensorEventRearm value for the state sensor at offset 1</p> <p>[0] – sensorEventRearm value for the state sensor at offset 0</p>

Type	Description
uint8	<b>stateValue0</b> 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	<b>stateValue1</b> 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	<b>stateValue2</b> 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	<b>stateValue6</b> 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	<b>stateValue7</b> 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.

2556 **28.8 Sensor Auxiliary Names PDR**

2557 The Sensor Auxiliary Names PDR may be used to provide optional information that names the sensor.  
 2558 This record may be used for a single numeric or state sensor, or multiple sensors if the sensor is a  
 2559 composite state sensor.

2560 The nameLanguageTag field can be used to identify the language (such as French, Italian, or English)  
 2561 that is associated with the particular sensorName. Table 72 describes the format of this PDR.

2562 **Table 72 – Sensor Auxiliary Names PDR format**

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

Type	Description
uint8	<p><b>sensorCount [1..M]</b></p> <p>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.</p> <p>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.</p>
uint8	<p><b>nameStringCount</b></p> <p>Number of following pairs [0..N] of nameLanguageTag + sensorName fields for sensor[1].</p>
strASCII	<p><b>nameLanguageTag [1]</b></p> <p>This field is absent if nameStringCount = 0.</p> <p>A null-terminated ISO646 ASCII string that holds a language tag, per <a href="#">RFC4646</a>, 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.</p> <p>special value: null string = 0x0000 = unspecified</p>
strUTF-16BE	<p><b>sensorName [1]</b></p> <p>This field is absent if nameStringCount = 0.</p> <p>A null-terminated unicode string for the auxiliary name of the sensor</p> <p>special value: null string = 0x0000 = name not provided</p>
...	...
strASCII	<b>nameLanguageTag [N]</b>
strUTF-16BE	<b>sensorName [N]</b>

2563 **28.9 OEM Unit PDR**

2564 The OEM Unit PDR is used to define one or more strings that are used as the name for an OEM Unit  
 2565 used for PLDM sensors or effecters. The OEM Unit is defined relative to the given Vendor ID and for a  
 2566 given terminus. The OEMUnitHandle value is required to be unique among all OEM Unit PDRs within a  
 2567 PDR Repository. The OEMUnitHandle value is not required to be unique across PDR Repositories.

2568 The record also includes a vendor-defined OEMUnitID value that identifies different types of OEM Units  
 2569 from the given vendor.

2570 The record allows the unit name to be specified using multiple character sets. The unitLanguageTag can  
 2571 be used to identify the language that is associated with the particular unitName (for example, whether the  
 2572 unitName is in French, Italian, English, and so on). Table 73 describes the format of this PDR.

2573

Table 73 – OEM Unit PDR format

Type	Description
–	<b>commonHeader</b> See 28.1.
uint16	<b>PLDMTerminusHandle</b> The terminus that originated this PDR
uint8	<b>OEMUnitHandle</b> An opaque number that is used to identify different OEM Units PDRs
uint32	<b>vendorIANA</b> The IANA Enterprise Number for the vendor that is defining the OEM Sensor Unit
uint8	<b>OEMUnitID</b> 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	<b>stringCount</b> The number 1..N of unitLanguageTag and unitName field pairs that follow this field
strASCII	<b>unitLanguageTag[1]</b> A null terminated ISO646 ASCII string that holds a language tag, per <a href="#">RFC4646</a> , 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	<b>unitName[1]</b> A null terminated unicode string that contains the name of the OEM Sensor Unit
...	...
strASCII	<b>unitLanguageTag[N]</b>
strUTF-16BE	<b>unitName[N]</b>

2574 **28.10 OEM State Set PDR**

2575 The OEM State Set PDR is used to identify the vendor and OEM State Set ID value when the stateSetID  
 2576 is treated as an OEMStateSetIDHandle. The PDR can also optionally be used to provide names for the  
 2577 different OEM-defined states. Each different state can be assigned a name in one or more languages. A  
 2578 contiguous range of state values can also be assigned a single set of names. It is also possible for the  
 2579 PDR to provide a “hint” to help an entity such as a MAP decide how to treat state values that are not  
 2580 explicitly specified in the PDR. The OEM State Set PDR is applicable to OEM State Sets for both sensors  
 2581 and effecters.

2582 Depending on what range the stateSetID value falls in, the stateSetID value in a PDR, such as the PLDM  
 2583 State Sensor PDR, either identifies the state set number for a particular state set defined in [DSP0249](#) or  
 2584 is a value that is interpreted as an OEMStateSetIDHandle. The OEMStateSetIDHandle value is used to  
 2585 form an association with a particular PLDMOEMStateSetPDR within the PDR Repository.  
 2586 OEMStateSetIDHandle values are thus required to be unique for each different PLDM OEM State Set  
 2587 PDR within a given PDR Repository.

2588 The following example describes the steps that could be taken to interpret the state value information  
 2589 from an event message that originated from a PLDM State Sensor. This includes showing the difference  
 2590 between using one of the standard state set numbers and an OEM State Set number.

- 2591 1) A PLDM Event Message is received from a state sensor.
- 2592 2) The TID, sensorID, sensorOffset, and state values (that is, eventState and previousEventState)  
 2593 are read from the message.
- 2594 3) The TID is used to look up the Terminus Locator Record and obtain the PLDMTerminusHandle  
 2595 value that is associated with the TID.
- 2596 4) PLDMTerminusHandle and sensorID values are used to look up the PLDM State Sensor PDR  
 2597 for the sensor.
- 2598 5) The Sensor Offset is used to get the stateSetID from the PLDM State Sensor PDR. If the  
 2599 stateSetID is in the range of standard IDs, the meaning of the state value is given according to  
 2600 the stateSetID defined by the state set identified in [DSP0249](#).
- 2601 6) Otherwise the stateSetID from the PLDM State Sensor PDR is used as an  
 2602 OEMStateSetIDHandle to look up the OEM State Set PDR that defines the OEM State Set. The  
 2603 PDR identifies the OEM that defined the state set and provides the OEM-specified State Set  
 2604 number (OEMStateSetID) for the state set. The state value from the event message can be  
 2605 used to locate the OEM State Value Record in the PLDM OEM State Set PDR that provides a  
 2606 name string for the particular OEM-defined state.

2607 Table 74 describes the format of the PDR.

2608 **Table 74 – OEM State Set PDR format**

Type	Description
–	<b>commonHeader</b> See 28.1.
uint16	<b>PLDMTerminusHandle</b> The terminus that originated this PDR
uint16	<b>OEMStateSetIDHandle</b> An OEM State Set within this PDR Repository. The value is taken from the range of OEMStateSet numbers defined in <a href="#">DSP0249</a> .  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	<b>vendorIANA</b> The IANA Enterprise Number for the vendor that is defining the OEM State Set given in this PDR
uint16	<b>OEMStateSetID</b> 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 <a href="#">DSP0249</a> .

Type	Description
enum8	<p><b>unspecifiedValueHint</b></p> <p>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.</p> <p>value: { treatAsUnspecified, treatAsError }</p>
uint8	<p><b>stateCount</b></p> <p>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.</p>
variable	<p><b>OEMStateValueRecord</b></p> <p>Zero or more OEM State Value Records as specified by the stateCount field. See Table 75.</p>

2609

**Table 75 – OEM State Value Record format**

Type	Description
uint8	<p><b>minStateValue</b></p> <p>The lowest state enumeration value that corresponds to the definition given in this OEM State Value Record instance.</p>
uint8	<p><b>maxStateValue</b></p> <p>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.</p> <p>If maxStateValue = minStateValue, the following strings apply only to a single state.</p> <p>If maxStateValue &gt; minStateValue, the following strings apply to state values in the range from minStateValue through maxStateValue.</p>
uint8	<p><b>stringCount</b></p> <p>The number 1..N of stateLanguageTag and stateName field pairs that follow this field.</p>
strASCII	<p><b>stateLanguageTag[1]</b></p> <p>A null terminated ISO646 ASCII string that holds a language tag, per <a href="#">RFC4646</a>, 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.</p> <p>special value: null string = unspecified</p>
strUTF-16BE	<p><b>stateName[1]</b></p> <p>A null terminated unicode string that contains the name for the state</p>
...	...
strASCII	<p><b>stateLanguageTag[N]</b></p>
strUTF-16BE	<p><b>stateName[N]</b></p>

2610 **28.11 Numeric Effector PDR**

2611 The Numeric Effector PDR is used to describe the semantics of a PLDM Numeric Effector to a party such  
 2612 as a MAP. It also includes the factors that are used for converting raw sensor readings to normalized  
 2613 units. The PDR also identifies the entity on which the effector is operating. Table 76 describes the format

2614 of the PDR. NOTE: The Numeric Effector PDR effectorID type in this clause has been changed in version  
 2615 1.1.1 of this specification from uint8 to uint16 to be consistent with SetNumericEffectorEnable command.

2616

2617

**Table 76 – Numeric Effector PDR format**

Type	Description
–	<b>commonHeader</b> See 28.1.
uint16	<b>PLDMTerminusHandle</b> A handle that identifies PDRs that belong to a particular PLDM terminus
uint16	<b>effectorID</b> ID of the effector relative to the given PLDM Terminus ID.
uint16	<b>entityType</b> The Type value for the entity that is associated with this effector. See 9.1 for more information.
uint16	<b>entityInstanceNumber</b> The Instance Number for the entity that is associated with this effector. See 9.1 for more information.
uint16	<b>containerID</b> The containerID for the containing entity that is associated with this effector. See 9.1 for more information.
uint16	<b>effectorSemanticID</b> This field either identifies a PLDM-defined effector semantic or provides an OEMEffectorSemanticHandle value, depending on what range the value falls in. If the effectorSemanticID 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 OEMEffectorSemanticHandle that can be used to locate an OEM Effector Semantic PDR that identifies the vendor and provides optional name information for the semantic. See <a href="#">DSP0249</a> for the definition of Effector Semantic ID values and ranges, and 21.3 for more information.  special value: {0x0000 = unspecified }
enum8	<b>effectorInit</b> value: { nolnit, // The Initialization Agent does not take any steps to initialize, // enable, or disable this particular sensor.  useInitPDR, // The sensor has an associated Numeric Effector Initialization // PDR that should be used to initialize the sensor.  enableEffector, // When the Initialization Agent runs, it enables this effector using // a SetNumericEffectorEnable command to set the // operationalState.  disableEffector // When the Initialization Agent runs, it disables this effector using // the SetNumericEffectorEnable command.  }
bool8	<b>effectorAuxiliaryNames PDR</b> true = effector has an Effector Auxiliary Names PDR false = effector does not have an associated Effector Auxiliary Names PDR

Type	Description
enum8	<p><b>baseUnit</b></p> <p>The base unit of the reading returned by this effector. See 27.1 for more information. value: { see Table 63 }</p>
sint8	<p><b>unitModifier</b></p> <p>A power-of-10 multiplier for the baseUnit. See 27.1 for more information.</p>
enum8	<p><b>rateUnit</b></p> <p>value: { None, Per MicroSecond, Per MilliSecond, Per Second, Per Minute, Per Hour, Per Day, Per Week, Per Month, Per Year }</p>
uint8	<p><b>baseOEMUnitHandle</b></p> <p>This value is used to locate the PLDM OEM Unit PDR that defines the OEMUnit if the OEMUnit value is used for the baseUnit.</p>
enum8	<p><b>auxUnit</b></p> <p>The base unit of the reading returned by this effector. See 27.2 for more information. value: { see Table 63 }</p>
sint8	<p><b>auxUnitModifier</b></p> <p>A power-of-10 multiplier for the auxUnit. See 27.2 for more information.</p>
enum8	<p><b>auxrateUnit</b></p> <p>value: { None, Per MicroSecond, Per MilliSecond, Per Second, Per Minute, Per Hour, Per Day, Per Week, Per Month, Per Year }</p>
uint8	<p><b>auxOEMUnitHandle</b></p> <p>This value is used to locate the PLDM OEM Unit PDR that defines the OEMUnit if the OEMUnit value is used for the auxUnit.</p>
bool8	<p><b>isLinear</b></p> <p>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".</p>
enum8	<p><b>effectorDataSize</b></p> <p>The bit width and format of reading and threshold values that the effector returns value: { uint8, sint8, uint16, sint16, uint32, sint32 }</p>
real32	<p><b>resolution</b></p> <p>The resolution of the effector in Units (see 27.7)</p>
real32	<p><b>offset</b></p> <p>A constant value that is added as part of the conversion process of converting a raw effector reading to Units (see 27.7).</p>
uint16	<p><b>accuracy</b></p> <p>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.</p>

Type	Description
uint8	<p><b>plusTolerance</b></p> <p>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 effector. 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.</p> <p>See 27.6 for more information about how tolerance is defined and used.</p>
uint8	<p><b>minusTolerance</b></p> <p>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 effector. 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.</p> <p>See 27.6 for more information about how tolerance is defined and used.</p>
real32	<p><b>stateTransitionInterval</b></p> <p>The length of time the effector takes to do an enabledState change (worst case), in seconds</p> <p>NOTE: Because this is floating point format, fractional seconds can be represented. The real32 format also supports a value for "Unknown".</p>
real32	<p><b>TransitionInterval</b></p> <p>The length of time the effector takes to have a setting change take effect (worst case), in seconds.</p>
uint8   sint8   uint16   sint16   uint32   sint32	<p><b>maxSettable</b></p> <p>The maximum legal setting value that the effector accepts. The size of this field is given by the effectorDataSize field in this PDR.</p> <p>This number is given in the same format as the reading returned by the effector. The conversion formula is used to convert this number to normalized units. See definition in 27.1.</p>
uint8   sint8   uint16   sint16   uint32   sint32	<p><b>minSettable</b></p> <p>The minimum legal setting value that the effector accepts. The size of this field is given by the effectorDataSize field in this PDR.</p> <p>This number is given in the same format as the reading returned by the effector. The conversion formula is used to convert this number to normalized units. See definition in 27.1.</p>
enum8	<p><b>rangeFieldFormat</b></p> <p>Indicates the format used for the following nominalValue, normalMax, and normalMin fields.</p> <p>value: { uint8, sint8, sint16, uint32, sint32, real32 }</p>
Bitfield8	<p><b>rangeFieldSupport</b></p> <p>This field indicates which of the fields that identify the operating ranges of the parameter set by the effector are supported. (This bitfield indicates whether the following nominalValue, normalMax, and so on, fields contain valid range values.)</p> <ul style="list-style-type: none"> <li>[7:5] – reserved</li> <li>[4] – 1b = ratedMin field supported</li> <li>[3] – 1b = ratedMax field supported</li> <li>[2] – 1b = normalMin field supported</li> <li>[1] – 1b = normalMax field supported</li> <li>[0] – 1b = nominalValue field supported</li> </ul>

Type	Description
uint8   sint8   uint16   sint16   uint32   sint32   real32	<p><b>nominalValue</b></p> <p>This value presents the nominal value for the parameter that is accepted by the effector. 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.</p> <p>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 effector (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.</p> <p>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 effector implementation. For example, if the nominal value for a voltage setting effector was 5.00 V, the nominalValue would typically be reported as 5.00 V even though the closest setting the effector implementation may be able to accept is 5.05 V.</p> <p>A common use of the nominalValue is as a source of part of the identifying “name” for an effector. For example, it is common for voltage effectors to be identified by their nominal reading. So, an effector with a nominal reading of +5.00 V would be referred to as a “+5 V effector”, while one with a nominal reading of +3.3 V would be referred to as a “+3.3 V effector”. 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 effector. For example, if the nominalValue is given as a real32 as 2.50000 V, but the resolution of the effector is 0.05 V, the nominalValue displayed would typically be rounded as 2.50 V.</p> <p>It is possible that a given effector may not be considered as having a nominal setting, in which case this field should be ignored. For example, a numeric effector that sets a count or size of some parameter may not be considered as having a nominal setting depending on its application.</p>
uint8   sint8   uint16   sint16   uint32   sint32   real32	<p><b>normalMax</b></p> <p>The upper limit of the normal operating range for the parameter that is set by the numeric effector. 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 effector.</p>
uint8   sint8   uint16   sint16   uint32   sint32   real32	<p><b>normalMin</b></p> <p>The lower limit of the normal operating range for the parameter that is set by the numeric effector. Effector thresholds are typically set for a value that is lower than normalMin to accommodate the affects of effector 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.</p>
uint8   sint8   uint16   sint16   uint32   sint32   real32	<p><b>ratedMax</b></p> <p>The upper limit of the rated operating range for the parameter that is set by the numeric effector. The monitored parameter is considered to be operating outside of rated operating range when this value is exceeded.</p>
uint8   sint8   uint16   sint16   uint32   sint32   real32	<p><b>ratedMin</b></p> <p>The lower limit of the rated operating range for the parameter that is set by the numeric effector. The monitored parameter is considered to be operating outside of rated operating range below this value.</p>

2618 **28.12 Numeric Effector Initialization PDR**

2619 The Numeric Effector Initialization PDR reports the values that are used when a PLDM Effector Sensor is  
 2620 initialized by a PLDM Initialization Agent. Table 77 describes the format of this PDR.

2621 **Table 77 – Numeric Effector Initialization PDR format**

Type	Description
–	<b>commonHeader</b> See 28.1.
uint16	<b>PLDMTerminusHandle</b> A handle that identifies PDRs that belong to a particular PLDM terminus
uint16	<b>effectorID</b> ID of the effector relative to the given PLDM Terminus ID
enum8	<b>effectorEnable</b> The operational state of the effector after it has been initialized. This state is written to the effector using the SetEffectorEnable command.  special value: {0xFF = do not issue a SetEffectorEnable command to set the Effector Operational State }
bitfield8	<b>initConditions</b> Identifies under which conditions the Initialization Agent must initialize or reinitialize this effector [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 effector whenever the device that holds the Initialization Agent has been restarted or reinitialized)
enum8	<b>effectorDataSize</b> The bit width of reading and threshold values that the effector returns value: { uint8, sint8, uint16, sint16, uint32, sint32 }
uint8   sint8   uint16   sint16   uint32   sint32	<b>effectorData</b> The numeric value written to the effector. The size of this field is determined by the value of the effectorDataSize field.

2622 **28.13 State Effector PDR**

2623 The State Effector PDR is used to provide information about a PLDM Composite State Effector. Table 78  
 2624 describes the format of this PDR.

2625 **Table 78 – State Effector PDR format**

Type	Description
–	<b>commonHeader</b> See 28.1.
uint16	<b>PLDMTerminusHandle</b> A handle that identifies PDRs that belong to a particular PLDM terminus
uint16	<b>effectorID</b> ID of the effector relative to the given PLDM Terminus ID
uint16	<b>entityType</b> The Type value for the entity that is associated with this sensor. See 9.1. for more information.
uint16	<b>entityInstanceNumber</b> The Instance Number for the entity that is associated with this sensor. See 9.1. for more information.
uint16	<b>containerID</b> The containerID for the containing entity that is associated with this sensor. See 9.1. for more information.
uint16	<b>effectorSemanticID</b> This field either identifies a PLDM-defined effector semantic or provides an OEMEffectorSemanticHandle value, depending on what range the value falls in. If the effectorSemanticID 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 OEMEffectorSemanticHandle that can be used to locate an OEM Effector Semantic PDR that identifies the vendor and provides optional name information for the semantic. See <a href="#">DSP0249</a> for the definition of Effector Semantic ID values and ranges, and 21.3 for more information.  special value: {0x0000 = unspecified }
enum8	<b>effectorInit</b> value: { nolnit, // The Initialization Agent does not take any steps to initialize, // enable, or disable this particular effector.  useInitPDR, // The effector has an associated State Effector Initialization PDR // that should be used to initialize the effector.  enableEffector, // When the Initialization Agent runs, it enables this effector using // a SetStateEffectorEnables command to set the // operationalState.  disableEffector. // When the Initialization Agent runs, it disables this effector using // the SetStateEffectorEnables command.  }
bool8	<b>effectorDescriptionPDR</b> true = effector has an effectorDescription PDR false = effector does not have an associated effectorDescription PDR

Type	Description
uint8	<p><b>compositeEffectorCount</b></p> <p>The number of state effectors in the terminus that are accessed under the effectorID given in this PDR.</p> <p>value: 0x01 to 0x08</p>
var	<p><b>possibleStates</b></p> <p>One instance of State Effector Possible States Fields (see Table 79) for each effector in the PLDM State Effector, up to effectorCount.</p>

2626

**Table 79 – State Effector Possible States Fields format**

Type	Description
uint16	<p><b>stateSetID</b></p> <p>A numeric value that identifies the PLDM State Set that is used with this effector.</p>
uint8	<p><b>possibleStatesSize</b></p> <p>The number of bytes (M) in the possibleStates bitfield.</p> <p>value: 0x01 to 0x20</p> <p>special value : 0x00 can be used to indicate a effector that is unavailable or disabled from use and should be ignored when accessing the parent composite effector with PLDM.</p>
bitfield8 x M	<p><b>possibleStates [subset of the State Set that is supported]</b></p> <p>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.</p> <p>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.</p> <p>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.</p> <p>[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</p> <p>...</p> <p>[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</p> <p>[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</p> <p>[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</p>

2627

**28.14 State Effector Initialization PDR**

2628  
2629

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

2630 The PDR always has eight state values. Dummy values must be used (0x00 is recommended) if the  
 2631 implementation does not have an effector that corresponds to a particular offset. Table 80 describes the  
 2632 format of the PDR.

2633 **Table 80 – State Effector Initialization PDR format**

Type	Description
–	<b>commonHeader</b> See 28.1.
uint16	<b>PLDMTerminusHandle</b> A handle that identifies PDRs that belong to a particular PLDM terminus
uint16	<b>effectorID</b> ID of the effector relative to the given PLDM terminus
uint16	<b>entityType</b> The Type value for the entity that is associated with this sensor. See 9.1 for more information.
uint16	<b>entityInstanceNumber</b> The Instance Number for the entity that is associated with this sensor. See 9.1 for more information.
uint16	<b>containerID</b> The containerID for the containing entity that is associated with this sensor. See 9.1 for more information.
bitfield8	<b>initConditions</b> Identifies the conditions under which the Initialization Agent must initialize or reinitialize this effector [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 effector whenever the device that holds the Initialization Agent has been restarted or reinitialized)
enum8	<b>effectorEnable</b> 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 effectorInitMask field of this PDR using the SetStateEffectorEnables command. special value: {0xFF = do not set the effectorOperationalStates}

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

2634 **28.15 Effector Auxiliary Names PDR**

2635 The Effector Auxiliary Names PDR may be used to provide optional information that names an effector.  
 2636 This record may be used for a single effector or multiple effectors if the effector is a composite state  
 2637 effector.

2638 The nameLanguageTag field can be used to identify the language (such as French, Italian, or English)  
 2639 that is associated with the particular effector name. Table 81 describes the format of this PDR.

2640 **Table 81 – Effector Auxiliary Names PDR format**

Type	Description
–	<b>commonHeader</b> See 28.1.
uint16	<b>PLDMTerminusHandle</b> A handle that identifies PDRs that belong to a particular PLDM terminus
uint16	<b>effectorID</b> ID of the effector relative to the given PLDM terminus
uint8	<b>effectorCount [1..M]</b> For each effector x in effectorCount, there can be 1..nameStringCount[x] strings, where each set of strings corresponds to a effector in a composite effector. The record must be populated sequentially starting from 1 regardless of whether an effector requires auxiliary names. Thus, each entry has at least one byte (the nameStringCount). Effectors that have offsets that are greater than effectorCount are treated as if they have no auxiliary names. For example, if a composite effector contains four effectors and only the third effector requires an auxiliary name, the effectorCount can be 3 and the nameStringCount for the first two sets of effector name information is 0.
<b>effector [1] names:</b>	
uint8	<b>nameStringCount</b> Number of following pairs [0..N] of nameLanguageTag + effectorName fields for effector[1].
strASCII	<b>nameLanguageTag[1]</b> This field is absent if nameStringCount = 0. A null terminated ISO646 ASCII string that holds a language tag, per <a href="#">RFC4646</a> , that identifies the primary language in which the effectorName 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 effectorName are provided. special value: null string = 0x0000 = unspecified
strUTF-16BE	<b>effectorName[1]</b> This field is absent if nameStringCount = 0. A null terminated unicode string for the name of the auxiliary effector special value: null string = 0x0000 = name not provided.
...	...
strASCII	<b>nameLanguageTag[N]</b>
strUTF-16BE	<b>effectorName[N]</b>
<b>effector [2] names:</b>	
...	
<b>effector [M] names:</b>	

2641 **28.16 OEM Effector Semantic PDR**

2642 The OEM Effector Semantic PDR is used to provide information about an OEM effector semantic used  
 2643 with one or more PLDM effectors that are represented in the PDRs. The information includes an ID for the  
 2644 vendor and a vendor-defined ID number for identifying the effector semantic. The PDR also allows one or  
 2645 more descriptive name strings to be provided for the vendor-defined effector semantic. The name strings  
 2646 may be provided in different character sets and languages.

2647 The OEMEffectorSemanticHandle value in the PDR is used by other PDRs, such as the PLDM State  
 2648 Effector PDR, to point to the particular PLDM OEM Effector Semantic PDR within the PDR Repository.  
 2649 OEMStateSetIDHandle values are thus required to be unique for each different PLDM OEM State Set  
 2650 PDR within a given PDR Repository.

2651 The OEMSemanticID field enables the vendor that defined the semantic to assign an ID value to its  
 2652 semantic. The OEMSemanticID field is thus defined relative to the given vendor ID.

2653 The OEM Effector Semantic PDR also contains a PLDMTerminusHandle value. The  
 2654 PLDMTerminusHandle is used to provide a record of the terminus from which the PDR was imported. It is  
 2655 expected that most vendors will define their OEMSemanticID values in a global manner in which the ID  
 2656 has the same meaning regardless of the PLDMTerminusHandle value.

2657 Table 82 describes the format of this PDR.

2658 **Table 82 – OEM Effector Semantic PDR format**

Type	Description
–	<b>commonHeader</b> See 28.1.
uint16	<b>PLDMTerminusHandle</b> This value is used to identify the terminus that originated this PDR.
uint8	<b>OEMEffectorSemanticHandle</b> An opaque number that is used to identify different OEM effector semantics that are defined by the given vendor on the given terminus. The value is used in PDRs such as the PLDM State Effector PDR to indicate that a vendor-defined effector semantic is being used and to locate the PLDM OEM Effector Semantic PDRs (if any) that provide the vendor-defined ID number and optional descriptive names for the effector semantic.
uint32	<b>vendorIANA</b> The IANA Enterprise Number for the vendor that is defining the OEM Sensor Unit
uint8	<b>OEMEffectorSemanticID</b> 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 effector semantic. Thus, the vendor can use this value to provide a constant ID that always identifies a particular Unit definition from that vendor.
uint8	<b>stringCount</b> The number 1..N of languageTag and name field pairs that follow this field. { 0 = no name information provided }
strASCII	<b>languageTag[1]</b> A null terminated ISO646 ASCII string that holds a language tag, per <a href="#">RFC4646</a> , 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	<b>name[1]</b> A null terminated unicode string that contains the name of the OEM Sensor Unit
...	...

Type	Description
strASCII	languageTag[N]
strUTF-16BE	name[N]

2659 **28.17 Entity Association PDR**

2660 The Entity Association PDR is used to form associations between entities, such as physical and logical  
 2661 entities. See clause 10 for more information. Table 83 describes the format of this PDR.

2662 **Table 83 – Entity Association PDR format**

Type	Description
–	<b>commonHeader</b> See 28.1.
uint16	<b>containerID</b> 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	<b>associationType</b> value: { physicalToPhysicalContainment, logicalContainment }
<i>Container Entity Identification Information</i>	
uint16	<b>containerEntityType</b>
uint16	<b>containerEntityInstanceNumber</b>
uint16	<b>containerEntityContainerID</b>
<i>Contained Entity Identification Information</i>	
uint8	<b>containedEntityCount</b> 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	<b>containedEntityType[1]</b>
uint16	<b>containedEntityInstanceNumber[1]</b>
uint16	<b>containedEntityContainerID[1]</b>
	...
uint16	<b>containedEntityType[N]</b>
uint16	<b>containedEntityInstanceNumber[N]</b>
uint16	<b>containedEntityContainerID[N]</b>

2663 **28.18 Entity Auxiliary Names PDR**

2664 The Entity Auxiliary Names PDR may be used to provide optional information that names a particular  
 2665 instance of an entity. The PDR can also be used to name a particular range of instances of an entity,  
 2666 provided that the instances share the same containerID.

2667 The nameLanguageTag field can be used to identify the language (such as French, Italian, or English)  
 2668 that is associated with the particular entity name. Table 84 describes the format of this PDR.

2669 **Table 84 – Entity Auxiliary Names PDR format**

Type	Description
–	<b>commonHeader</b> See 28.1.
uint16	<b>entityType</b>
uint16	<b>entityInstanceNumber</b>
uint16	<b>entityContainerID</b>
uint8	<b>sharedNameCount</b>  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.
<b>Entity auxiliary names:</b>	
uint8	<b>nameStringCount</b>  Number of following pairs [0..N] of nameLanguageTag + entityAuxName fields for entityAuxName[1].
strASCII	<b>nameLanguageTag [1]</b>  This field is absent if nameStringCount = 0.  A null terminated ISO646 ASCII string that holds a language tag, per <a href="#">RFC4646</a> , 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	<b>entityAuxName [1]</b>  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	<b>nameLanguageTag [N]</b>
strUTF-16BE	<b>entityAuxName [N]</b>

2670 **28.19 OEM EntityID PDR**

2671 The OEM EntityID PDR can be used to provide a vendor-specific EntityID definition when no PLDM  
 2672 predefined EntityID corresponds to the type of entity that the vendor wants to represent.

2673 When the entityType value is in the OEM range of values, the EntityID portion of the entityType field is  
 2674 OEM-defined. The EntityID value is then used as an OEMEntityIDHandle to locate the corresponding  
 2675 OEM EntityID PDR.

2676 OEM Entity Type PDRs need to be able to be exported by a terminus, such as a terminus on a hot-plug  
 2677 card. The numbers in a given vendor's Device PDRs must be picked a priori by the vendor. Thus,  
 2678 duplications may exist among the OEM EntityID values that different vendors choose. The Discovery  
 2679 Agent function is responsible for adjusting the OEM Entity Type values to resolve any conflicts that may  
 2680 occur when it integrates PDRs into the Primary PDR Repository. Users of OEM EntityID values must be  
 2681 aware that these values may differ between different PDR Repositories. That is, an OEM EntityID for  
 2682 "widget" from vendor "ABC" will not always have the same Entity ID value across PDRs.

2683 To facilitate the identification of particular OEM EntityIDs from a given vendor, each PDR includes a  
 2684 vendor-specific ID value that does not get altered by the Discovery Agent function. When used in  
 2685 conjunction with the vendor's ID, this provides a value that can always be used to identify the particular  
 2686 vendor-defined EntityID definition.

2687 Table 85 describes the format of this PDR.

2688 **Table 85 – OEM EntityID PDR format**

Type	Description
–	<b>commonHeader</b> See 28.1.
uint16	<b>PLDMTerminusHandle</b> This value is used to identify the terminus that originated this PDR.
uint16	<b>OEMEntityIDHandle</b> [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	<b>vendorIANA</b> The IANA Enterprise Number for the vendor that is defining the OEM PDR vendor-specific data
uint16	<b>vendorEntityID</b> 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	<b>stringCount</b> The number 1..N of entityIDLanguageTag and entityIDName field pairs that follow this field.

Type	Description
strASCII	<b>entityIDLanguageTag[1]</b> A null terminated ISO646 ASCII string that holds a language tag, per <a href="#">RFC4646</a> , 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	<b>entityIDName[1]</b> A null terminated unicode string that contains the name of the EntityID name
...	...
strASCII	<b>entityIDLanguageTag[N]</b>
strUTF-16BE	<b>entityIDName[N]</b>

2689 **28.20 Interrupt Association PDR**

2690 The Interrupt Association PDR is used to form associations between interrupt source entities and interrupt  
2691 target entities. See 11.10 for more information. Table 86 describes the format of this PDR.

2692 **Table 86 - Interrupt Association PDR format**

Type	Description
–	<b>commonHeader</b> See 28.1.
uint16	<b>PLDMTerminusHandle</b> 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	<b>sensorID</b> The ID of the sensor that monitors this interrupt at a source or target
enum8	<b>sourceOrTargetSensor</b> 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 }
<i>Target Entity Identification Information</i>	
uint16	<b>interruptTargetEntityType</b>
uint16	<b>interruptTargetEntityInstanceNumber</b>
uint16	<b>interruptTargetEntityContainerID</b>
<i>Source Entity Identification Information</i>	
uint8	<b>interruptSourceEntityCount</b> 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	<b>interruptSourcePLDMTerminusHandle[1]</b>

Type	Description
uint16	<b>interruptSourceEntityType[1]</b>
uint16	<b>interruptSourceEntityInstanceNumber[1]</b>
uint16	<b>interruptSourceEntityContainerID[1]</b>
uint16	<b>interruptSourceSensorID[1]</b>
	...
uint32	<b>interruptSourcePLDMTerminusHandle[N]</b>
uint16	<b>interruptSourceEntityType[N]</b>
uint16	<b>interruptSourceEntityInstanceNumber[N]</b>
uint16	<b>interruptSourceEntityContainerID[N]</b>
uint16	<b>interruptSourceSensorID[N]</b>

2693 **28.21 Event Log PDR**

2694 The Event Log PDR is used to describe characteristics of the PLDM Event Log (if implemented). The  
 2695 specification defines the existence of only a single, central PLDM Event Log function. Therefore, only one  
 2696 occurrence of a PLDM Event Log PDR shall exist in a Primary PDR Repository.

2697 Table 87 describes the format of this PDR.

2698 **Table 87 – Event Log PDR format**

Type	Description
–	<b>commonHeader</b> See 28.1.
uint32	<b>logSize</b> 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	<b>supportedLogClearingPolicies</b> 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	<p><b>entryIDTimeout</b></p> <p>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.</p> <p>special values: { 0x00 = no timeout, 0x01 = default minimum timeout is the same as the PDR Handle Timeout, <b>MC1</b>, (see clause 29), 0xFF = timeout &gt;254 seconds. Any timeout values that are less than the specified default minimum timeout are illegal. }</p>
uint8	<p><b>perEntryOverhead</b></p> <p>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.</p> <p>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.</p> <p>special value: 0xFF = unspecified</p>
uint8	<p><b>allocationGranularity</b></p> <p>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.</p> <p>special value: 0xFF = unspecified</p>
uint8	<p><b>percentUsedResolution</b></p> <p>Indicates the resolution of the storagePercentUsed value from the GetPLDMEventLogInfo command</p> <p>value: 1 to 100; all other values = reserved</p> <p>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 &lt;1% full, a storagePercentUsed value of 0x01 indicates that the log is 1% to &lt;2% full, and so on.</p> <p>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 &lt;5% full, a storagePercentUsed value of 0x01 indicates that the log is 5% to &lt;10% full, and so on.</p>

2699 **28.22 FRU Record Set PDR**

2700 The FRU Record Set PDR is used to describe characteristics of the PLDM FRU Record Set Data defined  
 2701 in [DSP0257](#). The information can be used to locate a Terminus that holds FRU Record Set Data in order  
 2702 to access that data using the commands specified in [DSP0257](#). The PDR also identifies the particular  
 2703 Entity that is associated with the FRU information.

2704 Table 88 describes the format of this PDR.

2705

2706

**Table 88 – FRU Record Set PDR format**

Type	Description
–	<b>commonHeader</b> See 28.1.
uint16	<b>PLDMTerminusHandle</b> The terminus that originated or maintains this PDR. .
uint16	<b>FRURecordSetIdentifier</b> A unique number per terminus that is used to identify the Record Set for the FRU Data for the associated entity. The Record Set value is used for accessing FRU Data using the commands specified in <a href="#">DSP0257</a> .
uint16	<b>entityType</b> The Type value for the entity that is associated with this FRU data.
uint16	<b>entityInstanceNumber</b> The Instance Number for the entity that is associated with this FRU data.
uint16	<b>containerID</b> The containerID for the containing entity that is associated with this FRU data.

2707 **28.23 OEM Device PDR**

2708 The OEM Device PDR can be used to provide OEM (vendor-specific) information. The OEM-specific data  
 2709 portion in an OEM Device PDR is limited to a maximum size of 64 KB. Higher-level system specifications  
 2710 may place additional limits on the size and number of OEM Device PDRs that may be supported in a  
 2711 given PLDM subsystem implementation. An OEM Device PDR must have at least one byte of  
 2712 VendorSpecificData.

2713 This type of PDR shall be copied by the Discovery Agent into the Primary PDR Repository dependent on  
 2714 the setting of the copyPDR field. The PDR may also be preconfigured into the Primary PDR Repository.  
 2715 That is, this PDR is not restricted to being only used or migrated from repositories that are separate from  
 2716 the Primary PDR Repository.

2717 The OEM PDR is a slightly smaller version of the OEM Device PDR that can be used in situations where  
 2718 it is not necessary or desired to associate the PDR to a particular terminus or have the information copied  
 2719 from a Device PDR Repository into the Primary PDR Repository.

2720 Table 89 describes the format of this PDR.

2721 **28.23.1 Copy Behavior**

2722 If the copyPDR parameter is set to copyToPrimaryRepository, the Discovery Agent shall overwrite any  
 2723 pre-existing PDRs for the terminus that have the same vendorIANA and VendorHandle values.

2724 **28.23.2 Removal Behavior**

2725 The OEM Device PDR is allowed to be removed from the Primary PDR Repository if the Discovery Agent  
 2726 detects that the terminus that is associated with the PDR has been removed or is no longer available.

2727

Table 89 – OEM Device PDR format

Type	Description
–	<b>commonHeader</b> See 28.1.
uint16	<b>PLDMTerminusHandle</b> 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	<b>copyPDR</b> value: { doNotCopy, copyToPrimaryRepository }
uint32	<b>vendorIANA</b> The IANA Enterprise Number for the vendor that is defining the OEM PDR vendor -specific data special value: 0 = unspecified
uint16	<b>OEMRecordID</b> 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	<b>dataLength</b> The number of following vendorSpecificData bytes starting from 0. 0 = 1 byte, 1 = 2 bytes, and so on
byte	<b>vendorSpecificData[0]</b>
...	...
byte	<b>vendorSpecificData[N]</b>

2728 **28.24 OEM PDR**

2729 The OEM PDR can be used to provide OEM (vendor-specific) information. The OEM-specific data portion  
 2730 in an OEM PDR is limited to a maximum size of 64 KB. Higher-level system specifications may place  
 2731 additional limits on the size and number of OEM PDRs that may be supported in a given PLDM  
 2732 subsystem implementation. An OEM PDR must have at least one byte of VendorSpecificData. The OEM  
 2733 Device PDR is an extended version of the OEM PDR that is used when it is necessary to associate the  
 2734 PDR to a particular terminus or to have the information copied from a Device PDR Repository into the  
 2735 Primary PDR Repository.

2736 Table 90 describes the format of this PDR.

2737

**Table 90 – OEM PDR format**

Type	Description
–	<b>commonHeader</b> See 28.1.
uint32	<b>vendorIANA</b> The IANA Enterprise Number for the vendor that is defining the OEM PDR vendor-specific data special value: 0 = unspecified
uint16	<b>OEMRecordID</b> 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	<b>dataLength</b> The number of following vendor-specific data bytes starting from 0 0 = 1 byte, 1 = 2 bytes, and so on.
byte	<b>vendorSpecificData[1]</b>
...	...
byte	<b>vendorSpecificData[N]</b>

2738 **29 Timing**

2739 Table 91 defines timing values that are specific to this document.

2740 **Table 91 – Monitoring and control timing specifications**

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

2741 **30 Command numbers**

2742 Table 92 defines the command numbers used in the requests and responses for the PLDM monitoring  
2743 and control commands defined in this specification.

2744 **Table 92 – Command numbers**

#	Command	Reference
<b>Terminus commands</b>		
0x01	SetTID (see <a href="#">DSP0240</a> )	See 16.1.
0x02	GetTID (see <a href="#">DSP0240</a> )	See 16.2
0x03	GetTerminusUID	See 16.3.
0x04	SetEventReceiver	See 16.4.
0x05	GetEventReceiver	See 16.5.
0x0A	PlatformEventMessage	See 16.6.
<b>Numeric Sensor commands</b>		
0x10	SetNumericSensorEnable	See 18.1.

#	Command	Reference
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.
<b>State Sensor commands</b>		
0x20	SetStateSensorEnables	See 20.1.
0x21	GetStateSensorReadings	See 20.2.
0x22	InitStateSensor	See 20.3.
<b>PLDM Effector commands</b>		
0x30	SetNumericEffectorEnable	See 22.1.
0x31	SetNumericEffectorValue	See 22.2.
0x32	GetNumericEffectorValue	See 22.3.
0x38	SetStateEffectorEnables	See 22.4.
0x39	SetStateEffectorStates	See 22.5.
0x3A	GetStateEffectorStates	See 22.6.
<b>PLDM Event Log commands</b>		
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.
<b>PDR Repository commands</b>		
0x50	GetPDRRepositoryInfo	See 26.1.
0x51	GetPDR	See 26.2.
0x52	FindPDR	See 26.3.
0x58	RunInitAgent	See 26.4.

2745

**ANNEX A  
(informative)**

**Change log**

2746  
2747  
2748  
2749  
2750

Version	Date	Description
1.0.0	2009-03-16	
1.0.1	2010-01-13	Update to correct address issues from TC ballot
1.1.0	2011-11-08	DMTF Standard. Added FRU Record Set PDR and description of FRU Record Set to Entity Association relationship. A 'rel' field that describes the relationship between the base unit and aux unit was added to the Numeric Sensor PDR format. This update also included edits for consistency, typos, and clarifications per Mantis entries, including: References to "effectorDescriptionPDR" and "sensorDescription PDR" in v1.0.x were changed to refer to the EffectorAuxiliaryNames and SensorAuxiliaryNames PDRs, respectively. The enumeration values of effectorOperationalState in Tables 37 and 43 were made consistent. Similarly, the enumeration values for sensorOperationalState in Table 19 & Table 30 were also made consistent. In Table 77, the type of effectorInIt was incorrectly specified as bool8 instead of enum8. In table 19, sensorEventMessageEnable type was specified as bool8 instead of enum8.
1.1.1	2017-01-10	Corrected the data type length of the "sensorID" and corresponding "effectorID" field from "uint8" to "uint16". This affects the following PDR definitions: 28.4 Numeric Sensor PDR 28.11 Numeric Effector PDR

2751  
2752

2753

## Bibliography

2754 DMTF DSP4014, *DMTF Process for Working Bodies 2.6*,  
2755 [http://www.dmtf.org/sites/default/files/standards/documents/DSP4014\\_2.6.pdf](http://www.dmtf.org/sites/default/files/standards/documents/DSP4014_2.6.pdf)

2756