



1
2 **Document Identifier: DSP0218**
3 **Date: 2021-10-27**
4 **Version: 1.1.1**

5 **Platform Level Data Model (PLDM) for Redfish**
6 **Device Enablement**

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

12 Copyright Notice

13 Copyright © 2019-2021 DMTF. All rights reserved.

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

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28 withdrawn or modified after publication, and shall be indemnified and held harmless by any party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

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

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

CONTENTS

36	Foreword	9
37	Acknowledgments	9
38	Introduction	11
39	Document conventions.....	11
40	1 Scope	12
41	2 Normative references	12
42	3 Terms and definitions	14
43	4 Symbols and abbreviated terms	16
44	5 Conventions	17
45	5.1 Reserved and unassigned values	17
46	5.2 Byte ordering	17
47	5.3 PLDM for Redfish Device Enablement data types	17
48	5.3.1 varstring PLDM data type	18
49	5.3.2 schemaClass PLDM data type	18
50	5.3.3 nnint PLDM data type.....	19
51	5.3.4 beiEncoding PLDM data type	19
52	5.3.5 beiTuple PLDM data type	20
53	5.3.6 beiTupleS PLDM data type	20
54	5.3.7 beiTupleF PLDM data type.....	20
55	5.3.8 beiTupleL PLDM data type	21
56	5.3.9 beiTupleV PLDM data type	21
57	5.3.10 beiNull PLDM data type.....	22
58	5.3.11 beiInteger PLDM data type.....	22
59	5.3.12 beiEnum PLDM data type	23
60	5.3.13 beiString PLDM data type	23
61	5.3.14 beiReal PLDM data type	24
62	5.3.15 beiBoolean PLDM data type.....	24
63	5.3.16 beiBytestring PLDM data type	24
64	5.3.17 beiSet PLDM data type.....	25
65	5.3.18 beiArray PLDM data type	25
66	5.3.19 beiChoice data PLDM type	25
67	5.3.20 beiPropertyAnnotation PLDM data type	26
68	5.3.21 beiRegistryItem PLDM data type.....	27
69	5.3.22 beiResourceLink PLDM data type.....	27
70	5.3.23 beiResourceLinkExpansion PLDM data type.....	27
71	5.3.24 beiLocator PLDM data type	27
72	5.3.25 rdeOpID PLDM data type	28
73	6 PLDM for Redfish Device Enablement version.....	28
74	7.1 Redfish Provider architecture overview	29
75	7.1.1 Roles.....	29
76	7.2 Redfish Device Enablement concepts.....	30
77	7.2.1 RDE Device discovery and registration.....	30
78	7.2.2 Data instances of Redfish schemas: Resources.....	32
79	7.2.3 Dictionaries	36
80	7.2.4 Redfish Operation support	43
81	7.2.5 PLDM RDE Events	53
82	7.2.6 Task support	55
83	7.3 Type code	56
84	7.4 Transport protocol type supported	56
85	7.5 Error completion codes.....	56
86	7.6 Timing specification	58

87	8	Binary Encoded JSON (BEJ)	59
88	8.1	BEJ design principles	59
89	8.2	SFLV tuples	60
90	8.2.1	Sequence number	60
91	8.2.2	Format	60
92	8.2.3	Length	60
93	8.2.4	Value	60
94	8.3	Deferred binding of data	61
95	8.4	BEJ encoding	63
96	8.4.1	Conversion of JSON data types to BEJ	63
97	8.4.2	Resource links	64
98	8.4.3	Registry items	64
99	8.4.4	Annotations	64
100	8.4.5	Choice encoding for properties that support multiple data types	66
101	8.4.6	Properties with invalid values	66
102	8.4.7	Properties missing from dictionaries	66
103	8.5	BEJ decoding	66
104	8.5.1	Conversion of BEJ data types to JSON	67
105	8.5.2	Annotations	68
106	8.5.3	Sequence numbers missing from dictionaries	68
107	8.5.4	Sequence numbers for read-only properties in modification Operations	69
108	8.6	Example encoding and decoding	69
109	8.6.1	Example dictionary	69
110	8.6.2	Example encoding	72
111	8.6.3	Example decoding	75
112	8.7	BEJ locators	77
113	9	Operational behaviors	78
114	9.1	Initialization (MC perspective)	78
115	9.1.1	Sample initialization ladder diagram	78
116	9.1.2	Initialization workflow diagram	79
117	9.2	Operation/Task lifecycle	81
118	9.2.1	Example Operation command sequence diagrams	81
119	9.2.2	Operation/Task overview workflow diagrams (Operation perspective)	85
120	9.2.3	RDE Operation state machine (RDE Device perspective)	93
121	9.3	Event lifecycle	106
122	10	PLDM commands for Redfish Device Enablement	108
123	11	PLDM for Redfish Device Enablement – Discovery and schema commands	110
124	11.1	NegotiateRedfishParameters command (0x01) format	110
125	11.2	NegotiateMediumParameters command (0x02) format	112
126	11.3	GetSchemaDictionary command (0x03) format	113
127	11.4	GetSchemaURI command (0x04) format	114
128	11.5	GetResourceETag command (0x05) format	115
129	11.6	GetOEMCount command (0x06) format	116
130	11.7	GetOEMName command (0x07) format	116
131	11.8	GetRegistryCount command (0x08) format	117
132	11.9	GetRegistryDetails command (0x09) format	117
133	11.10	SelectRegistryVersion command (0x0A) format	118
134	11.11	GetMessageRegistry command (0x0B) format	119
135	11.12	GetSchemaFile command (0x0C) format	120
136	12	PLDM for Redfish Device Enablement – RDE Operation and Task commands	121
137	12.1	RDEOperationInit command (0x10) format	121
138	12.2	SupplyCustomRequestParameters command (0x11) format	124
139	12.3	RetrieveCustomResponseParameters command (0x12) format	128
140	12.4	RDEOperationComplete command (0x13) format	129
141	12.5	RDEOperationStatus command (0x14) format	130

142	12.6 RDEOperationKill command (0x15) format	133
143	12.7 RDEOperationEnumerate command (0x16) format	134
144	13 PLDM for Redfish Device Enablement – Utility commands.....	135
145	13.1 RDEMultipartSend command (0x30) format	135
146	13.2 RDEMultipartReceive command (0x31) format.....	137
147	14 Additional Information	139
148	14.1 RDE Multipart transfers	139
149	14.1.1 Flag usage for RDEMultipartSend.....	139
150	14.1.2 Flag usage for RDEMultipartReceive	140
151	14.1.3 RDE Multipart transfer examples	140
152	14.2 Implementation notes	142
153	14.2.1 Schema updates.....	142
154	14.2.2 Storage of dictionaries.....	142
155	14.2.3 Dictionaries for related schemas	142
156	14.2.4 [MC] HTTP/HTTPS POST Operations	143
157	14.2.5 Consistency checking of read Operations.....	144
158	14.2.6 [MC] Placement of RDE Device resources in the outward-facing Redfish URI hierarchy	144
159	14.2.7 LogEntry and LogEntryCollection resources.....	145
160	14.2.8 On-demand pagination	145
161	14.2.9 Considerations for Redfish clients	145
162	14.2.10 OriginOfCondition in Redfish events	146
163	14.2.11 [MC] Merging dictionaries with OEM extensions.....	146
164	ANNEX A (normative) Change log	147
166		

167 Figures

168	Figure 1 – RDE Roles	30
169	Figure 2 – Example linking of Redfish Resource and Redfish Entity Association PDRs	35
170	Figure 3 – Schema linking without Redfish entity association PDRs	36
171	Figure 4 – Dictionary binary format.....	40
172	Figure 5 – DummySimple schema.....	70
173	Figure 6 – DummySimple dictionary – binary form.....	72
174	Figure 7 – Example Initialization ladder diagram.....	79
175	Figure 8 – Typical RDE Device discovery and registration	81
176	Figure 9 – Simple read Operation ladder diagram	82
177	Figure 10 – Complex Read Operation ladder diagram.....	83
178	Figure 11 – Write Operation ladder diagram	84
179	Figure 12 – Write Operation with long-running Task ladder diagram.....	85
180	Figure 13 – RDE Operation lifecycle overview (holistic perspective)	89
181	Figure 14 – RDE Task lifecycle overview (holistic perspective)	92
182	Figure 15 – Operation lifecycle state machine (RDE Device perspective).....	105
183	Figure 16 – Redfish event lifecycle overview	107
184	Figure 17 – RDEMultipartSend example	141
185	Figure 18 – RDEMultipartReceive example	142
186		

187 Tables

188	Table 1 – PLDM for Redfish Device Enablement data types and structures	17
189	Table 2 – varstring data structure	18
190	Table 3 – schemaClass enumeration	19
191	Table 4 – nnint encoding for BEJ	19
192	Table 5 – bejEncoding data structure	19
193	Table 6 – bejTuple encoding for BEJ	20
194	Table 7 – bejTupleS encoding for BEJ	20
195	Table 8 – bejTupleF encoding for BEJ	20
196	Table 9 – BEJ format codes (high nibble: data types)	21
197	Table 10 – bejTupleL encoding for BEJ	21
198	Table 11 – bejTupleV encoding for BEJ	22
199	Table 12 – bejNull value encoding for BEJ	22
200	Table 13 – bejInteger value encoding for BEJ	23
201	Table 14 – bejEnum value encoding for BEJ	23
202	Table 15 – bejString value encoding for BEJ	23
203	Table 15 – bejString special character escape sequences	23
204	Table 16 – bejReal value encoding for BEJ	24
205	Table 17 – bejReal value encoding example	24
206	Table 18 – bejBoolean value encoding for BEJ	24
207	Table 19 – bejBytestring value encoding for BEJ	25
208	Table 20 – bejSet value encoding for BEJ	25
209	Table 21 – bejArray value encoding for BEJ	25

210	Table 22 – bejChoice value encoding for BEJ	25
211	Table 23 – bejPropertyAnnotation value encoding for BEJ	26
212	Table 24 – bejPropertyAnnotation value encoding example	26
213	Table 25 – bejRegistryItem value encoding for BEJ	27
214	Table 25 – bejResourceLink value encoding for BEJ	27
215	Table 26 – bejResourceLinkExpansion value encoding for BEJ	27
216	Table 27 – bejLocator value encoding.....	27
217	Table 28 – rdeOpID data structure	28
218	Table 29 – Redfish dictionary binary format	38
219	Table 30 – Dictionary entry example for a property supporting multiple formats.....	41
220	Table 31 – Redfish Operations	43
221	Table 32 – Redfish operation headers.....	45
222	Table 33 – Redfish operation request query options	50
223	Table 34 – Query parameter support requirement	51
224	Table 34 – PLDM for Redfish Device Enablement completion codes.....	56
225	Table 35 – HTTP codes for standard PLDM completion codes	58
226	Table 36 – Timing specification	58
227	Table 37 – Sequence number dictionary indication.....	60
228	Table 38 – JSON data types supported in BEJ	61
229	Table 39 – BEJ deferred binding substitution parameters	62
230	Table 40 – Message annotation related property BEJ locator encoding.....	65
231	Table 41 – DummySimple dictionary (tabular form)	70
232	Table 42 – Initialization Workflow	80
233	Table 43 – Operation lifecycle overview	86
234	Table 44 – Task lifecycle overview	90
235	Table 45 – Task lifecycle state machine.....	94
236	Table 46 – Event lifecycle overview.....	106
237	Table 47 – PLDM for Redfish Device Enablement command codes	108
238	Table 48 – NegotiateRedfishParameters command format	111
239	Table 49 – NegotiateMediumParameters command format.....	113
240	Table 50 – GetSchemaDictionary command format.....	114
241	Table 51 – GetSchemaURI command format	114
242	Table 52 – GetResourceETag command format.....	116
243	Table 52 – GetOEMCount command format	116
244	Table 52 – GetOEMName command format	117
245	Table 52 – GetRegistryCount command format	117
246	Table 52 – GetRegistryDetails command format.....	118
247	Table 50 – SelectRegistryVersion command format	119
248	Table 50 – GetMessageRegistry command format	119
249	Table 50 – GetSchemaFile command format.....	120
250	Table 53 – RDEOperationInit command format	122
251	Table 54 – SupplyCustomRequestParameters command format	125
252	Table 55 – RetrieveCustomResponseParameters command format.....	129
253	Table 56 – RDEOperationComplete command format.....	130
254	Table 57 – RDEOperationStatus command format	130
255	Table 58 – RDEOperationKill command format	133
256	Table 59 – RDEOperationEnumerate command format	135
257	Table 60 – RDEMultipartSend command format.....	136

258	Table 61 – RDEMultipartReceive command format	138
259		

260

Foreword

261 The *Platform Level Data Model (PLDM)* for Redfish Device Enablement (DSP0218) was prepared by the
262 PMCI (Platform Management Components Intercommunications) Working Group of the DMTF.

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

Acknowledgments

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

267 Editors:

- 268 • Balaji Natrajan – Microchip Technology Inc.
269 • Bill Scherer – Hewlett Packard Enterprise

270 Contributors:

- 271 • Richelle Ahlvers – Broadcom Inc.
272 • Jeff Autor – Hewlett Packard Enterprise
273 • Patrick Caporale – Lenovo
274 • Mike Garrett – Hewlett Packard Enterprise
275 • Jeff Hilland – Hewlett Packard Enterprise
276 • Yuval Itkin –NVIDIA Corporation
277 • Ira Kalman – Intel
278 • Deepak Kodihalli – IBM
279 • Eliel Louzoun – Intel
280 • Ben Lytle – Hewlett Packard Enterprise
281 • Rob Mapes – Marvell
282 • Balaji Natrajan – Microchip Technology Inc.
283 • Edward Newman – Hewlett Packard Enterprise
284 • Zvika Perry Peleg – Cavium
285 • Scott Phuong – Cisco Systems, Inc.
286 • Jeffrey Plank – Microchip Technology Inc.
287 • Joey Rainville – Hewlett Packard Enterprise
288 • Patrick Schoeller – Hewlett Packard Enterprise
289 • Hemal Shah – Broadcom Inc.
290 • Bob Stevens – Dell Inc.
291 • Richard Thomaiyar – Intel
292 • Bill Vetter – Lenovo
293 • Ryan Weldon – Marvell

295

Introduction

296 The *Platform Level Data Model (PLDM) for Redfish Device Enablement Specification* defines messages
297 and data structures used for enabling PLDM-capable devices to participate in Redfish-based
298 management without needing to support either JavaScript Object Notation (JSON, used for operation
299 data payloads) or [Secure] Hypertext Transfer Protocol (HTTP/HTTPS, used to transport and configure
300 operations). This document specifies how to convert Redfish operations into a compact binary-encoded
301 JSON (BEJ) format transported over PLDM, including the encoding and decoding of JSON and the
302 manner in which HTTP/HTTPS headers and query options may be supported under PLDM. In this
303 specification, Redfish management functionality is divided between the three roles: the client, which
304 initiates management operations; the RDE Device, which ultimately services requests; and the
305 management controller (MC), which translates requests and serves as an intermediary between the client
306 and the RDE Device.

307 **Document conventions**

308 **Clause naming conventions**

309 While all clauses of this specification are relevant from the perspective of both MCs and RDE Devices, a
310 few clauses are primarily targeted at one or the other. This document uses the following naming
311 conventions for clauses:

- 312 • The titles of clauses that are primarily of interest to MCs are prefixed with “[MC]”.
- 313 • The titles of clauses that are primarily of interest to RDE Devices are prefixed with “[Dev]”
- 314 • Unless explicitly marked, the subclauses of a clause marked as being primarily of interest to
315 one role are also primarily of interest to that same role
- 316 • Clauses that are of primary interest to more than one role are not prefixed

317 NOTE This specification is designed such that clients have no need to be aware whether the RDE Device whose
318 data they are interacting with is supporting Redfish directly or through an MC proxy.

319 **Typographical conventions**

320 The following typographical conventions are used in this document:

- 321 • Document titles are marked in *italics*.

Platform Level Data Model (PLDM) for Redfish Device Enablement

1 Scope

This specification defines messages and data structures used for enabling PLDM devices to participate in Redfish-based management without needing to support either JavaScript Object Notation (JSON, used for operation data payloads) or [Secure] Hypertext Transfer Protocol (HTTP/HTTPS, used to transport and configure operations). This document specifies how to convert Redfish operations into a compact binary-encoded JSON (BEJ) format transported over PLDM, including the encoding and decoding of JSON and the manner in which HTTP/HTTPS headers and query options shall be supported under PLDM. This document does not specify the resources (data models) for use with RDE Devices or any details of handling the Redfish security model. Transferring firmware images is not intended to be within the scope of this specification as this function is the primary scope of [DSP0267](#), the PLDM for Firmware Update specification.

In this specification, Redfish management functionality is divided between the three roles: the client, which initiates management operations; the RDE Device, which ultimately services requests; and the management controller (MC), which translates requests and serves as an intermediary between the client and the RDE Device. Of these roles, the RDE Device and MC roles receive extensive treatment in this specification; however, the client role is no different from standard Redfish. An implementer of this specification is only required to support the features of one of the RDE Device or MC roles. In particular, an RDE Device is not required to implement MC-specific features and vice versa.

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 shall support Redfish Device Enablement over PLDM. However, if a system does support Redfish Device Enablement over PLDM or other functions described in this specification, the specification defines the requirements to access and use those functions over PLDM.

Portions of this specification rely on information and definitions from other specifications, which are identified in clause **Error! Reference source not found.**. Several of these references are particularly relevant:

- DMTF [DSP0266](#), *Redfish Scalable Platforms Management API Specification Redfish Scalable Platforms Management API Specification*, defines the main Redfish protocols.
- 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 [DSP0245](#), *Platform Level Data Model (PLDM) IDs and Codes Specification*, defines the values that are used to represent different type codes defined for PLDM messages.
- DMTF [DSP0248](#), *Platform Level Data Model (PLDM) for Platform Monitoring and Control Specification*, defines the event and Redfish PDR data structures referenced in this specification.

362 2 Normative references

- 363 The following referenced documents are indispensable for the application of this document. For dated or
364 versioned references, only the edition cited (including any corrigenda or DMTF update versions) applies.
365 For references without a date or version, the latest published edition of the referenced document
366 (including any corrigenda or DMTF update versions) applies. Earlier versions may not provide sufficient
367 support for this specification.
- 368 DMTF DSP0222, *Network Controller Sideband Interface (NC-SI) Specification 1.1*,
369 https://www.dmtf.org/sites/default/files/standards/documents/DSP0222_1.1.pdf
- 370 DMTF DSP0236, *MCTP Base Specification 1.2*,
371 http://dmtf.org/sites/default/files/standards/documents/DSP0236_1.2.pdf
- 372 DMTF DSP0240, *Platform Level Data Model (PLDM) Base Specification 1.1*,
373 http://dmtf.org/sites/default/files/standards/documents/DSP0240_1.1.pdf
- 374 DMTF DSP0241, *Platform Level Data Model (PLDM) Over MCTP Binding Specification 1.0*,
375 http://dmtf.org/sites/default/files/standards/documents/DSP0241_1.0.pdf
- 376 DMTF DSP0245, *Platform Level Data Model (PLDM) IDs and Codes Specification 1.3*,
377 http://dmtf.org/sites/default/files/standards/documents/DSP0245_1.3.pdf
- 378 DMTF DSP0248, *Platform Level Data Model (PLDM) for Platform Monitoring and Control Specification*
379 1.1, http://dmtf.org/sites/default/files/standards/documents/DSP0248_1.1.pdf
- 380 DMTF DSP0266, *Redfish Scalable Platforms Management API Specification 1.6*,
381 http://www.dmtf.org/sites/default/files/standards/documents/DSP0266_1.6.pdf
- 382 DMTF DSP0267, *PLDM for Firmware Update Specification 1.0*,
383 https://www.dmtf.org/sites/default/files/standards/documents/DSP0267_1.0.pdf
- 384 DMTF DSP4004, *DMTF Release Process 2.4*,
385 http://dmtf.org/sites/default/files/standards/documents/DSP4004_2.4.pdf
- 386 ECMA International Standard ECMA-404, *The JSON Data Interchange Syntax*, <http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf>
- 388 IETF RFC2781, *UTF-16, an encoding of ISO 10646*, February 2000,
389 <http://www.ietf.org/rfc/rfc2781.txt>
- 390 IETF STD63, *UTF-8, a transformation format of ISO 10646* <http://www.ietf.org/rfc/std/std63.txt>
- 391 IETF RFC4122, *A Universally Unique Identifier (UUID) URN Namespace*, July 2005,
392 <http://www.ietf.org/rfc/rfc4122.txt>
- 393 IETF RFC4646, *Tags for Identifying Languages*, September 2006,
394 <http://www.ietf.org/rfc/rfc4646.txt>

- 395 IETF RFC7231, R. Fielding et al., *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*,
396 <https://tools.ietf.org/html/rfc7231>
- 397 IETF RFC 7232, R. Fielding et al., *Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests*,
398 <http://www.ietf.org/rfc/rfc7232.txt>
- 399 IETF RFC 7234, R. Fielding et al., *Hypertext Transfer Protocol (HTTP/1.1): Caching*,
400 <https://tools.ietf.org/rfc/rfc7234.txt>
- 401 ISO 8859-1, *Final Text of DIS 8859-1, 8-bit single-byte coded graphic character sets — Part 1: Latin alphabet No. 1*, February 1998
- 402 ISO/IEC Directives, Part 2, *Rules for the structure and drafting of International Standards*,
<http://isotc.iso.org/livelink/livelink.exe?func=ll&objId=4230456&objAction=browse&sort=subtype>
- 405 ITU-T X.690 (08/2015), *Information technology – ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)*,
406 <http://handle.itu.int/11.1002/1000/12483>
- 408 Open Data Protocol, <https://www.oasis-open.org/standards#odatav4.0>

409 3 Terms and definitions

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

412 The terms "shall" ("required"), "shall not", "should" ("recommended"), "should not" ("not recommended"),
413 "may", "need not" ("not required"), "can" and "cannot" in this document are to be interpreted as described
414 in [ISO/IEC Directives, Part 2](#), Clause 7. The terms in parentheses are alternatives for the preceding term,
415 for use in exceptional cases when the preceding term cannot be used for linguistic reasons. Note that
416 [ISO/IEC Directives, Part 2](#), Clause 7 specifies additional alternatives. Occurrences of such additional
417 alternatives shall be interpreted in their normal English meaning.

418 The terms "clause", "subclause", "paragraph", and "annex" in this document are to be interpreted as
419 described in [ISO/IEC Directives, Part 2](#), Clause 6.

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

423 Refer to [DSP0240](#) for terms and definitions that are used across the PLDM specifications, [DSP0248](#) for
424 terms and definitions used specifically for PLDM Monitoring and Control, and to [DSP0266](#) for terms and
425 definitions specific to Redfish. For the purposes of this document, the following additional terms and
426 definitions apply.

427 3.1

428 Action

429 Any standard Redfish action defined in a standard Redfish Schema or any custom OEM action defined in
430 an OEM schema extension

431 3.2

432 Annotation

433 Any of several pieces of metadata contained within BEJ or JSON data. Rather than being defined as part
434 of the major schema, annotations are defined in a separate, global annotation schema.

435 **3.3**

436 **Client**

437 Any agent that communicates with a management controller to enable a user to manage Redfish-
438 compliant systems and RDE Devices

439 **3.4**

440 **Collection**

441 A Redfish container holding an array of independent Redfish resource Members that in turn are typically
442 represented by a schema external to the one that contains the collection itself.

443 **3.5**

444 **Device Component**

445 A top-level entry point into the schema hierarchy presented by an RDE Device

446 **3.6**

447 **Dictionary**

448 A binary lookup table containing translation information that allows conversion between BEJ and JSON
449 formats of data for a given resource

450 **3.7**

451 **Discovery**

452 The process by which an MC determines that an RDE Device supports PLDM for Redfish Device
453 Enablement

454 **3.8**

455 **Major Schema**

456 The primary schema defining the format of a collection of data, usually a published standard Redfish
457 schema.

458 **3.9**

459 **Member**

460 Any of the independent resources contained within a collection

461 **3.10**

462 **Metadata**

463 Information that describes data of interest, such as its type format, length in bytes, or encoding method

464 **3.11**

465 **OData**

466 The [Open Data protocol](#), a source of annotations in Redfish, as defined by OASIS.

467 **3.12**

468 **OEM Extension**

469 Any manufacturer-specific addition to major schema

470 **3.13**

471 **Property**

472 An individual datum contained within a Resource

473 **3.14**

474 **RDE Device**

475 Any PLDM terminus containing an RDE Provider that requires the intervention of an MC to receive
476 Redfish communications

477 **3.15**

478 **RDE Provider**

479 Any RDE Device that responds to RDE Operations. See also **Redfish Provider**.

480 **3.16**

481 **RDE Operation**

482 The sequence of PLDM messages and operations that represent a Redfish Operation being executed by
483 an MC and/or an RDE Device on behalf of a client. See also **Redfish Operation**.

484 **3.17**

485 **Redfish Operation**

486 Any Redfish operation transmitted via HTTP or HTTPS from a client to an MC for execution. See also
487 **RDE Operation**.

488 **3.18**

489 **Redfish Provider**

490 Any entity that responds to Redfish Operations. See also **RDE Provider**.

491 **3.19**

492 **Registration**

493 The process of enabling a compliant RDE Device with an MC to be an RDE Provider

494 **3.20**

495 **Resource**

496 A hierarchical set of data organized in the format specified in a Redfish Schema.

497 **3.21**

498 **Schema**

499 Any regular structure for organizing one or more fields of data in a hierarchical format

500 **3.22**

501 **Task**

502 Any Operation for which an RDE Device cannot complete execution in the time allotted to respond to the
503 PLDM triggering command message sent from the MC and for which the MC creates standard Redfish
504 Task and TaskMonitor objects

505 **3.23**

506 **Triggering Command**

507 The PLDM command that supplies the last bit of data needed for an RDE Device to begin execution of an
508 RDE Operation

509 **3.24**

510 **Truncated**

511 When applied to a dictionary, one that is limited to containing conversion information for properties
512 supported by an RDE Device

513

514 4 Symbols and abbreviated terms

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

517 **4.1**

518 **BEJ**

519 Binary Encoded JSON, a compressed binary format for encoding JSON data

520 **4.2**

521 **JSON**

522 JavaScript Object Notation

523 **4.3**

524 **RDE**

525 Redfish Device Enablement

526 5 Conventions

527 Refer to [DSP0240](#) for conventions, notations, and data types that are used across the PLDM
528 specifications.

529 5.1 Reserved and unassigned values

530 Unless otherwise specified, any reserved, unspecified, or unassigned values in enumerations or other
531 numeric ranges are reserved for future definition by the DMTF.

532 Unless otherwise specified, numeric or bit fields that are designated as reserved shall be written as 0
533 (zero) and ignored when read.

534 5.2 Byte ordering

535 As with all PLDM specifications, unless otherwise specified, the byte ordering of multibyte numeric fields
536 or multibyte bit fields in this specification shall be "Little Endian": The lowest byte offset holds the least
537 significant byte and higher offsets hold the more significant bytes.

538 5.3 PLDM for Redfish Device Enablement data types

539 **Table 1** lists additional abbreviations and descriptions for data types that are used in message field and
540 data structure definitions in this specification.

541 **Table 1 – PLDM for Redfish Device Enablement data types and structures**

Data Type	Interpretation
varstring	A multiformat text string per clause 5.3.1
schemaClass	An enumeration of the various schemas associated with a collection of data, encoded per clause 5.3.2
nnint	A nonnegative integer encoded for BEJ per clause 5.3.3
bejEncoding	JSON data encoded for BEJ per clause 5.3.4
bejTuple	A BEJ tuple, encoded per clause 5.3.5

Data Type	Interpretation
bejTupleS	A BEJ Sequence Number tuple element, encoded per clause 5.3.6
bejTupleF	A BEJ Format tuple element, encoded per clause 5.3.7
bejTupleL	A BEJ Length tuple element, encoded per clause 5.3.8
bejTupleV	A BEJ Value tuple element, encoded per clause 5.3.9
bejNull	Null data encoded for BEJ per clause 5.3.10
bejInteger	Integer data encoded for BEJ per clause 5.3.11
bejEnum	Enumeration data encoded for BEJ per clause 5.3.12
bejString	String data encoded for BEJ per clause 5.3.13
bejReal	Real data encoded for BEJ per clause 0
bejBoolean	Boolean data encoded for BEJ per clause 5.3.15
bejBytestring	Bytestring data encoded for BEJ per clause 5.3.16
bejSet	Set data encoded for BEJ per clause 5.3.17
bejArray	Array data encoded for BEJ per clause 5.3.18
bejChoice	Choice data encoded for BEJ per clause 5.3.19
bejPropertyAnnotation	Property Annotation encoded for BEJ per clause 5.3.20
bejRegistryItem	A Redfish Registry Message encoded for BEJ per clause 5.3.21
bejResourceLink	Resource Link data encoded for BEJ per clause 5.3.22
bejResourceLinkExpansion	Resource Link data expanded to include schema data encoded for BEJ per clause 5.3.23
bejLocator	An intra-schema locator for Operation targeting; formatted per clause 5.3.24
rdeOpID	An Operation identifier used to link together the various command messages that comprise a single RDE Operation; formatted per clause 5.3.25

542 **5.3.1 varstring PLDM data type**

543 The varstring PLDM data type encapsulates a PLDM string that can be encoded in of any of several
 544 formats.

545 **Table 2 – varstring data structure**

Type	Description
enum8	stringFormat Values: { UNKNOWN = 0, ASCII = 1, UTF-8 = 2, UTF-16 = 3, UTF-16LE = 4, UTF-16BE = 5 }
uint8	stringLengthBytes Including null terminator
variable	stringData Must be null terminated

546 **5.3.2 schemaClass PLDM data type**

547 The schemaClass PLDM data type enumerates the different categories of schemas used in Redfish. RDE
 548 uses 5 main classes of schemas:

- 549 • MAJOR: the main schema containing the data for a Redfish resource. This class covers the
 vast majority of schemas for Redfish resources.
- 551 • EVENT: the standard DMTF-published event schema, for occurrences that clients may wish to
 be notified about.
- 553 • ANNOTATION: the standard DMTF-published annotation schema that captures metadata about
 a major schema or payload. This schemaClass shall not be used as the primary schema for
 BEJ encodings as annotations are specially encoded alongside the primary schema.
- 556 • COLLECTION_MEMBER_TYPE: for resources that correspond to Redfish collections, this
 class enables access to the major schema for members of that collection from the context of the
 collection resource. (Unlike regular resources, collections in Redfish are unversioned and
 contain multiple members.) This schemaClass shall not be used for BEJ encodings.
- 560 • ERROR: the standard DMTF-published error schema that documents an extended error when a
 Redfish operation cannot be completed.
- 562 • REGISTRY: A device-specific collection of Redfish registry messages used for errors and
 events. This schemaClass shall not be used as the primary schema for BEJ encodings as
 registry items are specially encoded alongside the primary schema via the bejRegistryItem type
 (see 5.3.21).

566 **Table 3 – schemaClass enumeration**

Type	Description
enum8	schemaType Values: { MAJOR = 0, EVENT = 1, ANNOTATION = 2, COLLECTION_MEMBER_TYPE = 3, ERROR = 4, REGISTRY = 5 }

567 **5.3.3 nnint PLDM data type**

568 The nnint PLDM data type captures the BEJ encoding of nonnegative Integers via the following encoding:

569 The first byte shall consist of metadata for the number of bytes needed to encode the numeric value in
 570 the remaining bytes. Subsequent bytes shall contain the encoded value in little-endian format. As
 571 examples, the value 65 shall be encoded as 0x01 0x41; the value 130 shall be encoded as 0x01 0x82;
 572 and the value 1337 shall be encoded as 0x02 0x39 0x05.

573 **Table 4 – nnint encoding for BEJ**

Type	Description
uint8	Length (N) in bytes of data for the integer to be encoded
uint8	Integer data [0] (Least significant byte)
uint8	Integer data [1] (Second least significant byte)
...	...
uint8	Integer data [N-1] (Most significant byte)

574 **5.3.4 bejEncoding PLDM data type**

575 The bejEncoding PLDM data type captures an overall hierarchical BEJ-encoded block of hierarchical
576 data.

577 **Table 5 – bejEncoding data structure**

Type	Description
ver32	BEJ Version; shall be either 1.0.0 (0xF1F0F000) or 1.1.0 (0xF1F1F000) for this specification. The actual version represented shall be at least as high as the minimum version required to support any BEJ PLDM data types included in the encoding. All BEJ PLDM data types are version 1.0 unless explicitly marked as requiring a higher version. RDE devices shall not use a BEJ encoding version higher than that supported by the MC. MCs shall not use a BEJ encoding version with an RDE Device higher than the version supported by that device.
uint16	Reserved for BEJ flags
schemaClass	Defines the primary schema type for the data encoded in bejTuple below. Shall be one of MAJOR, EVENT, or ERROR.
bejTuple	The encoded tuple data, defined in clause 5.3.5

578 **5.3.5 bejTuple PLDM data type**

579 The bejTuple PLDM data type encapsulates all the data for a single piece of data encoded in BEJ format.

580 **Table 6 – bejTuple encoding for BEJ**

Type	Description
bejTupleS	Tuple element for the Sequence Number field, defined in clause 5.3.6 and described in clause 8.2.1
bejTupleF	Tuple element for the Format field, defined in clause 5.3.7 and described in clause 8.2.2
bejTupleL	Tuple element for the Length field, defined in clause 5.3.8 and described in clause 8.2.3
bejTupleV	Tuple element for the Value field, defined in clause 5.3.9 and described in clause 8.2.4

581 **5.3.6 bejTupleS PLDM data type**

582 The bejTupleS PLDM data type captures the Sequence Number BEJ tuple element described in clause
583 8.2.1.

584 **Table 7 – bejTupleS encoding for BEJ**

Type	Description
nnint	Sequence number indicating the specific data item contained within this tuple. The sequence number is encoded as a nonnegative integer (nnint type) and is enhanced to indicate the dictionary to which it refers. More specifically, the low-order bit of the encoded integer is metadata used to select the dictionary within which the property encoded in the tuple may be found, and shall be one of the following values: 0b: Primary schema (including any OEM extensions) dictionary as was selected in the outermost bejEncoding PLDM data type element containing this bejTupleS 1b: Annotation schema dictionary The remainder of the integer corresponds to the sequence number encoded in the dictionary. Dictionary encodings do not include the dictionary selector flag bit.

585 **5.3.7 bejTupleF PLDM data type**

586 The bejTupleF PLDM data type captures the Format BEJ tuple element described in clause 8.2.2

587 **Table 8 – bejTupleF encoding for BEJ**

Type	Description
bitfield8	<p>Format code; the high nibble represents the data type and the low nibble represents a series of flag bits</p> <ul style="list-style-type: none"> [7:4] - principal data type; see Table 9 below for values [3] - reserved flag. 1b indicates the flag is set [2] - nullable_property flag ***. 1b indicates the flag is set [1] - read_only_property_and_top_level_annotation flag **. 1b indicates the flag is set [0] - deferred_binding flag *. 1b indicates the flag is set

588 * The deferred_binding flag shall only be set in conjunction with BEJ String data and shall never be set
589 when encoding the format of a property inside a dictionary. See clause 8.3.590 ** The nullable property flag shall only be set when encoding the format of a property inside a dictionary.
591 See clause 7.2.3.2.592 *** The read_only_property_and_top_level_annotation flag has distinct meanings when in or not in the
593 context of a dictionary. In a dictionary, it means that a property is read-only. See clause 7.2.3.2. In a BEJ
594 encoding, it marks a nested top-level annotation. See clause 8.4.4.1. Decoding context thus uniquely
595 determines the meaning of this flag bit.596 **Table 9 – BEJ format codes (high nibble: data types)**

Code	BEJ Type	PLDM Type in Value Tuple Field *
0000b	BEJ Set	bejSet
0001b	BEJ Array	bejArray
0010b	BEJ Null	bejNull
0011b	BEJ Integer	bejInteger
0100b	BEJ Enum	bejEnum
0101b	BEJ String	bejString
0110b	BEJ Real	bejReal
0111b	BEJ Boolean	bejBoolean
1000b	BEJ Bytestring	bejBytestring
1001b	BEJ Choice	bejChoice
1010b	BEJ Property Annotation	bejPropertyAnnotation
1011b	BEJ Registry Item	bejRegistryItem
1100b – 1101b	Reserved	n/a
1110b	BEJ Resource Link	bejResourceLink
1111b	BEJ Resource Link Expansion	bejResourceLinkExpansion

597 **5.3.8 bejTupleL PLDM data type**

598 The bejTupleL PLDM data type captures the Length BEJ tuple element described in clause 8.2.3.

599

Table 10 – bejTupleL encoding for BEJ

Type	Description
nnint	Length in bytes of value tuple field

600 **5.3.9 bejTupleV PLDM data type**

601 The bejTupleV PLDM data type captures the Value BEJ tuple element described in clause 8.2.4.

602 **Table 11 – bejTupleV encoding for BEJ**

Type	Description
bejNull, bejInteger, bejEnum, bejString, bejReal, bejBoolean, bejBytestring, bejSet, bejArray, bejChoice, bejPropertyAnnotation, bejResourceLink, or bejResourceLinkExpansion	Value tuple element; exact type shall match that of the Format tuple element contained within the same tuple per Table 9 . For example, if a tuple has 0011b (BEJ Integer) as the Format tuple element, then the data encoded in the value tuple element will be of type bejInteger.

603 **5.3.10 bejNull PLDM data type**

604 The length tuple value for bejNull data shall be zero.

605 **Table 12 – bejNull value encoding for BEJ**

Type	Description
(none)	No fields

606 **5.3.11 bejInteger PLDM data type**607 Integer data shall be encoded as the shortest sequence of bytes (little endian) that represent the value in
608 two's complement encoding. This implies that if the value is positive and the high bit (0x80) of the MSB in
609 an unsigned representation would be set, the unsigned value will be prefixed with a new null (0x00) MSB
610 to mark the value as explicitly positive.
611

612

Table 13 – bejInteger value encoding for BEJ

Type	Description
uint8	Data [0] (Least significant byte of twos complement encoding of integer)
uint8	Data [1] (Second least significant byte of twos complement encoding of integer)
...	...
uint8	Data [N-1] (Most significant byte of twos complement encoding of integer)

613 **5.3.12 bejEnum PLDM data type**

614

Table 14 – bejEnum value encoding for BEJ

Type	Description
nnint	Integer value of the sequence number for the enumeration option selected

615 **5.3.13 bejString PLDM data type**

616 All BEJ strings shall be UTF-8 encoded and null-terminated.

617

Table 15 – bejString value encoding for BEJ

Type	Description
uint8	Data [0] (First character of string data)
uint8	Data [1] (Second character of string data)
...	...
uint8	Data [N-1] (Last character of string data)
uint8	Null terminator 0x00

618 The special characters that require escaping in JSON format shall also be escaped in bejString
619 encodings, using the backslash character ('\'):

620

Table 16 – bejString special character escape sequences

Character	Escape sequence
Double quote	\"
Backslash	\\"
Forward slash	\/
Backspace	\b
Form feed	\f
Line feed	\n
Carriage return	\r

621 NOTE Missing escape characters will likely cause JSON text to be malformed. RDE Devices and MCs should
622 validate correctness of BEJ String data to avoid this occurrence.

623 **5.3.14 bejReal PLDM data type**624 BEJ encoding for *whole*, *fract*, and *exp* that represent the base 10 encoding *whole.fract* × 10^{*exp*}.625 NOTE There is no need to express special values (positive infinity, negative infinity, NaN, negative zero) because
626 these cannot be expressed in JSON.627 **Table 17 – bejReal value encoding for BEJ**

Type	Description
nnint	Length of <i>whole</i>
bejInteger	<i>whole</i> (includes sign for the overall real number)
nnint	Leading zero count for <i>fract</i>
nnint	<i>fract</i>
nnint	Length of <i>exp</i>
bejInteger	<i>exp</i> (includes sign for the exponent)

628 In order to distinguish between the cases where the exponent is zero and the exponent is omitted
629 entirely, an omitted exponent shall be encoded with a length of zero bytes; the exponent of zero shall be
630 encoded with a single byte (of value zero). (These cases are numerically identical but visually distinct in
631 standard text-based JSON encoding.)632 As an example, **Table 18** shows the encoding of the JSON number “1.0005e+10”:633 **Table 18 – bejReal value encoding example**

Type	Bytes	Description
nnint	0x01 0x01	Length of <i>whole</i> (1 byte)
bejInteger	0x01	<i>whole</i> (1)
nnint	0x01 0x03	leading zero count for <i>fract</i> (3)
nnint	0x01 0x05	<i>fract</i> (5)
nnint	0x01 0x01	Length of <i>exp</i> (1)
bejInteger	0xA	<i>Exp</i> (10)

634 **5.3.15 bejBoolean PLDM data type**

635 The bejBoolean PLDM data type captures boolean data.

636 **Table 19 – bejBoolean value encoding for BEJ**

Type	Description
uint8	Boolean value { 0x00 = logical false, all other = logical true }

637 **5.3.16 bejBytestring PLDM data type**638 The bejBytestring PLDM data type captures a generic ordered sequence of bytes. As binary data and not
639 a true string type, no null terminator should be applied.

640

Table 20 – bejBytestring value encoding for BEJ

Type	Description
uint8	Data [0] (First byte of string data)
uint8	Data [1] (Second byte of string data)
...	...
uint8	Data [N-1] (Last byte of string data)

641 5.3.17 bejSet PLDM data type

642 The bejSet PLDM data type captures a JSON Object that in turn gathers a series of properties that may
 643 be of disparate types.

644

Table 21 – bejSet value encoding for BEJ

Type	Description
nnint	Count of set elements
bejTuple	First set element
bejTuple	Second set element
...	...
bejTuple	N th set element (N = Count)

645 5.3.18 bejArray PLDM data type

646 The bejArray PLDM data type captures a JSON Array that in turn gathers an ordered sequence of
 647 properties all of a common type.

648

Table 22 – bejArray value encoding for BEJ

Type	Description
nnint	Count of array elements
bejTuple	First array element
bejTuple	Second array element
...	...
bejTuple	N th array element (N = Count)

649 5.3.19 bejChoice data PLDM type

650 The bejChoice PLDM data type captures JSON data encoded when it can be of multiple formats.
 651 Inserting the bejChoice PLDM type alerts a decoding process that multiformat data is coming up in the
 652 BEJ datastream.

653

Table 23 – bejChoice value encoding for BEJ

Type	Description
bejTuple	Selected option

654 **5.3.20 bejPropertyAnnotation PLDM data type**

655 The bejPropertyAnnotation PLDM data type captures the encoding of a property annotation in the form
 656 property@annotationtype.annotationname. When the bejTupleF format code is set to
 657 bejPropertyAnnotation, the sequence number bejTupleS in the outer bejTuple shall be for the annotated
 658 property. The value bejTupleV of the outer bejTuple shall be as follows:

659 **Table 24 – bejPropertyAnnotation value encoding for BEJ**

Type	Description
bejTupleS	Sequence number for annotation property name, including the schema selector bit to mark this as being from the annotation dictionary, as defined in clause 5.3.6
bejTupleF	Format for annotation data applying to the property indicated by the sequence number above, as defined in clause 5.3.7. Implementers should be aware that this format need not match the format for the annotated property.
bejTupleL	Length in bytes of data in the bejTupleV field following, as defined in clause 5.3.8
bejTupleV	Annotation data applying to the property indicated by the sequence number above, as defined in clause 5.3.9

660 As an example, **Table 25** shows the encoding of the annotation:

661 "Status@Redfish.RequiredOnCreate" : false

662 **Table 25 – bejPropertyAnnotation value encoding example**

Type	Bytes	Description
bejTupleS	0x01 0x12	Sequence number for "Status" in the current schema. The low-order bit is clear to show that this sequence number is not from the annotation dictionary. Note The actual sequence number provided here is for illustrative purposes only and may not reflect the current number for "Status" in any particular dictionary
bejTupleF	0x0A	BEJ Property Annotation
bejTupleL	0x01 0x06	Length of the annotation data. The remaining entries in this table correspond to the bejTupleV entry, which in this case is the Boolean RequiredOnCreate data.
Note: The remaining rows shown in this example are collectively the bejTupleV field for the first tuple above.		
bejTupleS	0x01 0x27	Sequence number for "Redfish.RequiredOnCreate", The low-order bit is set to mark this sequence number as being from the annotation dictionary. Note The actual sequence number provided here is for illustrative purposes only and may not reflect the current number for "Redfish.RequiredOnCreate"
bejTupleF	0x01	BEJ boolean
bejTupleL	0x01 0x01	Length of the annotation value: one byte
bejTupleV	0x00	False

663 **5.3.21 bejRegistryItem PLDM data type**

664 The bejRegistryItem PLDM data type represents a registry message item, referenced in another schema
 665 such as event, error, or message. The bejRegistryItem PLDM data type requires BEJ version 1.1.0 (see
 666 clause 5.3.4).

667 **Table 26 – bejRegistryItem value encoding for BEJ**

Type	Description
bejTupleS	The sequence number for a message item from the registry dictionary. This sequence number shall be interpreted as from the registry dictionary, NOT from the primary schema for the enclosing bejEncoding

668 **5.3.22 bejResourceLink PLDM data type**

669 The bejResourceLink PLDM data type represents the URI that links to another Redfish Resource,
 670 specified via a resource ID for the target Redfish Resource PDR. When the bejTupleF format code is set
 671 to BEJ Resource Link in BEJ-encoded data, the four bejTupleF flag bits shall each be 0b.

672 **Table 27 – bejResourceLink value encoding for BEJ**

Type	Description
nnint	ResourceId of Redfish Resource PDR for linked schema

673 **5.3.23 bejResourceLinkExpansion PLDM data type**

674 The bejResourceLinkExpansion PLDM data type captures a link to another Redfish Resource, such as a
 675 related Redfish resource, that is expanded inline in response to a \$expand Redfish request query
 676 parameter (see clause 7.2.4.3.3). When the bejTupleF format code is set to BEJ Resource Link
 677 Expansion in BEJ-encoded data, the bejTupleF flag bits must not be set.

678 **Table 28 – bejResourceLinkExpansion value encoding for BEJ**

Type	Description
nnint	ResourceId of Redfish Resource PDR for linked schema
bejEncoding	BEJ data for expanded resource

679 **5.3.24 bejLocator PLDM data type**

680 The use of BEJ locators is detailed in clause 8.7. All sequence numbers within a BEJ locator shall
 681 reference the same schema dictionary. As each of the sequence numbers is of potentially different length,
 682 reading a sequence number in a BEJ locator must be done by first reading all previous sequence
 683 numbers in the locator. As is standard for BEJ sequence number assignment, if sequence number M
 684 corresponds to an array, sequence number M + 1 (if present) will correspond to a zero-based index within
 685 the array.

686 **Table 29 – bejLocator value encoding**

Type	Description
nnint	LengthBytes Total length in bytes of the N sequence numbers comprising this locator

Type	Description
bejTupleS	Sequence number [0]
bejTupleS	Sequence number [1]
bejTupleS	Sequence number [2]
...	...
bejTupleS	Sequence number [N - 1]

687 **5.3.25 rdeOpID PLDM data type**

688 The rdeOpID PLDM data type is an Operation identifier that can be used to link together the various
689 command messages that comprise a single RDE Operation.

690 **Table 30 – rdeOpID data structure**

Type	Description
uint16	OperationIdentifier Numeric identifier for the Operation. Operation identifiers with the most significant bit set (1b) are reserved for use by the MC when it instantiates Operations. Operation identifiers with the most significant bit clear (0b) are reserved for use by the RDE Device when it instantiates Operations in response to commands from other protocols that it chooses to make visible via RDE. The value 0x0000 is reserved to indicate no Operation.

691 **6 PLDM for Redfish Device Enablement version**

692 The version of this Platform Level Data Model (PLDM) for Redfish Device Enablement Specification shall
693 be 1.1.0 (major version number 1, minor version number 1, update version number 0, and no alpha
694 version).

695 In response to the GetPLDMVersion command described in [DSP0240](#), the reported version for Type 6
696 (PLDM for Redfish Device Enablement, this specification) shall be encoded as 0xF1F1F000.

697 **7 PLDM for Redfish Device Enablement overview**

698 This specification describes the operation and format of request messages (also referred to as
699 commands) and response messages for performing Redfish management of RDE Devices contained
700 within a platform management subsystem. These messages are designed to be delivered using PLDM
701 messaging.

702 Traditionally, management has been affected via myriad proprietary approaches for limited classes of
703 devices. These disparate solutions differ in feature sets and APIs, creating implementation and
704 integration issues for the management controller, which ends up needing custom code to support each
705 one separately. This consumes resources both for development of the custom code and for memory in
706 the management controller to support it. Redfish simplifies matters by enabling a single approach to
707 management for all RDE Devices.

708 Implementing the Redfish protocol as defined by [DSP0266](#) is a big challenge when passing requests to
709 and from devices such as network adapters that have highly limited processing capabilities and memory
710 space. Redfish's messages are prohibitively large because they are encoded for human readability in
711 HTTP/HTTPS using JavaScript Object Notation (JSON). This specification details a compressed
712 encoding of Redfish payloads that is suitable for such devices. It further identifies a common method to
713 use PLDM to communicate these messages between a management controller and the devices that host

714 the data the operations target. The functionality of providing a complete Redfish service is distributed
715 across components that function in different roles; this is discussed in more detail in clause 7.1.1.

716 The basic format for PLDM messages is defined in [DSP0240](#). The specific format for carrying PLDM
717 messages over a particular transport or medium is given in companion documents to the base
718 specification. For example, [DSP0241](#) defines how PLDM messages are formatted and sent using MCTP
719 as the transport. Similarly, [DSP0222](#) defines how PLDM messages are formatted and sent using NC-SI
720 and RBT as the transport. The payloads for PLDM messages are application specific. The Platform Level
721 Data Model (PLDM) for Redfish Device Enablement specification defines PLDM message payloads that
722 support the following items and capabilities:

- 723 • Binary Encoded JSON (BEJ)
 - 724 – Simplified compact binary format for communicating Redfish JSON data payloads
 - 725 – Captures essential schema information into a compact binary dictionary so that it does not need
726 to be transferred as part of message payloads
 - 727 – Defined locators allow for selection of a specific object or property inside the schema's data
728 hierarchy to perform an operation
 - 729 – Encoders and decoders account for the unordered nature of BEJ and JSON properties
- 730 • RDE Device Registration for Redfish
 - 731 – A mechanism to determine the schemas the RDE Device supports, including OEM custom
732 extensions
 - 733 – A mechanism to determine parameters for limitations on the types of communication the RDE
734 Device can perform, the number of outstanding operations it can support, and other
735 management parameters
- 736 • Messaging Support for Redfish Operations via BEJ
 - 737 – Read, Update, Post, Create, Delete Operations
 - 738 – Asynchrony support for Operations that spawn long-running Tasks
 - 739 – Notification Events for completion of long-running Tasks and for other RDE Device-specific
740 happenings¹
 - 741 – Advanced operations such as pagination and ETag support

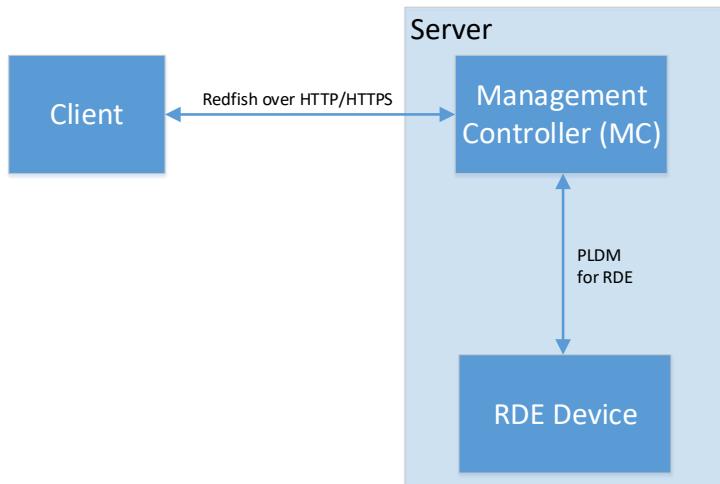
742 7.1 Redfish Provider architecture overview

743 In PLDM for Redfish Device Enablement, standard Redfish messages are generated by a Redfish client
744 through interactions with a user or a script, and communicated via JavaScript Object Notation (JSON)
745 over HTTP or HTTPS to a management controller (MC). The MC encodes the message into a binary
746 format (BEJ) and sends it over PLDM to an appropriate RDE Device for servicing. The RDE Device
747 processes the message and returns the response back over PLDM to the MC, again in binary format.
748 Next, the MC decodes the response and constructs a standard Redfish response in JSON over HTTP or
749 HTTPS for delivery back to the client.

750 7.1.1 Roles

751 RDE divides the processing of Redfish Operations into three roles as depicted in Figure 1.

1 The format for the data contained within Events is defined in [DSP0248](#). The way that events are used is defined in this specification.



752

753

Figure 1 – RDE Roles

754 The **Client** is a standard Redfish client, and needs no modifications to support operations on the data for
755 a device using the messages defined in this specification.

756 The **MC** functions as a proxy Redfish Provider for the RDE Device. In order to perform this role, the MC
757 discovers and registers the RDE Device by interrogating its schema support and building a representation
758 of the RDE Device's management topology. After this is done, the MC is responsible for receiving Redfish
759 messages from the client, identifying the RDE Device that supplies the data relevant to the request,
760 encoding any payloads into the binary BEJ format, and delivering them to the RDE Device via PLDM.
761 Finally, the MC is responsible for interacting with the RDE Device as needed to get the response to the
762 Redfish message, translating any relevant bits from BEJ back to the JSON format used by Redfish, and
763 returning the result back to the client. The MC may also act as a client to manage RDE Devices; for this
764 purpose, the MC may communicate directly with the RDE Device using BEJ payloads and the PLDM for
765 Redfish Device Enablement commands detailed in this specification.

766 The **RDE Device** is an RDE Provider. To perform this role, the RDE Device must define a management
767 topology for the resources that organize the data it provides and communicate it to the MC during the
768 discovery and registration process. The RDE Device is also responsible for receiving Redfish messages
769 encoded in the binary BEJ format over PLDM and sending appropriate responses back to the MC; these
770 messages can correspond to a variety of operations including reads, writes, and schema-defined actions.

7.2 Redfish Device Enablement concepts

772 This specification relies on several key concepts, detailed in the subsequent clauses.

7.2.1 RDE Device discovery and registration

774 The processes by which an RDE Device becomes known to the MC and thus visible to clients are known
775 as Discovery and Registration. Discovery consists of the MC becoming aware of an RDE Device and
776 recognizing that it supports Redfish management. Registration consists of the MC interrogating specific
777 details of the RDE Device's Redfish capabilities and then making it visible to external clients. An example
778 ladder diagram and a typical workflow for the discovery and registration process may be found in clause
779 9.1.

7.2.1.1 RDE Device discovery

781 The first step of the discovery process begins when the MC detects the presence of a PLDM capable
782 device on a particular medium. The technique by which the MC determines that a device supports PLDM

783 is outside the scope of this specification; details of this process may be found in the PLDM base
784 specification ([DSP0240](#)). Similarly, the technique by which the MC may determine that a device found on
785 one medium is the same device it has previously found on another medium is outside the scope of this
786 specification.

787 After the MC knows that a device supports PLDM, the next step is to determine whether the device
788 supports appropriate versions of required PLDM Types. For this purpose, the MC should use the base
789 PLDM GetPLDMLTypes command. In order to advertise support for PLDM for Redfish Device Enablement,
790 a device shall respond to the GetPLDMLTypes request with a response indicating that it supports both
791 PLDM for Platform Monitoring and Control (type 2, [DSP0248](#)) and PLDM for Redfish Device Enablement
792 (type 6, this specification). If it does, the MC will recognize the device as an RDE Device.

793 Next, the MC may use the base PLDM GetPLDMCommands command once for each of the Monitoring
794 and Control and Redfish Device Enablement PLDM Types to verify that the RDE Device supports the
795 required commands. The required commands for each PLDM Type are listed in Table 50. As with the
796 GetPLDMLTypes command, use of this command is optional if the MC has some other technique to
797 understand which commands the RDE Device supports. At this point, RDE Device discovery at the PLDM
798 level is complete.

799 Once the MC has discovered the RDE Device, it invokes the NegotiateRedfishParameters command
800 (clause 11.1) to negotiate baseline details for the RDE Device. This step is mandatory unless the MC has
801 previously issued the NegotiateRedfishParameters command to the RDE Device on a different medium.
802 Baseline Redfish parameters include the following:

- 803 • The RDE Device's RDE Provider name
- 804 • The RDE Device's support for concurrency. This is the number of Operations the RDE Device
805 can support simultaneously
- 806 • RDE feature support

807 The final step in discovery is for the MC to invoke the NegotiateMediumParameters command (clause
808 11.2) in order to negotiate communication details for the RDE Device. The MC invokes this command on
809 each medium it plans to communicate with the RDE Device on as it discovers the RDE Device on that
810 medium. Medium details include the following:

811 The size of data that can be sent in a single message on the medium

812 **7.2.1.2 RDE Device registration**

813 In the registration process, the MC interrogates the RDE Device about the hierarchy of Redfish resources
814 it supports in order to act as a proxy, transparently mirroring them to external clients. The MC may skip
815 registration of the RDE Device if the PDR/Dictionary signature retrieved via the
816 NegotiateRedfishParameters command matches one previously retrieved and the MC still has the PDRs
817 and dictionaries cached.

818 In PLDM for Redfish Device Enablement, each² Redfish resource is uniquely identified by a Resource
819 Identifier that maps from the identifier to a collection of schemas that define the data for it. The identifiers
820 in turn are collected together into Redfish Resource PDRs; resources that share a common set of
821 schemas and are linked to from a common parent (such as sibling collections members) are enumerated
822 within the same PDR. Data for secondary schemas such as annotations or the message registry is linked
823 together with the major schema in the PDR structure. The resources link together to form a management
824 topology of one or more trees called device components; each resource corresponds to a node in one (or
825 more) of these trees.

² The LogEntryCollection and LogEntry resources are an exception to this; see clause 14.2.7 for a description of special handling for them.

826 The first step in performing the registration is for the MC to collect an inventory of the PDRs supported by
827 the RDE Device. There are three main PDRs of potential interest here: Redfish Resource PDRs, that
828 represent an instance of data provided by the RDE Device; Redfish Entity Association PDRs, that
829 represent the logical linking of data; and Redfish Action PDRs that represent special functions the RDE
830 Device supports. While every RDE Device must support at least one resource and thus at least one
831 Redfish Resource PDR, Redfish Action PDRs are only required if the device supports schema-defined
832 actions and Redfish Entity Association PDRs are only required under limited circumstances detailed in
833 clause 7.2.2. The MC shall collect this information by first calling the PLDM Monitoring and Control
834 GetPDRRepositoryInfo command to determine the total number of PDRs the RDE Device supports. It
835 shall then use the PLDM Monitoring and Control GetPDR command to retrieve details for each PDR from
836 the RDE Device.

837 As it retrieves the PDR information, the MC should build an internal representation of the data hierarchy
838 for the RDE Device, using parent links from the Redfish Resource PDRs and association links from the
839 Redfish Entity Association PDRs to define the management topology trees for the RDE Device.

840 After the MC has built up a representation of the RDE Device's management topology, the next step is to
841 understand the organization of data for each of the tree nodes in this topology. To this end, the MC
842 should first check the schema name and version indicated in each Redfish Resource PDR to understand
843 what the RDE Device supports. For any of these schemas, the MC may optionally retrieve a binary
844 dictionary containing information that will allow it to translate back and forth between BEJ and JSON
845 formats. It may do this by invoking the GetSchemaDictionary (clause 11.2) command with the ResourceID
846 contained in the corresponding Redfish Resource PDR.

847 NOTE While the MC may typically be expected to retrieve Redfish PDRs and dictionaries when it first registers an
848 RDE Device, there is no requirement that implementations do so. In particular, some implementations may
849 determine that one or more dictionaries supported by an RDE Device are already supported by other
850 dictionaries the MC has stored. In such a case, downloading them anew would be an unnecessary
851 expenditure of resources.

852 After the MC has all the schema information it needs to support the RDE Device's management topology,
853 it can then offer (by proxy) the RDE Device's data up to external clients. These clients will not know that
854 the MC is interpreting on behalf of an RDE Device; from the client perspective, it will appear that the client
855 is accessing the RDE Device's data directly.

856 7.2.2 Data instances of Redfish schemas: Resources

857 In the Redfish model, data is collected together into logical groupings, called resources, via formal
858 schemas. One RDE Device might support multiple such collections, and for each schema, might have
859 multiple instances of the resource. For example, a RAID disk controller could have an instance of a disk
860 resource (containing the data corresponding to the Redfish disk schema) for each of the disks in its RAID
861 set.

862 Each resource is represented in this specification by a resource identifier contained within a Redfish
863 Resource PDR (defined in [DSP0248](#)). OEM extensions to Redfish resources are considered to be part of
864 the same resource (despite being based on a different schema) and thus do not require distinct Redfish
865 Resource PDRs.

866 Each RDE Device is responsible for identifying a management topology for the resources it supports and
867 reflecting these topology links in the Redfish Resource and Redfish Entity Association PDRs presented to
868 the MC. This topology takes the form of a directed graph rooted at one or more nodes called device
869 components. Each device component shall proffer a single Redfish Resource PDR as the logical root of
870 its own portion of the management topology within the RDE Device.

871 Links between resources can be modeled in three different ways. Direct subordinate linkage, such as
872 physical enclosure or being a component in a ComputerSystem, may be represented by setting the
873 ContainingResourceID field of the Redfish Resource PDR to the Resource ID for the parent resource. In

874 Redfish terminology, this relation is used to show subordinate resources. The parent field for the logical
875 root of a device component is set to EXTERNAL, 0x0000.

876 Logical links between resources can also be modeled. In cases where a resource and the resource to
877 which it is related are both contained within an RDE Device, these links are handled implicitly by filling in
878 the Links section of the Redfish resource when data for the resource is retrieved from the RDE Device.

879 Alternatively, logical links between resources may be represented by creating instances of Redfish Entity
880 Association PDRs (defined in [DSP0248](#)) to capture these links. In Redfish terminology, this relation is
881 used to show related resources. For example, as shown in Figure 2, the drives in a RAID subsystem are
882 subordinate to the storage controller that manages them, but are also linked to the standard Chassis
883 object. A Redfish Entity Association PDR shall only be used when a resource meets all three of the
884 following criteria:

- 885 1) The resource is contained within the RDE Device. If it is not, it does not need to be part of the
886 RDE Device's management topology model.
- 887 2) The resource is subordinate to another resource contained within the RDE Device. If it is not,
888 the resource can be linked directly to the resource outside the RDE Device by setting its parent
889 field to EXTERNAL.
- 890 3) The resource needs to be linked to another resource outside the RDE Device.

891 **7.2.2.1 Alignment of resources**

892 While determining how to lay out the Redfish Resource PDRs for an RDE Device may seem to be a
893 daunting task at first glance, it is actually relatively straightforward. By examining the Links section of the
894 various schemas that the RDE Device needs to support, one will see that the tree hierarchy for them is
895 already defined. Simply put, then, the RDE Device manufacturer will set up one PDR per resource or
896 group of sibling resources that share the same schema definitions and reflect the same parentage trees
897 for the PDRs as is already present for the resources in their corresponding Redfish schema definitions.

898 NOTE For collections, the RDE Device shall offer one PDR for the collection as a whole and one PDR for each set
899 of sibling entries within the collection. This is necessary to enable the MC to use the correct dictionary when
900 encoding data for a Create operation applied to an empty collection.

901 **7.2.2.2 Example linking of PDRs within RDE Devices**

902 This clause presents examples of the way an RDE Device can link Redfish Resource PDRs together to
903 present its data for management.

904 The example in Figure 2 models a simple rack-mounted server with local RAID storage. In this example,
905 we see a Redfish Resource PDR offering an instance of the standard Redfish Storage resource, with
906 ResouceID 123. This PDR has ContainingResouceID (abbreviated ContainingRID in the figure) set to
907 EXTERNAL as the RDE Device should be subordinate to the Storage Collection under ComputerSystem.

908 NOTE It is up to the MC to make final determinations as to where resources should be added within the Redfish
909 hierarchy. While general guidance may be found in clause 14.2.6, the technique by which MCs may
910 ultimately make such decisions is out of scope for this specification.

911 The StorageController has two Redfish Resource PDRs that list it as their container: one that offers data
912 in the VolumeCollection resource and one that offer data for four Disk resources. Finally, the PDR that
913 offers VolumeCollection resource is marked as the container for a Redfish Resource PDR that offers data
914 for the Volume resource.

915 The connections discussed so far are all direct parent linkages in the Redfish Resource PDRs because
916 the links they represent are the direct subordinate resource links from the standard Redfish storage
917 model. However, the Redfish storage model also includes notations that drives are related to (contained
918 within) a volume and that drives are related to (present inside) a chassis. These resource relations can be
919 modeled using Redfish Entity Association PDRs if the MC is managing the links. Alternatively, they can

920 be implicitly managed by the RDE Device. In this case, the RDE Device will expose the links itself by
921 filling in a Links section of the relevant resource data with references to the linked resources. While the
922 RDE Device could in theory provide a Redfish Entity Association PDR for this case, it serves no purpose
923 for the MC.

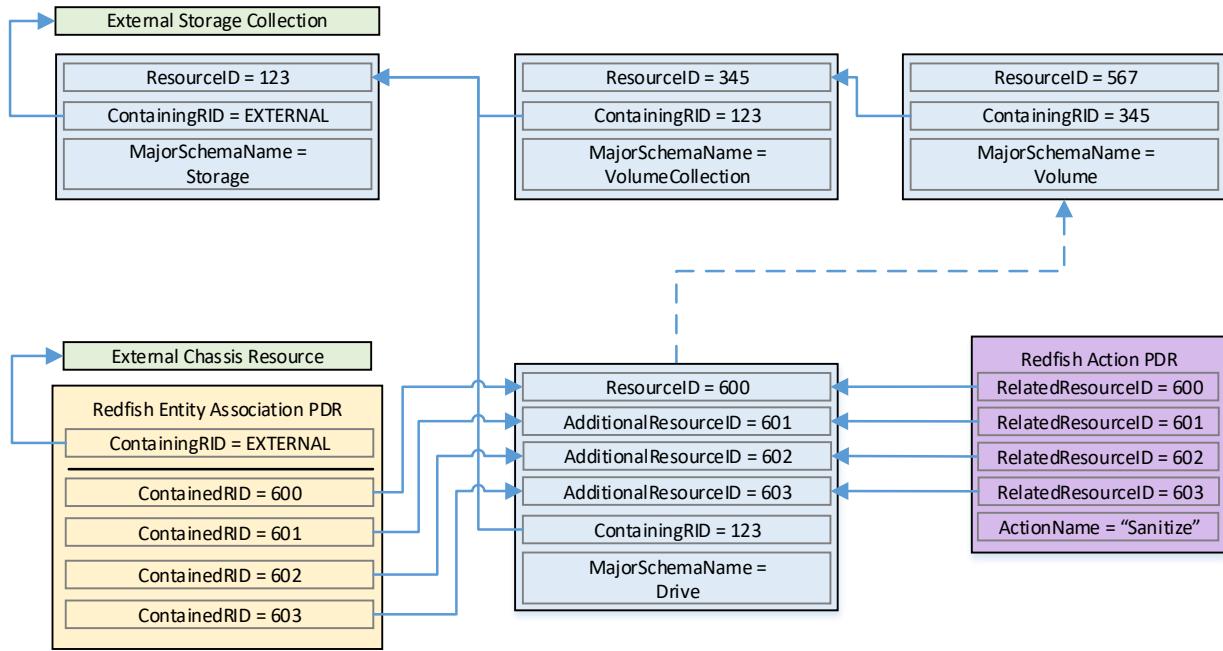
924 In general, a Redfish Entity association PDR should be used when a resource is subordinate to another
925 resource within the RDE Device but must also be linked to from another resource external to the RDE
926 Device.

927 In the example in Figure 2, the relation between the drives and the outside Chassis resource is
928 promulgated with a Redfish Entity Association PDR. This PDR lists the four drives as the four
929 ContainingResourceIDs for the association, marking them to be contained within the chassis. The
930 ContainingResourceID for this relation contains the value EXTERNAL, to show that the drives are visible
931 outside the resource hierarchy maintained by the RDE Device. By contrast, the linkage between the
932 drives and the Volume resource is implicitly maintained by the RDE Device. This is shown in the figure via
933 the dashed arrows.

934 Finally, each of the drives supports a Sanitize operation. This is shown by instantiating a Redfish Action
935 PDR naming the Sanitize action and linking it to each of the drives.

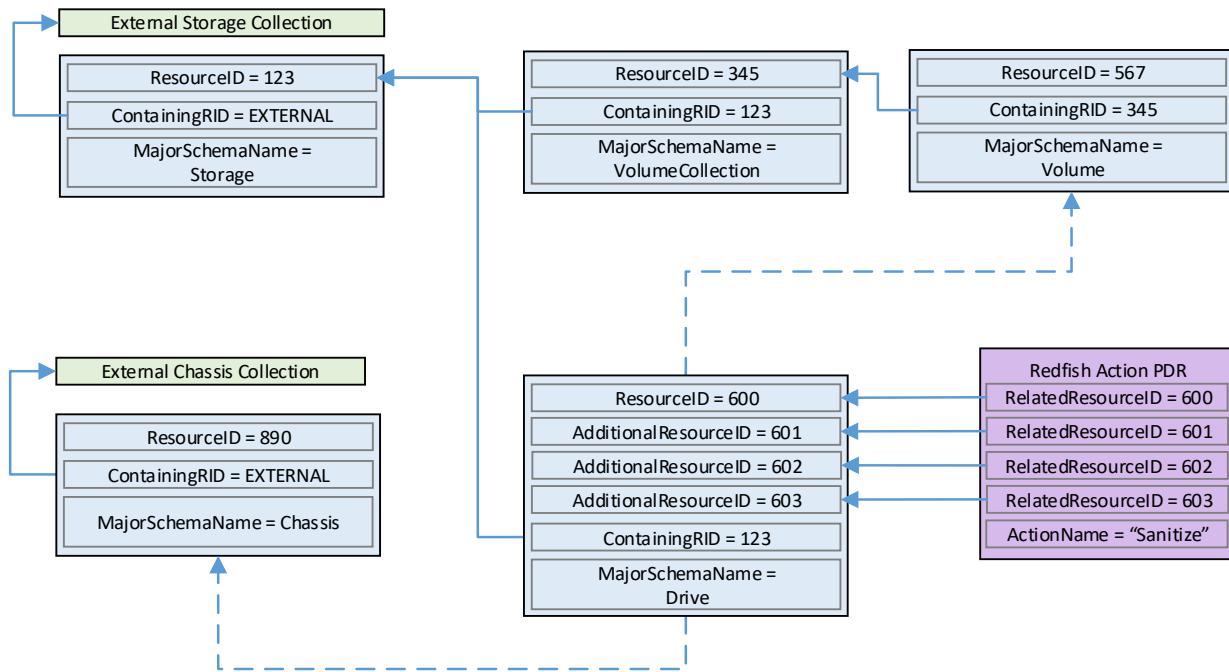
936 As an alternative to the PDR layout of Figure 2, in

937 Figure 3, the RDE Device exposes its own chassis resource (labeled as Resource ID 890) rather than
 938 having the drives be part of an external chassis. The PDR for this chassis resource shows
 939 ContainingResourceID EXTERNAL to demonstrate that it belongs in the system chassis collection
 940 resource. With this modification, the links between the chassis resource and the drives can be managed
 941 internally by the RDE Device and hence no Redfish Entity Association PDR is necessary.



942
943

944 **Figure 2 – Example linking of Redfish Resource and Redfish Entity Association PDRs**



945
946

Figure 3 – Schema linking without Redfish entity association PDRs**7.2.3 Dictionaries**

In standard Redfish, data is encoded in JSON. In this specification, data is encoded in Binary Encoded JSON (BEJ) as defined in clause 8. In order to translate between the two encodings, the MC uses a schema lookup table that captures key metadata for fields contained within the schema. The dictionary is necessary because some of the JSON tokens are omitted from the BEJ encoding in order to achieve a level of compactness necessary for efficient processing by RDE Devices with limited memory and computational resources. In particular, the names of properties and the string values of enumerations are skipped in the BEJ encoding.

Each Redfish resource PDR can reference up to four classes of dictionaries for the schemas it can use³:

- Standard Redfish data schema (aka the major schema)
- Standard Redfish Event schema
- Standard Redfish Annotation schema
- Standard Redfish Error schema

Major and Event Dictionaries may be augmented to contain OEM extension data as defined in the Redfish base specification, [DSP0266](#).

Event, Error, and Annotation Dictionaries shall be common to all resources that an RDE Device provides.

Dictionaries for standard Redfish schemas are published on the DMTF Redfish website at <http://redfish.dmtf.org/dictionaries>. Naturally, these dictionaries do not include OEM extensions. RDE Devices may support their resources either with

the standard dictionaries or with custom dictionaries that may include OEM extensions, and that may also be truncated to contain only entries for properties supported by the RDE Device.

7.2.3.1 Canonizing a schema into a dictionary

In Redfish schemas, the order of properties is indeterminate and properties are identified by name identifiers that are of unbounded length. While this is beneficial from a human readability perspective, from a strict information-theoretical point of view, using long strings for this purpose is grossly inefficient: a numeric value of $\text{Log}_2(n\text{Children})$ bits ought to be sufficient. To make this work in practice, we impose a canonical ordering that assigns each property or enumeration value a numeric sequence number. Sequence numbers shall be assigned according to the following rules:

- 1) The children properties (properties immediately contained within other properties such as sets or arrays) shall collectively receive an independent set of sequence numbers ranging from zero to $N - 1$, where N is the number of children. Sequence numbers for properties that do not share a common parent are not related in any way.
- 2) For the initial revision of a Redfish schema (usually v1.0), sequence numbers shall be assigned according to a strict alphabetical ordering of the property names from the schema.
- 3) In order to preserve backward compatibility with earlier versions of schemas, for subsequent revisions of Redfish schemas, the sequence numbers for child properties added in that revision shall be assigned sequence numbers N to $N + A - 1$, where N is the number of sequence numbers assigned in the previous revision and A is the number of properties added in the present revision. (In other words, we append to the existing set and use sequence numbers

³ The COLLECTION_MEMBER_TYPE schema class from clause 5.3.2 is not represented in the PDR. It can be retrieved on demand by the MC from the RDE Device via the GetSchemaDictionary command of clause 11.3.

- beginning with the next one available.) The new sequence numbers shall be assigned according to a strict alphabetical ordering of their names from the schema.
- 4) In the event that a property is deleted from a schema, its sequence number shall not be reused; the sequence number for the deleted property shall forever remain allocated to that property.
 - 5) As with properties, the values of an enumeration shall collectively receive an independent set of sequence numbers ranging from zero to $N - 1$, where N is the number of enumeration values. Sequence numbers for enumeration values not belonging to the same enumeration are not related in any way.
 - 6) For the initial version of a Redfish schema, sequence numbers for enumeration values shall be assigned according to a strict alphabetical ordering of the enumeration values from the schema.
 - 7) In order to preserve backward compatibility with earlier versions of schemas, for subsequent revisions of Redfish schemas, the sequence numbers for enumeration values added in that revision shall be assigned sequence numbers N to $N + A - 1$, where N is the number of sequence numbers assigned in the previous revision and A is the number of enumeration values added in the present revision. The new sequence numbers shall be assigned according to a strict alphabetical ordering of their value strings from the schema.
 - 8) In the event that an enumeration value is deleted from a schema, its sequence number shall not be reused; the sequence number for the deleted enumeration value shall forever remain allocated to that enumeration value.

After the sequence numbers for properties and enumeration values are assigned, they shall be collected together with other information from the Redfish and OEMs schema to build a dictionary in the format detailed in clause 7.2.3.2. For every Redfish Resource PDR the RDE Device offers, it shall maintain a dictionary that it can send to the MC on demand in response to a GetSchemaDictionary command (clause 11.2).

NOTE Rules 2 and 3 above imply that schema child properties may not be in strict alphabetical order. For example, suppose a property node in a schema started with child fields "red", "orange", and "yellow" in version 1.0. Because this is the initial version, the fields would be alphabetized: "orange" would get sequence number 0; "red", 1; and "yellow" would get 2. If version 1.1 of the schema were to add "blue" and "green", they would be assigned sequence numbers 3 and 4 respectively (because that is the alphabetical ordering of the new properties). The initial three properties retain their original sequence numbers.

For all custom dictionaries, including all truncated dictionaries, the sequence numbers listed for standard Redfish schema properties supported by the RDE Device shall match the sequence numbers for those same properties from the standard dictionary. This allows MCs to potentially merge related dictionaries from RDE Devices that share a common class.

Sequence numbers for array elements shall be assigned to match the zero-based index of the array element.

NOTE The ordering rules provided in this clause apply to dictionaries only. In particular, data encoded in either JSON or BEJ format is by definition unordered.

7.2.3.2 Dictionary binary format

The binary format of dictionaries shall be as follows. All integer fields are stored little endian:

Table 31 – Redfish dictionary binary format

Type	Dictionary Data
uint8	VersionTag Dictionary format version tag: 0x00 for DSP0218 v1.0.0, v1.1.0, v1.1.1

Type	Dictionary Data
bitfield8	<p>DictionaryFlags Flags for this dictionary: [7:1] - reserved for future use [0] - truncation_flag; if 1b, the dictionary is truncated and provides entries for a subset of the full Redfish schema</p>
uint16	<p>EntryCount Number N of entries contained in this dictionary</p>
uint32	<p>SchemaVersion Version of the Redfish schema encapsulated in this dictionary, in standard PLDM format. 0xFFFFFFFF for an unversioned schema. The version of the schema may be read from the filename of the schema file.</p>
uint32	<p>DictionarySize Size in bytes of the dictionary binary file. This value can be used as a safeguard to compare the various offsets given in subsequent fields against: buffer overruns can be avoided by validating that the offsets remain within the binary dictionary space.</p>
bejTupleF	<p>Format [0] Entry 0 property format. The read_only_property_and_top_level_annotation flag in the bejTupleF structure shall be set if the property is annotated as read only in the Redfish schema. The nullable_property in the bejTupleF structure shall be set if the property is annotated as nullable in the Redfish schema.</p>
uint16	<p>SequenceNumber [0] Entry 0 property sequence number</p>
uint16	<p>ChildPointerOffset [0] Entry 0 property child pointer offset in bytes from the beginning of the dictionary. Shall be 0x0000 if Format [0] is not one of {BEJ Set, BEJ Array, BEJ Enum and BEJ Choice} or in cases where a set or array contains no children elements.</p>
uint16	<p>ChildCount [0] Entry 0 child count; shall be 0x0000 if Format [0] is not one of {BEJ Set, BEJ Array, BEJ Enum}. For a BEJ Array, the child count shall be expressed as 1.</p>
uint8	<p>NameLength [0] Entry 0 property/enumeration value name string length. Name length, including null terminator, shall be a maximum of 255 characters. Shall be 0x00 for an anonymous format option of a BEJ Choice-formatted property or for anonymous array entries.</p>
uint16	<p>NameOffset [0] Entry 0 property name string offset in bytes from the beginning of the dictionary. Shall be 0x0000 for an anonymous format option of a BEJ Choice-formatted property or for anonymous array entries.</p>
...	...
bejTupleF	<p>Format [N – 1] Entry (N – 1) property format. The read_only_property_and_top_level_annotation flag in the bejTupleF structure shall be set if the property is annotated as read only in the Redfish schema. The nullable_property in the bejTupleF structure shall be set if the property is annotated as nullable in the Redfish schema.</p>
uint16	<p>SequenceNumber [N – 1] Entry (N – 1) property sequence number</p>

Type	Dictionary Data
uint16	ChildPointerOffset [N – 1] Entry (N – 1) property child pointer offset in bytes from the beginning of the dictionary. Shall be 0x0000 if Format [N – 1] is not one of {BEJ Set, BEJ Array, BEJ Enum and BEJ Choice}.
uint16	ChildCount [N – 1] Entry (N – 1) child count; shall be 0x0000 if Format [N] is not one of {BEJ Set, BEJ Array, BEJ Enum}. For a BEJ Array, the child count shall be expressed as 1.
uint8	NameLength [N – 1] Entry (N – 1) property/enumeration value name string length. Name length, including null terminator, shall be a maximum of 255 characters. Shall be 0x00 for an anonymous format option of a BEJ Choice-formatted property or for anonymous array entries.
uint16	NameOffset [N – 1] Entry (N – 1) property name string offset in bytes from the beginning of the dictionary. Shall be 0x0000 for an anonymous format option of a BEJ Choice-formatted property or for anonymous array entries.
strUTF-8	Name [0] Entry 0 property name string (not present for children nodes of BEJ Choice format properties or anonymous array entries)
...	
strUTF-8	Name [N – 1] Entry (N – 1) property name string (not present for children nodes of BEJ Choice format properties or anonymous array entries)
uint8	CopyrightLength Dictionary copyright statement string length. Copyright, including null terminator, shall be a maximum of 255 characters. May be 0x00 in which case the Copyright field below shall be omitted.
strUTF-8	Copyright Copyright statement for the dictionary. Shall be omitted if CopyrightLength is 0.

1028 Intuitively, the dictionary binary format may be thought of as a header (orange) followed by an array of
 1029 entry data (blue) followed by a table of the strings (green) naming the properties and enumeration values
 1030 for the entries. Figure 4 displays this data in graphical format:

1031

	Byte offset			
DWORD	+0	+1	+2	+3
00	VersionTag 0x00	DictionaryFlags	EntryCount ₂	EntryCount ₁
01	SchemaVersion ₄	SchemaVersion ₃	SchemaVersion ₂	SchemaVersion ₁
02	DictionarySize ₄	DictionarySize ₃	DictionarySize ₂	DictionarySize ₁
03	Format[0]	SequenceNumber[0] ₂	SequenceNumber[0] ₁	ChildPointerOffset[0] ₂
04	ChildPointerOffset[0] ₁	ChildCount[0] ₂	ChildCount[0] ₁	NameLength[0]
05	NameOffset[0] ₂	NameOffset[0] ₁

	Byte offset			
DWORD	+0	+1	+2	+3
06
...	Format[N-1]	SequenceNumber[N-1] ₂	SequenceNumber[N-1] ₁	ChildPointerOffset[N-1] ₂
...	ChildPointerOffset[N-1] ₁	ChildCount[N-1] ₂	ChildCount[N-1] ₁	NameLength[N-1]
...	NameOffset[N-1] ₂	NameOffset[N-1] ₁	Name[0] ₁ *	Name[0] ₂ *
...	Name[0] ₃ *	...	Name[0]terminator*	...
...
...	Name[N-1] ₁ *	Name[N-1] ₂ *	Name[N-1] ₃ *	...
...	Name[N-1]terminator*	CopyrightLength	Copyright ₁	...
...	Copyrightterminator			

1032

Figure 4 – Dictionary binary format

1033 * Name strings will not be present in the dictionary for anonymous format options of BEJ Choice-
 1034 formatted properties or for anonymous array entries.

1035 **7.2.3.2.1 Hierarchical organization of entries**

1036 Within this binary format, the entries shall be sorted into clusters representing a breadth-first traversal of
 1037 the hierarchy presented by a schema. Each cluster shall in turn consist of all the sibling nodes contained
 1038 within a common parent, sorted by sequence number per the rules defined in clause 7.2.3 above. An
 1039 example of this organization may be found in clause 8.6.1.

1040 NOTE While not mandatory, it is acceptable that multiple dictionary entries may point to a common complex
 1041 subtype to allow reuse of that information and reduce the overall size of the dictionary. For example,
 1042 Resource.status is commonly used multiple times within the same schema, so having a single offset for it
 1043 can trim some length from the dictionary.

1044 **7.2.3.3 Properties that support multiple formats**

1045 For properties that support multiple formats, the dictionary shall contain an entry linking the property
 1046 name string to the BEJ Choice format. This choice entry shall in turn link to a series of anonymous child
 1047 entries (name offset = 0x0000) that are of the various data formats supported by the property. For
 1048 example, if a TCP/IP hostname property supports both string ("www.dmtf.org") and numeric (the 32-bit
 1049 equivalent of 72.47.235.184) values, the dictionary might contain rows such as the following:

1050 **Table 32 – Dictionary entry example for a property supporting multiple formats**

Row	Sequence Number	Format	Name	Child Pointer
...
15	0	choice	"hostname"	18

Row	Sequence Number	Format	Name	Child Pointer
...
18	0	string	null	null
19	1	integer	null	null
...

1051 NOTE Following the rules for sequence number assignment (see clause 7.2.3.1), each cluster of properties
 1052 contained within a given set and each cluster of enumeration values are numbered separately. Hence
 1053 sequence numbers may be repeated within a dictionary.

1054 An exception to this rule is that properties that support null and exactly one other data format shall be
 1055 collapsed into a single entry in the dictionary listing only the non-null data format. The nullable_property
 1056 bit in the `bejTupleF` value of the format entry in the dictionary shall be set to 1b in this case. This case is
 1057 common in the standard Redfish schemas, where most properties are nullable. This is flagged with the
 1058 “nullable” keyword in the CSDL schemas, but in the JSON schemas, it manifests as the supported type
 1059 list for the property consisting of `NULL` and either a solitary second type or a collection of strings that form
 1060 an enumeration.

1061 **7.2.3.4 Annotation dictionary format**

1062 Standard Redfish annotations are derived from three sources: the Redfish, odata, and message
 1063 schemas. The annotations that can be part of a JSON payload are collected together into the `redfish-`
 1064 `payload-annotations.vX.Y.Z.json` schema file. This clause details special notes that apply to building the
 1065 annotation dictionary:

- The dictionary entries for properties in the annotation dictionary shall include the entire name of the annotation, beginning with the '@' sign and including both the annotation source (one of redfish, message, or odata) and the annotation's name itself. For example, the dictionary Name field for the `@odata.id` property shall be an offset to the string "`@odata.id`".
- The dictionary entries for patternProperties in the annotation dictionary shall be stripped of the wildcard patterns before the '@' sign and of the trailing '\$' sign but shall otherwise be treated identically to standard properties. For example, the dictionary Name field for the "`^([a-zA-Z_][a-zA-Z0-9_]*?)@Message.ExtendedInfo$`" patternProperty shall be an offset to the string "`@Message.ExtendedInfo`".
- In accordance with the rules presented in clause 7.2.3, the top-level entries for annotations (those containing the names of the annotations themselves) shall be sorted alphabetically together for the initial version of the schema's dictionary, and shall be appended to the list with each schema revision. Stated explicitly, the annotations from the properties and patternProperties shall be comingled together within the entries for each revision of the dictionary.
- Dictionary entries for children properties of annotations, such as the anonymous string value array entries for `@Redfish.AllowableValues` shall be structured and formatted per the rules presented in clause 7.2.3.

1084 **7.2.3.5 Registry dictionary format**

1085 Redfish messages are used in multiple places, including annotations, events, and errors. The actual
 1086 message data may be retrieved from any of the various message registries including standard Redfish
 1087 and OEM registries. These messages are referred to by name as the value of a string field in hosting
 1088 schemas, so names such as “`NetworkDevice.1.0.LinkFlapDetected`” appear in BEJ-encoded JSON data

for previous versions of this specification. To reduce the size of such encodings, RDE version 1.1 introduces the notion of a Registry dictionary that can be referenced via the `bejRegistryItem` encoding format. Replacing the message name with a sequence number in the Registry dictionary achieves a reduction in encoded data for messages. This clause details special notes that apply to building the registry dictionary:

- The registry dictionary shall consist of a top-layer set named “registry”
 - Entries within the set shall be named for each of the registry items supported by the RDE Device. The full odata name for these entries shall be incorporated in the dictionary, and they shall be sorted lexicographically.
 - The type of the registry items shall be `bejString`, and they shall be flagged as read-only.
- Both full and truncated registry dictionaries are permitted.
- MCs shall not attempt to merge registry dictionaries from different devices or dictionaries retrieved from the same device at different times.
- If using the DMTF dictionary builder tool (see clause 7.2.3.7), see the tool documentation for information on how to build the registry dictionary for a device.
- Schema entries that correspond to registry items shall be encoded in dictionaries as being of type `bejString`, not `bejRegistryItem`. This ensures backward compatibility with earlier versions of the RDE specification

7.2.3.6 Links between schemas

Links in Redfish schemas, identifiable as entries with Odata type `odata.id`, shall be represented in dictionaries as entries with format = `bejString`. As described in clause 8.4.2, runtime encoding of Odata links may be performed via any of `bejString` (with deferred bindings), `bejResourceLink`, or (for expansion) `bejResourceLinkExpansion`. This is a special case wherein a valid encoding may differ from the type specified in the dictionary.

7.2.3.7 Building dictionaries

Available online at <https://github.com/DMTF/RDE-Dictionary>, the RDE dictionary builder automates the process of building an RDE dictionary from CSDL formatted schemas.

It supports standard Redfish schemas, standalone OEM schemas, and OEM extensions to standard Redfish schemas and can build full or truncated dictionaries. For more information about installation, usage and examples of using the dictionary builder, refer to the `README.md` file at the above URL.

7.2.4 Redfish Operation support

Redfish Operations are sent from a client to a Redfish Provider that is able to process them and respond appropriately. These operations are encoded in JSON and transported via either the HTTP or the HTTPS protocol.

In this specification, the MC is the Redfish Provider to which the client sends operations. However, rather than responding directly, the MC is a proxy that conveys these operations to the RDE Devices that maintain the data and can provide responses to client requests. The proxied operations (that are transmitted to the RDE Device as RDE Operations) are encoded in BEJ (clause 8) and transported via PLDM. The MC, in its role as proxy Redfish Provider for the RDE Devices, translates the JSON/HTTP(S) requests from the client into BEJ/PLDM for the RDE Device, and then translates the BEJ/PLDM response from the RDE Device into a JSON/HTTP(S) response for the client.

1130 **7.2.4.1 Primary Operations**

1131 There are seven primary Redfish Operations. These are summarized in Table 33.

1132 **Table 33 – Redfish Operations**

Operation	Verb	Description
Read	GET	Retrieve data values for all properties contained within a resource.
Update	PATCH	Write updates to properties within a resource. May be to the entire resource, to a subtree rooted at any point within the resource, or to a leaf node.
Replace	PUT	Write replacements for all properties within a resource.
Create	POST	Append a new set of child data to a collection (array).
Delete	DELETE	Remove a set of child data from a collection.
Action	POST	Invoke a schema-defined Redfish action.
Head	HEAD	Retrieve just headers for the data contained in a schema.

1133 The only Redfish Operation that is required to be supported in RDE is Read; however, it is expected that
1134 implementations will support Update as well. Create and Delete are conditionally required for RDE
1135 Devices that contain collections; Action is conditionally required for RDE Devices that support Redfish
1136 schema-defined actions. The Head and Replace Redfish Operations are strictly optional.1137 **7.2.4.1.1 HTTP/HTTPS and Redfish**1138 A full discussion of the HTTP/HTTPS protocol is beyond the scope of this specification; however, a
1139 minimalist overview of key concepts relevant to Redfish Device Enablement follows. Readers are directed
1140 to [DSP0266](#) for more detailed information on the usage of HTTP and HTTPS with Redfish and to
1141 standard documentation for more general information on the HTTP/HTTPS protocols themselves.1142 **7.2.4.1.1.1 Redfish Operation requests**1143 Every Redfish request has a target URI to which it should be applied; this URI is the target of the
1144 HTTP/HTTPS verb listed in Table 33. The URI may consist of several parts of interest for purposes of this
1145 specification: a prefix that points to the RDE Device being managed, a subpath within the RDE Device
1146 management topology, a specific resource selection preceded by an octothorp character (#), and one or
1147 more query options preceded by a question mark (?) character.1148 Many, but not all, Redfish requests have a JSON payload associated with them. For example, a POST
1149 operation to create a new child element in a collection would normally contain a JSON payload for the
1150 data being supplied for that new child element.1151 Finally, every Redfish HTTP/HTTPS request will contain a series of headers, each of which modifies it in
1152 some fashion.1153 **7.2.4.1.1.2 Redfish Operation responses**1154 The response to a Redfish HTTP/HTTPS request will also contain several elements. First, the response
1155 will contain a status code that represents the result of the operation. Like for requests, [DSP0266](#) defines
1156 several response headers that may need to be supplied in conjunction with a Redfish response. Finally, a
1157 JSON payload may be present such as in the case of a read operation.

1158 **7.2.4.1.1.3 Generic handling of Redfish Operations**

1159 Generically, to handle processing of a Redfish HTTP/HTTPS request, the MC will typically implement the
1160 following steps. This overview ignores error conditions, timeouts, and long-lived Tasks. A much more
1161 detailed treatment may be found in clause 9.

- 1162 1) Parse the prefix of the supplied URI to pinpoint the RDE Device that the operation targets.
- 1163 2) Parse the RDE Device portion of the URI to identify the specific place in the RDE Device's
1164 management topology targeted by the operation.
- 1165 3) Identify the Redfish Resource PDR that represents that portion of the data.
- 1166 4) Using the HTTP/HTTPS verb and other request information, determine the type of Redfish
1167 operation that the client is trying to perform.
- 1168 5) Translate any request headers (clause 7.2.4.2) and query options (clause 7.2.4.3) into
1169 parameters to the corresponding PLDM request message(s).
- 1170 6) Translate the JSON payload, if present, into a corresponding BEJ (clause 8) payload for the
1171 request, using a dictionary appropriate for the target Redfish Resource PDR.
- 1172 7) Send the PLDM for Redfish Device Enablement RDEOperationInit command (clause 12.1) to
1173 begin the Operation.
- 1174 8) Send any BEJ payload to the RDE Device via one or more PLDM for Redfish Device
1175 Enablement RDEMultipartSend commands (clause 13.1) unless it was small enough to be
1176 inlined in the RDEOperationInit command.
- 1177 9) Send any request parameters to the RDE Device via the PLDM for Redfish Device Enablement
1178 SupplyCustomRequestParameters command (clause 12.2).
- 1179 10) If there was a payload but no request parameters, send the RDEOperationStatus command
1180 (clause 12.5).
- 1181 11) Retrieve and decode any BEJ-encoded JSON data for any Operation response payloads via
1182 one or more PLDM for Redfish Device Enablement RDEMultipartReceive commands (clause
1183 13.2).
- 1184 12) Retrieve any response parameters via the PLDM for Redfish Device Enablement
1185 RetrieveCustomResponseHeaders command (clause 12.3).
- 1186 13) Send the PLDM for Redfish Device Enablement RDEOperationComplete command (clause
1187 12.4) to inform the RDE Device that it may discard any data structures associated with the
1188 Task.
- 1189 14) Translate the BEJ response payload, if present, into JSON format for return to the client, using
1190 an appropriate dictionary.
- 1191 15) Prepare and send the final response to the client, adding the various HTTP/HTTPS response
1192 headers (clause 7.2.4.2) appropriate to the type of Redfish operation that was just performed.

1193 **7.2.4.2 Redfish operation headers**

1194 Several HTTP/HTTPS transport layer headers modify Redfish operations when translated in the context
1195 of RDE Operations. These are summarized in Table 34. Implementation notes for how the MC and RDE
1196 Device shall support some of these modifiers – when attached to Redfish operations – may be found in
1197 the indicated subsections. For headers not listed here, the implementation is outside the scope of this
1198 specification; implementers shall refer to [DSP0266](#) and standard HTTP/HTTPS documentation for more
1199 information on processing these headers.

1200

Table 34 – Redfish operation headers

Header	Clause	Where Used	Description
Request Headers			
If-Match	7.2.4.2.1	Request	If-Match shall be supported on PUT and PATCH requests for resources for which the RDE Device returns ETags, to ensure clients are updating the resource from a known state.
If-None-Match	7.2.4.2.2	Request	If this HTTP header is present, the RDE Device will only return the requested resource if the current ETag of that resource does not match the ETag sent in this header. If the ETag specified in this header matches the resource's current ETag, the status code returned from the GET will be 304.
Custom HTTP/ HTTPS Headers	7.2.4.2.3	Request and Response	Non-standard headers used for custom purposes.
Response Headers			
ETag	7.2.4.2.4	Response	An identifier for a specific version of a resource, often a message digest.
Link	7.2.4.2.5	Response	Link headers shall be returned as described in the clause on Link Headers in DSP0266 .
Location	7.2.4.2.6	Response	Indicates a URI that can be used to request a representation of the resource. Shall be returned if a new resource was created.
Cache-Control	7.2.4.2.7	Response	This header shall be supported and is meant to indicate whether a response can be cached or not
Allow	7.2.4.2.8	Response	Shall be returned with a 405 (Method Not Allowed) response to indicate the valid methods for the specified Request URI. Should be returned with any GET or HEAD operation to indicate the other allowable operations for this resource.
Retry-After	7.2.4.2.9	Response	Used to inform a client how long to wait before requesting the Task information again.

1201 **7.2.4.2.1 If-Match request header**1202 The MC shall support the If-Match header when applied to Redfish HTTP/HTTPS PUT and PATCH
1203 operations; support for other Redfish operations is optional.

1204 The parameter for this header is an ETag.

1205 In order to support this header, the MC shall convey the supplied ETag to the RDE Device via the
1206 ETag[0] field of the PLDM SupplyCustomRequestParameters command (clause 12.2) request message
1207 and supply the value ETAG_IF_MATCH for the ETagOperation field of the same message. For this
1208 header, the MC shall supply the value 1 for the ETagCount field of the request message.1209 When the RDE Device receives an ETAG_IF_MATCH within the ETagOperation field in the
1210 SupplyCustomRequestParameters command, it shall verify that the ETag matches the current state of the
1211 targeted schema data instance before proceeding with the RDE Operation. In the event of a mismatch, it
1212 shall respond to the SupplyCustomRequestParameters command with completion code
1213 ERROR_ETAG_MATCH.1214 In the event that both an If-Match and If-None-Match request header are supplied by the client, the MC
1215 shall respond with HTTP status code 400 – Bad Request – to the client and stop processing the request.
1216 The MC shall not send such a malformed request to the RDE Device.

1217 **7.2.4.2.2 If-None-Match request header**

1218 The MC may optionally support the If-None-Match header when applied to Redfish HTTP/HTTPS GET
1219 and HEAD operations.

1220 The parameter for this header is a comma-separated list of ETags.

1221 In order to support this header, the MC shall convey the supplied ETag(s) to the RDE Device via the
1222 ETag[i] fields of the PLDM SupplyCustomRequestParameters command (clause 12.2) request message
1223 and supply the value ETAG_IF_NONE_MATCH for the ETagOperation field of the same message. For
1224 this header, the MC shall supply the value N for the ETagCount field of the request message where N is
1225 the number of entries in the comma-separated list.

1226 When the RDE Device receives an ETAG_IF_NONE_MATCH within the ETagOperation field in the
1227 SupplyCustomRequestParameters command, it shall verify that none of the supplied ETags matches the
1228 current state of the targeted schema data instance before proceeding with the RDE Operation. In the
1229 event of a match, it shall respond to the SupplyCustomRequestParameters command with completion
1230 code ERROR_ETAG_MATCH.

1231 In the event that both an If-Match and If-None-Match request header are supplied by the client, the MC
1232 shall respond with HTTP status code 400 – Bad Request – to the client and stop processing the request.
1233 The MC shall not send such a malformed request to the RDE Device.

1234 **7.2.4.2.3 Custom HTTP headers**

1235 The MC shall support custom headers when applied to any Redfish HTTP/HTTPS operation. For
1236 purposes of this specification, an RDE custom header shall be considered as one with a prefix “PLDM-
1237 RDE-“. Unless explicitly specified in this specification, no standard handling is described for RDE custom
1238 headers either in this specification or in [DSP0266](#). All discussion of custom headers in this specification
1239 shall be restricted to HTTP/HTTPS custom headers of this form.

1240 The parameters for custom headers will vary by actual header type.

1241 In order to support RDE custom headers, the MC shall bundle them (including the PLDM-RDE prefix) into
1242 the request message for an invocation of the SupplyCustomRequestParameters command (clause 12.2).
1243 To do so, the MC shall set the HeaderCount request parameter to the number of custom request
1244 parameters. For each RDE custom request parameter n , the MC shall set HeaderName[n] and
1245 HeaderParameter[n] to the name and value of the request parameter, respectively. Custom headers other
1246 than those prefixed “PLDM-RDE-“ shall not be supplied to RDE Devices in this manner.

1247 When the RDE Device receives RDE custom request parameters, it may perform any custom handling for
1248 the parameter. If it does not support a specific RDE custom request parameter received, the RDE Device
1249 shall respond with the ERROR_UNRECOGNIZED_CUSTOM_HEADER completion code.

1250 Similarly, when the RDE Device has custom response parameters to send back to a client, it shall set the
1251 HaveCustomResponseParameters flag in the OperationExecutionFlags response field of the
1252 RDEOperationInit, SupplyCustomRequestParameters, or RDEOperationStatus command to ask the MC
1253 to retrieve these parameters. Then, in response to the RetrieveCustomResponseParameters command
1254 (clause 12.3), the RDE Device shall set the ResponseHeaderCount field to the number of custom
1255 response headers it wants to send back to the client. For each custom response parameter n , the RDE
1256 Device shall set HeaderName[n] and HeaderParameter[n] to the name and value of the response
1257 parameter, respectively.

1258 Following completion of the main Operation, the MC shall check the HaveCustomResponseParameters
1259 flag in the OperationExecutionFlags response field to see if the RDE Device is supplying custom
1260 response headers (which should have a PLDM-RDE prefix). If the flag is set (with value 1b), the MC shall
1261 use the RetrieveCustomResponseParameters command (clause 12.3) to recover them from the RDE
1262 Device. The MC shall then append the recovered headers to the Redfish Operation response.

1263 **7.2.4.2.3.1 PLDM-RDE-Expand-Type**

1264 The MC may optionally support use of the PLDM-RDE-Expand-Type header when it receives a Redfish
1265 HTTP/HTTPS GET operation with the \$expand query option (see clause 7.2.4.3.3) to convey the
1266 expansion type parameter to the RDE Device.

1267 The parameter for this header is the type of expansion to be used in the expansion, one of
1268 EXPAND_DOT ("."), EXPAND_TILDE ("~"), or EXPAND_STAR ("*") and shall match the parameter given
1269 as the value of the \$expand query option. If this header is not supplied to the RDE Device, expansion
1270 shall default to type EXPAND_DOT. If no expansion type is supplied to the \$expand query option, the MC
1271 may either send this header with the default type (EXPAND_DOT) or omit it.

1272 **7.2.4.2.4 ETag response header**

1273 The MC shall provide an ETag header in response to every Redfish HTTP/HTTPS GET or HEAD
1274 operation.

1275 The parameter for this header is an ETag.

1276 In order to support this header, the RDE Device shall generate a digest of the schema data instance after
1277 each modification to the data in accordance with [RFC 7232](#). When the MC begins a GET or HEAD
1278 operation to the RDE Device via a PLDM RDEOperationInit command (clause 12.1), the RDE Device
1279 shall populate the ETag field in the response message to the command where the RDE Operation has
1280 completed (one of RDEOperationInit, SupplyCustomRequestParameters, or RDEOperationStatus) with
1281 this digest.

1282 When it receives an ETag field in the response message for a completed RDE Operation, the MC shall
1283 then populate this header with the digest it receives.

1284 **7.2.4.2.5 Link response header**

1285 The MC shall provide one or more Link headers in response to every Redfish HTTP/HTTPS GET and
1286 HEAD operation as described in [DSP0266](#).

1287 The parameter for this header is a URI.

1288 This header has three forms as described in [DSP0266](#); all three shall be supported by MCs. The handling
1289 for these three forms is detailed in the next three clauses.

1290 No special action is needed on the part of an RDE Device to support any form of the link response
1291 header.

1292 **7.2.4.2.5.1 Schema form**

1293 The MC shall provide a link header with "rel=describedby" to provide a schema link for the data that is or
1294 would be returned in response to a Redfish HTTP/HTTPS GET or HEAD operation. The MC may obtain
1295 this link in any of several manners:

- 1296 • An @odata.context annotation in read data may contain the schema reference.
- 1297 • The MC may have the schema reference cached.
- 1298 • The MC may retrieve the schema reference directly from the PDR encapsulating the instance of
1299 the schema data by invoking the PLDM GetSchemaURI command (clause 11.4).

1300 An example of a schema form link header is as follows; readers are referred to [DSP0266](#) for more detail:

1301

Link: </redfish/v1/JsonSchemas/ManagerAccount.v1_0_2.json>; rel=describedby

7.2.4.2.5.2 Annotation form

The MC should provide a link header to provide an annotation link for the data that is or would be returned in response to a Redfish HTTP/HTTPS GET or HEAD operation. The MC may obtain this link in any of several manners:

- The MC may inspect annotations to determine whether @odata or @Redfish annotations are used.
- The MC may retrieve the schema reference directly from the PDR encapsulating the instance of the schema data by invoking the PLDM GetSchemaURI command (clause 11.4)

An example of an annotation form link header is as follows; readers are referred to [DSP0266](#) for more detail:

Link: < http://redfish.dmtf.org/schemas/Settings.json >

7.2.4.2.5.3 Passthrough form

The MC shall translate link annotations returned from the RDE Device in response to a Redfish HTTP/HTTPS GET operation into link headers. In this form, the MC shall also include the schema path to the link.

An example of a passthrough form link header is as follows; readers are referred to [DSP0266](#) for more detail:

Link: </redfish/v1/AccountService/Roles/Administrator>; path=/Links/Role
--

7.2.4.2.6 Location response header

The MC shall provide a Location header in response to every Redfish HTTP/HTTPS POST that effects a successful create operation. The MC shall also provide a Location header in response to every Redfish Operation that spawns a long-running Task when executed as an RDE Operation.

The parameter for this header is a URI.

In order to support this header for completed create operations, the RDE Device shall populate the NewResourceId response parameter in the response message for the RetrieveCustomResponseParameters command (clause 12.3) with the Resource ID of the newly created collection element. Upon receipt, the MC shall combine this resource ID with the topology information contained in the Redfish Resource PDRs for the targeted PDR up through the device component root to create a local URI portion that it shall then combine with its external management URI for the RDE Device to build a complete URI for the newly added collection element. The MC shall then populate this header with the resulting URI.

In order to support this header for Redfish Operations that spawn long-running Tasks when executed as RDE Operations, the MC shall generate a TaskMonitor URL for the Operation and populate the Location header with the generated URL. See clause 7.2.6 for more details.

7.2.4.2.7 Cache-Control response header

The MC shall provide a Cache-Control header in response to every Redfish HTTP/HTTPS GET or HEAD operation.

In order to support this header for HTTP/HTTPS GET operations, the RDE Device shall mark the CacheAllowed flag in the OperationExecutionFlags field of the response message for the triggering command for the read or head Operation with an indication of the caching status of data read.

When the MC reads the CacheAllowed flag in the OperationExecutionFlags field of the response message for a completed RDE Operation, it shall populate the Cache-Control response header with an

1344 appropriate value. Specifically, if the RDE Device indicates that the data is cacheable, the MC shall
1345 interpret this as equivalent to the value “public” as defined in [RFC 7234](#); otherwise, the MC shall interpret
1346 this as equivalent to the value “no-store” as defined in [RFC 7234](#).

1347 **7.2.4.2.8 Allow response header**

1348 The MC shall provide an Allow header in response to every Redfish HTTP/HTTPS operation that is
1349 rejected by the RDE Device specifically for the reason of being a disallowed operation, giving the
1350 ERROR_NOT_ALLOWED completion code (clause 7.5). The MC shall additionally provide an Allow
1351 response header in response to every GET (or HEAD, if supported) Redfish operation.

1352 In order to support this header, when the RDE Device responds to an RDE command with
1353 ERROR_NOT_ALLOWED, or in response to a GET or HEAD Redfish operation, it shall populate the
1354 PermissionFlags field of its response message with an indication of the operations that are permitted.

1355 When the MC reads the PermissionFlags field of the response message for a completed RDE Operation,
1356 the MC shall populate this header with the supplied information.

1357 **7.2.4.2.9 Retry-After response header**

1358 The MC shall provide a Retry-After header in response to every non-HEAD Redfish Operation that when
1359 conveyed to the RDE Device results in any transient failure (ERROR_NOT_READY; see clause 7.5).

1360 The parameter for this header is the length of time in seconds the client should wait before retrying the
1361 request.

1362 When the RDE Device needs to defer an RDE Operation, it shall return ERROR_NOT_READY in
1363 response to the RDEOperationInit command that begins the Operation. The RDE Device must now
1364 choose whether to supply a specific deferral timeframe or to use the default deferral timeframe. To specify
1365 a specific deferral timeframe, the RDE Device shall also set the HaveCustomResponseParameters flag in
1366 the OperationExecutionFlags response field of the RDEOperationInit command to inform the MC that it
1367 should retrieve deferral information. Then, if it did set the HaveCustomResponseParameters flag, in
1368 response to the RetrieveCustomResponseParameters command (clause 12.3), the RDE Device shall set
1369 the DeferralTimeframe and DeferralUnits parameters appropriately to indicate how long it is requesting
1370 the client to wait before resubmitting the request.

1371 As an alternative to specifying a deferral timeframe via the response message for
1372 RetrieveCustomResponseParameters, the RDE Device may skip setting the
1373 HaveCustomResponseParameters flag in the OperationExecutionFlags response field of the
1374 RDEOperationInit command to request that the MC supply a default deferral timeframe on its behalf.

1375 When it receives the response to the RDEOperationInit command, the MC shall check the
1376 HaveCustomResponseParameters flag in the OperationExecutionFlags response field to see if the RDE
1377 Device has an extended response. If the flag is set (with value 1b), the MC shall use the
1378 RetrieveCustomResponseParameters command (clause 12.3) to recover the deferral timeframe from the
1379 DeferralTimeframe and DeferralUnits fields of the response message. If the flag was not set, or if the RDE
1380 Device supplied an unknown deferral timeframe (0xFF), the MC shall use a default value of 5 seconds. It
1381 shall then populate this header with the deferral value.

1382 Both the MC and RDE Device shall be prepared for possibility that the client may retry the operation
1383 before this deferral timeframe elapses: Operations can be re-initiated by impatient end users.

1384 **7.2.4.3 Redfish Operation request query options**

1385 In addition to HTTP/HTTPS headers, the standard Redfish management protocol defines several query
1386 options that a client may specify in a URI to narrow the request in Redfish GET Operations. For any query
1387 option not listed here, the MC may support it in a fashion as described in [DSP0266](#).

1388

Table 35 – Redfish operation request query options

Query Option	Clause	Description	Example
\$skip	0	Integer indicating the number of Members in the Resource Collection to skip before retrieving the first resource.	http://resourcecollection?\$skip=5
\$top	7.2.4.3.2	Integer indicating the number of Members to include in the response.	http://resourcecollection?\$top=30
\$expand	7.2.4.3.3	Expand schema links, gluing data together into a single response. Collection: Collection by name * = all links . = all but those in Links	http://resourcecollection?\$expand=collection(\$levels=4)
\$levels	7.2.4.3.4	Qualifier on \$expand; number of links to expand out	http://resourcecollection?\$expand=collection(\$levels=4)
\$select	7.2.4.3.5	Top-level or a qualifier on \$expand; says to return just the specified properties	http://resourcecollection?\$select=FirstName,LastName http://resourcecollection?\$expand=collection(\$select=FirstName,LastName;\$levels=4)
excerpt	7.2.4.3.6	Returns a subset of the resource's properties that match the defined Excerpt schema annotation.	http://resource?excerpt
\$filter	n/a	Limit results of a READ operation to a subset of the resource collection's members based on a \$filter expression that follows the OData-Protocol Specification.	n/a: \$filter is not supported in this specification
only	n/a	Applies to resource collections. If the target resource collection contains exactly one member, clients can use this query parameter to return that member's resource.	n/a: only is not supported in this specification

1389 Support requirements for query parameters are described in Table 36.

1390

Table 36 – Query parameter support requirement

Query Option	RDE Device	MC
\$skip	Optional	Should support
\$top	Optional	Should support
\$expand	Optional	Should support
\$levels	Optional	May support
\$select	Optional	May support
\$filter	Should not support	May support

1391 **7.2.4.3.1 \$skip query option**1392 The MC should support \$skip query options when provided as part of a target URI for a Redfish
1393 HTTP/HTTPS GET operation.

1394 The parameter for this query option is an integer representing the number of members of a resource
1395 collection to skip over. See [DSP0266](#) for more details on the usage of \$skip.

1396 To support this query option, the MC shall supply the \$skip parameter in the CollectionSkip field of the
1397 SupplyCustomRequestParameters (clause 12.2) request message. In the event that this query option is
1398 not supplied as part of the target URI for an HTTP/HTTPS GET operation, the MC shall supply a value of
1399 zero in this field if it otherwise needs to supply extended request parameters; it shall not send the
1400 SupplyCustomRequestParameters just to supply a value of zero for the CollectionSkip field.

1401 When processing an RDE read Operation for a resource collection, the RDE Device shall check the
1402 CollectionSkip parameter from the SupplyCustomRequestParameters request message to determine the
1403 number of members to skip over in its response, per [DSP0266](#). In the event that the MC did not indicate
1404 the presence of extended request parameters, the RDE Device shall interpret this as a CollectionSkip
1405 value of zero. If the parameter for \$skip equals the number of elements in the collection, the RDE Device
1406 shall return an empty list. If the parameter for \$skip exceeds the number of elements in the collection, the
1407 RDE Device shall return ERROR_OPERATION_FAILED and, in accordance with the Redfish standard
1408 [DSP0266](#) respond with an annotation specifying that the value is invalid (see
1409 QueryParameterOutOfRange in the Redfish base message registry).

1410 7.2.4.3.2 \$top query option

1411 The MC should support \$top query options when provided as part of the target URI for a Redfish
1412 HTTP/HTTPS GET operation.

1413 The parameter for this query option is an integer representing the number of members of a resource
1414 collection to return. See [DSP0266](#) for more details on the usage of \$top. If the parameter for \$top
1415 exceeds the remaining number of members in a resource collection, the number returned shall be
1416 truncated to those remaining. For a \$top value of zero, the response shall consist of an empty list.

1417 To support this query option, the MC shall supply the \$top parameter in the CollectionTop field of the
1418 SupplyCustomRequestParameters (clause 12.2) request message. In the event that this query option is
1419 not supplied as part of the target URI for an HTTP/HTTPS GET operation, the MC shall supply a value of
1420 0xFFFF in this field; it shall not send the SupplyCustomRequestParameters just to supply a value of
1421 unlimited for the CollectionTop field.

1422 When processing an RDE read Operation for a resource collection, the RDE Device shall check the
1423 CollectionTop parameter from the SupplyCustomRequestParameters request message to determine the
1424 number of members to respond with, per [DSP0266](#). The RDE Device shall interpret a value of 0xFFFF as
1425 indicating that there is no limit to the number of members it should return for the referenced resource
1426 collection. In the event that the MC did not indicate the presence of extended request parameters, the
1427 RDE Device shall interpret this as a CollectionTop value of unlimited.

1428 7.2.4.3.3 \$expand query option

1429 The MC should support \$expand query options when provided as part of the target URI for a Redfish
1430 HTTP/HTTPS GET operation.

1431 The parameter for this query option is a string representing the links (Navigation properties) to expand in
1432 place, “gluing together” the results of multiple reads into a single JSON response payload. This parameter
1433 may be an absolute string specifying the exact link to be expanded, or it may be any of three wildcards.
1434 The first wildcard, an asterisk (*), means that all links should be expanded. The second wildcard, a dot (.),
1435 means that subordinate links (those that are directly referenced i.e., not in the Links Property section of
1436 the resource) should be expanded. The third wildcard, a tilde (~), means that dependent links (those that
1437 are not directly referenced i.e., in the Links Property section of the resource) should be expanded. See
1438 [DSP0266](#) for more details on the usage of \$expand.

1439 To support an expansion type wildcard received from the Redfish Client, the MC should send the PLDM-
1440 RDE-Expand-Type custom header described in clause 7.2.4.2.3.1 to the RDE Device via the
1441 SupplyCustomRequestParameters command of clause 12.2.

1442 If the \$levels query option qualifier is not present in conjunction with the \$expand query option, the MC
1443 shall treat this as equivalent to \$levels=1.

1444 To support the \$expand query option, the RDE Device should concatenate linked resource data into the
1445 BEJ data it returns for an RDE read Operation, using the bejResourceLinkExpansion PLDM data type
1446 described in Clause 5.3.23.

1447 **7.2.4.3.4 \$levels query option qualifier**

1448 The MC should support the \$levels qualifier to the \$expand query option when provided as part of the
1449 target URI for a Redfish HTTP/HTTPS GET operation or when provided implicitly by having \$expand
1450 provided as part of a Redfish HTTP/HTTPS GET operation without having the \$levels query option
1451 qualifier supplied.

1452 The parameter for this query option is an integer representing the number of schema links to expand into.
1453 If no \$level qualifier is present, the MC shall interpret this as equivalent to \$levels=1.

1454 To support this parameter, the MC can select between two choices: passing it on to the RDE Device or
1455 supporting it itself. The method by which this choice is made is implementation-specific and out of scope
1456 for this specification. If the RDE Device indicates that it cannot support \$levels expansion by setting the
1457 expand_support bit to zero in the DeviceCapabilitiesFlags in the response message to the
1458 NegotiateRedfishParameters command (clause 11.1), or if the expansion type is not "All Links" (see
1459 clause 7.2.4.3.3), the MC shall not select passing it to the RDE Device.

1460 If the MC chooses to pass this query option to the RDE Device, it shall transmit the supplied value to the
1461 RDE Device via the SupplyCustomRequestParameters command in the LinkExpand parameter.

1462 If the MC chooses to handle this query option itself, it shall recursively issue reads to "expand out" data
1463 for links embedded in data it reads. Such links may be identified during the BEJ decode process as tuples
1464 with a format of bejResourceLink (clause 5.3.21). The corresponding value of the node represents the
1465 Resource ID for the Redfish Resource PDR representing the data to embed within the structure of data
1466 already read. The \$levels qualifier dictates the depth of recursion for this process.

1467 When the RDE Device receives a LinkExpand value of greater than zero in extended request parameters
1468 as part of an RDE read operation, it shall "expand out" all resource links (as defined in [DSP0266](#)) to the
1469 indicated depth by encoding them as bejResourceLinkExpansions in the response BEJ data for the
1470 command. If the RDE Device previously did not set the expand_support flag in the
1471 DeviceCapabilitiesFlags field of the NegotiateRedfishParameters command, it may instead ignore the
1472 value (treating it as zero).

1473 Implementers should refer to [DSP0266](#) for more details and caveats to be applied when expanding links
1474 with \$levels > 1.

1475 **7.2.4.3.5 \$select query option qualifier**

1476 The MC may support \$select as a qualifier to the \$expand query option or as a standalone query option,
1477 provided in either case as part of the target URI for a Redfish HTTP/HTTPS GET operation.

1478 The parameter for this query option is a string containing a comma-separated list of properties to be
1479 retrieved from the GET operation; the caller is asking that all other properties be suppressed. See
1480 [DSP0266](#) for more details on the usage of \$select.

1481 If it supports this parameter, the MC should perform the GET operation normally up to the point of
1482 retrieving BEJ-formatted data from the RDE Device. When decoding the BEJ data, however, the MC
1483 should silently discard any property not part of the \$select list.

1484 No action is needed on the part of an RDE Device to support this query option.

1485 **7.2.4.3.6 Excerpt query option**

1486 The MC may support the excerpt query option when provided as part of a target URI for a Redfish
1487 HTTP/HTTPS GET operation. There is no parameter for this command.

1488 To support this parameter, the MC shall set the excerpt_flag in the OperationFlags field of the
1489 RDEOperationInit request command. Thereafter, no special treatment is required on the part of the MC.

1490 When the RDE Device is flagged that the client requested an excerpt, it may support the request by
1491 restricting properties returned in the read to those flagged with the excerpt schema annotation. If the
1492 schema does not contain any such flagged properties, or if the RDE Device does not support the excerpt
1493 query option, it shall return the complete resource.

1494 Further details of the excerpt query option may be found in [DSP0266](#).

1495 **7.2.4.4 HTTP/HTTPS status codes**

1496 The MC shall comply with [DSP0266](#) in all matters pertaining to the HTTP/HTTPS status codes returned
1497 for Redfish GET, PATCH, PUT, POST, DELETE, and HEAD operations. Typical status codes for
1498 operational errors may be found in clause 7.5.

1499 **7.2.4.5 Multihosting and Operations**

1500 A single RDE Device may find that it is attached to multiple MCs. This can introduce complications from
1501 concurrency if conflicting Operations are issued and requires an RDE Device to decide whether an
1502 Operation should be visible to an MC other than the one that issued it. Support for multiple MCs is out of
1503 scope for this specification. In particular, the behavior of the RDE Device in the face of concurrent
1504 commands from multiple MCs is undefined.

1505 **7.2.5 PLDM RDE Events**

1506 An Event is an abstract representation of any happening that transpires in the context of the RDE Device,
1507 particularly one that is outside of the normal command request/response sequence. A Redfish Message
1508 Event consists of JSON data that includes elements such as the index of a standardized text string and a
1509 collection of parameters that provide clarification of the specifics of the Event that has transpired. The full
1510 schema for Events may be found in the standard Redfish Message schema; additionally, OEM extensions
1511 to this schema are possible.

1512 In this specification, a second class of events, Task Executed Events, allow RDE Devices to report that a
1513 Task has finished executing and that the MC should retrieve Operation results. The data for these events
1514 includes elements such as the Operation identifier and the resource with which the Operation is
1515 associated.

1516 As with any other PLDM eventing, the RDE Device advertises that it supports Events by listing support for
1517 the PLDM for Platform Monitoring and Control SetEventReceiver command (see [DSP0248](#)). The MC, for
1518 its part, may then select between two methods by which it will know that Events are available. If the MC
1519 configured the RDE Device to use asynchronous events through the SetEventReceiver command, the
1520 RDE Device shall use the PLDM for Platform Monitoring and Control PlatformEventMessage command
1521 (see [DSP0248](#)) to inform the MC by sending the Event directly. Otherwise, the RDE Device can be
1522 configured to polling mode using the same SetEventReceiver command. The MC uses the PLDM for
1523 Platform Monitoring and Control PollForPlatformEventMessage command (see [DSP0248](#)) for this

1524 purpose. The selection of any polling interval is determined by the MC and is outside the scope of this
1525 specification.

1526 Whether retrieved synchronously or asynchronously, once the MC gets the Event, it may process it.
1527 Redfish Message Events are packaged using the redfishMessageEvent eventClass; Task Executed
1528 Events are packaged using the redfishTaskExecutedEvent eventClass (see [DSP0248](#) for both
1529 eventClasses).

1530 Handling of Task Executed Events is described with Tasks in clause 7.2.6. For Redfish Message Events,
1531 the MC may decode the BEJ-formatted payload of Event data using the appropriate Event schema
1532 dictionary specific to the PDR from which the message was sent.

1533 For a more detailed view of the Event lifecycle, see clause 9.3.

1534 NOTE Events are optional in standard Redfish; however, support for Task Executed Events is mandatory in this
1535 specification if the RDE Device supports asynchronous execution for long-running Operations.

1536 7.2.5.1 [MC] Event subscriptions

1537 In Redfish, a client may request to be notified whenever a Redfish Event occurs. Per [DSP0266](#), to do so,
1538 the client uses a Redfish CREATE operation to add a record to the EventSubscription collection. This
1539 record in turn contains information on the various Event types that the client wishes to receive Events for.
1540 To unsubscribe, the client uses a Redfish DELETE operation to remove its record. Among other
1541 properties, the EventSubscription record contains a URI to which the Event should be forwarded. MCs
1542 that support Events shall support at least one Redfish event subscription.

1543 Event types are global across all schemas; there is no provision at this time ([DSP0266](#) v1.6) in Redfish
1544 for a client to subscribe to just one schema at a time. Further, there is generally no capacity for an RDE
1545 Device to send an HTTP/HTTPS record directly to an external recipient. Events are optional in Redfish;
1546 however, if the MC chooses to provide Event subscription support, it must comply with the following
1547 requirements:

- 1548 • The MC shall provide full support for the EventSubscription collection as a Redfish Provider per
1549 [DSP0266](#).
- 1550 • When it receives an Event subscription request (in the form of a Redfish CREATE operation on
1551 the EventSubscription collection), the MC shall parse the EventTypes array property of the
1552 request to identify the type or types of Events the client is interested in receiving
- 1553 • When the MC receives a Redfish Message Event from an RDE Device, it shall check the
1554 EventType of the Event received against the desired EventTypes for each active client. For
1555 each match, the MC shall forward the Event (translating any @Message.ExtendedInfo
1556 annotations, of course, from BEJ to JSON) to the client as a standard Redfish Provider for the
1557 Event service.

1558 7.2.6 Task support

1559 In PLDM for Redfish Device Enablement, every Redfish HTTP/HTTPS operation is effected as an RDE
1560 Operation. Most Operations, once sent to the RDE Device for execution, may be executed quickly and
1561 the results sent directly in the response message to the request message that triggered them.

1562 It may however transpire that in order for an RDE Device to complete an Operation, it requires more time
1563 than the available window within which the RDE Device is required to send a response. In this case, the
1564 RDE Device has two possible paths to follow. If the current number of extant Tasks is less than the RDE
1565 Device/MC capability intersection (as determined from the call to NegotiateRedfishParameters; see
1566 clause 11.1), the RDE Device shall mark the Operation as a long-running Task and execute it
1567 asynchronously. Otherwise, the RDE Device shall return ERROR_CANNOT_CREATE_OPERATION in
1568 its response message to indicate that no new Task slots are available (see clause 7.5).

1569 While the internal data structures used by an RDE Device to manage an Operation are outside the scope
 1570 of this specification, they should include at a minimum the rdeOpID assigned (usually by the MC) when
 1571 the Operation was first created. This allows the MC to reference the Task in subsequent commands to kill
 1572 it (RDEOperationKill, clause 12.6) or query its status (RDEOperationStatus, clause 12.5).

1573 For its part, the MC shall provide full support for the Task collection as a Redfish Provider per [DSP0266](#).
 1574 When the MC finds that an Operation has spawned a Task, it shall perform the following steps in order to
 1575 comply with the requirements of [DSP0266](#):

1576 2) The MC shall instantiate a new TaskMonitor URL and a new member of the Task collection. The
 1577 TaskMonitor URL should incorporate or reference (such as via a lookup table) the following data so
 1578 that it can map from the TaskMonitor URL back to the correct Redfish resource – and thus the
 1579 correct dictionary – for providing status query updates:

- 1580 a) The ResourceID for the resource to which the RDE Operation was targeted
- 1581 b) The rdeOpID for the Operation itself

1582 2) The MC shall return response code 202, Accepted, to the client and include the Location response
 1583 header populated with the TaskMonitor URL.

1584 3) In response to a subsequent Redfish GET Operation applied to the TaskMonitor URL or to the Task
 1585 collection member, the MC shall invoke the RDEOperationStatus (see clause 12.5) command to
 1586 obtain the latest status for the Operation and communicate it to the client in accordance with
 1587 [DSP0266](#). If the GET was applied to a TaskMonitor URL and the Operation has been completed, the
 1588 MC shall supply the completed results to the client.

1589 a) If the result of the RDEOperationStatus command was that the Operation has finished
 1590 execution, the MC shall delete both the TaskMonitor URL and the Task collection member
 1591 associated with the Operation.

1592 4) In response to a Redfish DELETE Operation applied to the TaskMonitor URL or to the Task
 1593 collection member, the MC shall attempt to abort the associated Operation via the RDEOperationKill
 1594 (see clause 12.6) command. It shall then remove both the TaskMonitor URL and the Task collection
 1595 member.

1596 5) If the RDE Operation finishes before the client polls the TaskMonitor URL, the MC may collect and
 1597 store the results of the Operation.

1598 a) In accordance with [DSP0266](#), the MC should retain Operation results until the client retrieves
 1599 them. It may refuse to accept further Operations until previous results have been claimed.

1600 b) If the client attempts to collect Operation results after the MC has discarded them, the MC shall
 1601 respond with an error HTTP status code as defined in [DSP0266](#).

1602 When the RDE Device finishes execution of a Task, it generates a Task Executed Event to inform the MC
 1603 of this status change. The MC can then retrieve the results (via RDEOperationStatus) and eventually
 1604 forward them to the client. To mark the Task as complete and allow the RDE Device to discard any
 1605 internal data structures used to manage the Task, the MC shall call RDEOperationComplete (clause
 1606 12.4).

1607 For a more detailed overview of the Operation/Task lifecycle from the MC's perspective, see clause
 1608 7.2.4.1.1.3. A detailed flowchart of the Operation/Task lifecycle may be found in clause Error! Reference
 1609 source not found., and a finite state machine for the Task lifecycle (from the RDE Device's perspective)
 1610 may be found in clause 9.2.3.

1611 7.3 Type code

1612 Refer to [DSP0245](#) for a list of PLDM Type Codes in use. This specification uses the PLDM Type Code
 1613 000110b as defined in [DSP0245](#).

1614 **7.4 Transport protocol type supported**

1615 PLDM can support bindings over multiple interfaces; refer to [DSP0245](#) for the complete list. All transport
1616 protocol types can be supported for the commands defined in Table 50.

1617 **7.5 Error completion codes**

1618 Table 37 lists PLDM completion codes for Redfish Device Enablement. The usage of individual error
1619 completion codes is defined within each of the PLDM command clauses. When communicating results
1620 back to the client, implementations should provide HTTP error codes as described below.

1621 **Table 37 – PLDM for Redfish Device Enablement completion codes**

Value	Name	Description	HTTP Error Code
Various	PLDM_BASE_CODES	Refer to DSP0240 for a full list of PLDM Base Code Completion values that are supported.	See below.
128 (0x80)	ERROR_BAD_CHECKSUM	A transfer failed due to a bad checksum and should be restarted.	MC should retry transfer. If retry fails, 500 Internal Server Error
129 (0x81)	ERROR_CANNOT_CREATE_OPERATION	An Operation-based command failed because the RDE Device could not instantiate another Operation at this time.	503 Service Unavailable
130 (0x82)	ERROR_NOT_ALLOWED	The client and/or MC is not allowed to perform the requested Operation.	405 Method Not Allowed
131 (0x83)	ERROR_WRONG_LOCATION_TYPE	A Create, Delete, or Action Operation attempted against a location that does not correspond to the right type.	405 Method Not Allowed
132 (0x84)	ERROR_OPERATION_ABANDONED	An Operation-based command other than completion was attempted with an Operation that has timed out waiting for the MC to progress it in the Operation lifecycle.	410 Gone
133 (0x85)	ERROR_OPERATION_UNKILLABLE	An attempt was made to kill an Operation that has already finished execution or that cannot be aborted.	409 Conflict
134 (0x86)	ERROR_OPERATION_EXISTS	An Operation initialization was attempted with an rdeOpID that is currently active.	N/A – MC retries with a new rdeOpID
135 (0x87)	ERROR_OPERATION_FAILED	An Operation-based command other than completion was attempted with an Operation that has encountered an error in the Operation lifecycle.	400 Bad Request
136 (0x88)	ERROR_UNEXPECTED	A command was sent out of context, such as sending SupplyCustomRequestParameters when Operation initialization flags did not indicate that the Operation requires them	500 Internal Server Error

Value	Name	Description	HTTP Error Code
138 (0x89)	ERROR_UNSUPPORTED	An attempt was made to initialize an operation not supported by the RDE Device, to write to a property that the RDE Device does not support, or a command was issued containing a text string in a format that the recipient cannot interpret.	400 Bad Request
144 (0x90)	ERROR_UNRECOGNIZED_CUSTOM_HEADER	The RDE Device received a custom PLDM-RDE header (via SupplyCustomRequestParameters) that it does not support	412 Precondition Failed
145 (0x91)	ERROR_ETAG_MATCH	The RDE Device received one or more ETags that did not match an If-Match or If-None-Match request header	412, Precondition Failed (If-Match) or 304, not modified (If-None-Match)
146 (0x92)	ERROR_NO SUCH RESOURCE	An Operation command was invoked with a resource ID that does not exist	404, Not Found
147 (0x93)	ETAG_CALCULATION_ONGOING	Calculating the ETag in response to the GetResourceETag command is taking too long to provide an immediate response	N/A – MC retries with the same command later

1622 HTTP Error codes returned when Operations complete with standard PLDM completion codes should be
 1623 as follows:

1624

Table 38 – HTTP codes for standard PLDM completion codes

Name	Description	HTTP Error Code
SUCCESS	Normal success	200 Success, 202 Accepted for an Operation that spawned a Task, or 204 No Content for an Action that has no response
ERROR	Generic error	400 Bad Request
ERROR_INVALID_DATA	Invalid data or a bad parameter value	500 Internal Server Error
ERROR_INVALID_LENGTH	Incorrectly formatted request method	500 Internal Server Error
ERROR_NOT_READY	Device transiently busy	503 Service Unavailable
ERROR_UNSUPPORTED_PLDM_CMD	Command not supported	501 Not Implemented

Name	Description	HTTP Error Code
ERROR_INVALID_PLDM_TYPE	Not a supported PLDM type	501 Not Implemented

1625 7.6 Timing specification

1626 Table 39 below defines timing values that are specific to this document. The table below defines the
 1627 timing parameters defined for the PLDM Redfish Specification. In addition, all timing parameters listed in
 1628 [DSP0240](#) for command timeouts, command response times, and number of retries shall also be followed.

1629 **Table 39 – Timing specification**

Timing specification	Symbol	Min	Max	Description
PLDM Base Timing	PNx PTx (see DSP0240)	(See DSP0240)	(See DSP0240)	Refer to DSP0240 for the details on these timing values.
Operation/Transfer abandonment	T _{abandon}	120 seconds	none	<p>Time between when the RDE Device is ready to advance an Operation through the Operation lifecycle and when the MC must have initiated the next step. If the MC fails to do so, the RDE Device may consider the Operation as abandoned.</p> <p>Also used in follow up to a GetSchemaDictionary command to mark the time between when the MC receives one chunk of dictionary data and when it must request the next chunk. If the MC fails to do so, the RDE Device may consider the transfer as abandoned.</p>

1630 8 Binary Encoded JSON (BEJ)

1631 This clause defines a binary encoding of Redfish JSON data that will be used for communicating with
 1632 RDE Devices. At its core, BEJ is a self-describing binary format for hierarchical data that is designed to
 1633 be straightforward for both encoding and decoding. Unlike in ASN.1, BEJ uses no contextual encodings;
 1634 everything is explicit and direct. While this requires the insertion of a bit more metadata into BEJ encoded
 1635 data, the tradeoff benefit is that no lookahead is required in the decoding process. The result is a
 1636 significantly streamlined representation that fits in a very small memory footprint suitable for modern
 1637 embedded processors.

1638 8.1 BEJ design principles

1639 The core design principles for BEJ are focused around it being a compact binary representation of JSON
 1640 that is easy for low-power embedded processors to encode, decode, and manipulate. This is important
 1641 because these ASICs typically have highly limited memory and power budgets; they must be able to
 1642 process data quickly and efficiently. Naturally, it must be possible to fully reconstruct a textual JSON
 1643 message from its BEJ encoding.

- 1644 The following design principles guided the development of BEJ:
- 1645 1) It must be possible to support full expressive range of JSON.
- 1646 2) The encoding should be binary and compact, with as much of the encoding as possible
1647 dedicated to the JSON data elements. The amount of space afforded to metadata that conveys
1648 elements such as type format and hierarchy information should be carefully limited.
- 1649 2) There is no need to support multiple encoding techniques for one type of data; there is therefore
1650 no need to distinguish which encoding technique is in use.
- 1651 3) Schema information – such as the names of data items – does not need to be encoded into BEJ
1652 because the recipient can use a prior knowledge of the data organization to determine semantic
1653 information about the encoded data. In contrast to JSON, which is unordered, BEJ must adopt
1654 an explicit ordering for its data to support this goal.
- 1655 4) The need for contextual awareness should be minimized in the encoding and decoding process.
1656 Supporting context requires extra lookup tables (read: more memory) and delays processing
1657 time. Everything should be immediately present and directly decodable. Giving up a few bytes
1658 of compactness in support of this goal is a worthwhile tradeoff.

1659 **8.2 SFLV tuples**

1660 Each piece of JSON data is encoded as a tuple of PLDM type `bejTuple` and consists of the following:

- 1661 1) Sequence number: the index within the canonical schema at the current hierarchy level for the
1662 datum. For collections and arrays, the sequence number is the 0-based array index of the
1663 current element.
- 1664 2) Format: the type of data that is encoded.
- 1665 3) Length: the length in bytes of the data.
- 1666 4) Value: the actual data, encoded in a format-specific manner.

1667 These tuple elements collectively describe a single piece of JSON data; each piece of JSON data is
1668 described by a separate tuple. Requirements for each tuple element are detailed in the following clauses.

1669 SFLV tuples are represented by elements of the `bejTuple` PLDM type defined in clause 5.3.5.

1670 **8.2.1 Sequence number**

1671 The Sequence Number tuple field serves as a stand-in for the JSON property name assigned to the data
1672 element the tuple encodes. Sequence numbers align to name strings contained within the dictionary for a
1673 given schema. Sequence numbers are represented by elements of the `bejTupleS` PLDM type defined in
1674 clause 5.3.6.

1675 The low-order bit of a sequence number shall indicate the dictionary to which it belongs according to the
1676 following table:

1677 **Table 40 – Sequence number dictionary indication**

Bit Pattern	Dictionary
0b	Main Schema Dictionary (as was defined in the <code>bejEncoding</code> PLDM object for this tuple)
1b	Annotation Dictionary

1678 **8.2.2 Format**

1679 The Format tuple field specifies the kind of data element that the tuple is representing.

1680 Formats are represented by elements of the `bejTupleF` PLDM type defined in clause 5.3.7.

1681 **8.2.3 Length**

1682 The Length tuple field details the length in bytes of the contents of the Value tuple field.

1683 Lengths are represented by elements of the `bejTupleL` PLDM type defined in clause 5.3.8.

1684 **8.2.4 Value**

1685 The Value tuple field contains an encoding of the actual data value for the JSON element described by
 1686 this tuple. The format of the value tuple field is variable but follows directly from the format code in the
 1687 Format tuple field.

1688 The following JSON data types are supported in BEJ:

1689 **Table 41 – JSON data types supported in BEJ**

BEJ Type	JSON Type	Description
Null	null	An empty data type
Integer	number	A whole number: any element of JSON type number that contains neither a decimal point nor an exponent
Enum	enum	An enumeration of permissible values in string format
String	string	A null-terminated UTF-8 text string
Real	number	A non-whole number: any element of JSON type number that contains at least one of a decimal point or an exponent
Boolean	boolean	Logical true/false
Bytestring	string (of base-64 encoded data)	Binary data
Set	No named type; data enclosed in { }	A named collection of data elements that may have differing types
Array	No named type; data enclosed in []	A named collection of zero or more copies of data elements of a common type
Choice	special	The ability of a named data element to be of multiple types
Property Annotation	special	An annotation targeted to a specific property, in the format <code>property@annotation</code>
Unrecognized	special	Used to perform a pass-through encoding of a data element for which the name cannot be found in a dictionary for the corresponding schema
Schema Link	special	Used to capture JSON references to external schemas
Expanded Schema Link	special	Used to expand data from a linked external schema

1690 If the deferred_binding flag (see the `bejTupleF` PLDM type definition in clause 5.3.7) is set, the string
 1691 encoded in the value tuple element contains substitution macros that the MC is to supply on behalf of the
 1692 RDE Device when populating a message to send back to the client. See clause 8.3 for more details.

1693 Values are represented by elements of the `bejTupleV` PLDM type defined in clause 5.3.9.

1694 8.3 Deferred binding of data

1695 The data returned to a client from a Redfish operation typically contains annotation metadata that specify
 1696 URIs and other bits of information that are assigned by the MC when it performs RDE Device discovery
 1697 and registration. In practice, the only way for an RDE Device to know the values for these annotations
 1698 would be for it to somehow query the MC about them. Instead, we define substitution macros that the
 1699 RDE Device may use to ask the MC to supply these bits of information on its behalf. RDE Devices shall
 1700 not invoke substitution macros for information that they know and can provide themselves.

1701 All substitution macros are bracketed with the percent sign (%) character. While it would in theory be
 1702 possible for the MC to check every string it decodes for the presence of this escape character, in practice
 1703 that would be an inefficient waste of MC processing time. Instead, the RDE Device shall flag any string
 1704 containing substitution macros with the deferred binding bit set to inform the MC of their presence; the
 1705 MC shall only perform macro substitution if the deferred binding bit is set. The MC shall support the
 1706 deferred bindings listed in Table 42.

1707 **Table 42 – BEJ deferred binding substitution parameters**

Macro	Data to be substituted	Example substitutions
<code>%%</code>	A single % character	%
<code>%L<resource-ID></code>	The MC-assigned URI of an RDE Provider defined resource (specified by a resource ID within the target PDR), or <code>/invalid.PDR<resource-ID></code> if unrecognized resource ID	<code>/invalid.PDR123</code>
<code>%P<resource-ID>.PAGE<pagination-offset></code>	The MC-assigned URI of an RDE Provider defined resource (specified by a resource ID within the target PDR) with a given numerical pagination offset, or <code>/invalid.PDR<resource-ID>.PAGE<pagination-offset></code> if unrecognized resource ID or pagination offset < 1	<code>/invalid.PDR101.PAGE-1</code>
<code>%PD</code>	The MC-assigned URI for an MC-managed PCIeDevice.PCleDevice correlating to this RDE Device, or <code>/invalid.PCleDevice</code> if one cannot be identified. If the RDE Device manages its own PCIeDevice.PCleDevice resource, it shall use the <code>%L</code> binding when referring to it,	<code>/invalid.PCleDevice</code>

Macro	Data to be substituted	Example substitutions
%PF<function-info>	<p>The MC-assigned URI for an MC-managed PCIeFunction.PcieFunction correlating to the RDE Device's function matching function-info, or /invalid.PcieFunction.<function-info> if one cannot be identified.</p> <p>Function-info shall be a string of lower-case hexadecimal digits corresponding to the PCIe function number for the function.</p> <p>If the RDE Device manages its own PCIeFunction.PcieFunction resource, it shall use the %L binding when referring to it,</p>	/invalid.PcieFunction.nonhexdigits /redfish/v1/chassis/1/PCIeDevices/NIC/PCIeFunctions/1
%PI	<p>The MC-assigned URI for an MC-managed PCIeDevice.PcieInterface correlating to this RDE Device, or /invalid.PcieInterface if one cannot be identified.</p> <p>If the RDE Device manages its own PCIeDevice.PcieInterface resource, it shall use the %L binding when referring to it</p>	/invalid.PcieInterface
%S	The MC-assigned link to the ComputerSystem resource within which the RDE Device is located	/redfish/v1/Systems/437XR1138R2
%C	The MC-assigned link to the Chassis resource within which the RDE Device is located	/redfish/v1/Chassis/1U
%M	The metadata URL for the service	/redfish/v1/\$metadata
%T<resource-ID>.<n>	The MC-assigned target URI for the n th Action from the Redfish Action PDR or PDRs linked to a resource within a Redfish Resource PDR, or "/invalid.<resource-ID>.<n>" if no such action exists	/redfish/v1/Systems/437XR1138R2/Storage/1/Actions/Storage.SetEncryptionKey /invalid.123.6
%I<resource-ID>	The MC-assigned instance identifier for the collection element representing an RDE Device (specified by the resource ID of the target PDR), or "invalid" if the PDR does not correspond to a resource immediately contained within a collection managed by the MC	437XR1138R2 invalid
%U	The UEFI Device Path assigned to the RDE Device by the MC and/or BIOS	PciRoot(0x0)/Pci(0x1,0x0)/Pci(0x0,0x0)/Scsi(0xA, 0x0)
%.	Terminates a previous substitution. Shall be used only in the event that numeric data immediately follows a %T, %P, or %L macro	n/a
Any other character preceded by a % character	None – the MC shall pass the sequence exactly as found	%p %X

1708 8.4 BEJ encoding

1709 This clause presents implementation considerations for the BEJ encoding process. For standard resource
1710 encoding (as opposed to annotations), the BEJ conversion dictionary is built to encode the same
1711 hierarchical data format as the schema itself. Implementations should therefore track their context inside
1712 the dictionary in parallel with tracking their location in the data to be encoded. While not mandatory, a
1713 recursive implementation will prove in most cases to be the easiest approach to realize this tracking.

1714 Like with JSON encodings of data, there is no defined ordering for properties in BEJ data; encoders are
1715 therefore free to encode properties in any order.

1716 8.4.1 Conversion of JSON data types to BEJ

1717 Recognition of [JSON](#) data types enables them to be encoded properly. In Redfish, every property is
1718 encoded in the format “property_name” : property_value. Whitespace between syntactic elements is
1719 ignored in JSON encodings.

1720 8.4.1.1 JSON objects

1721 A JSON object consists of an opening curly brace (“{”), zero or more comma-separated properties, and
1722 then a closing curly brace (“}`). JSON objects shall be encoded as BEJ sets with the properties inside the
1723 curly braces encoded recursively as the value tuple contents of the BEJ set. Following the precedent
1724 established in JSON, the properties contained within a JSON object may be encoded in BEJ in any order.
1725 In particular, the encoding order for a collection of properties is not required to match their respective
1726 sequence numbers.

1727 8.4.1.2 JSON arrays

1728 A JSON array consists of an opening square brace (“[”), zero or more comma-separated JSON values all
1729 of a common data type (typically objects in Redfish), and then a closing square brace. JSON arrays shall
1730 be encoded as BEJ arrays with the data inside the square braces encoded recursively as instances of the
1731 value tuple contents of the BEJ array. The immediate contents of a JSON array shall be encoded in order
1732 corresponding to their array indices.

1733 The sequence numbers for BEJ array immediate child elements shall match the zero-based array index
1734 of the children. These sequence numbers are not represented in the dictionary; it is the responsibility of a
1735 BEJ encoder/decoder to understand that this is how array data instances are handled.

1736 8.4.1.3 JSON numbers

1737 In JSON, there is no distinction between integer and real data; both are collected together as the number
1738 type. For BEJ, numeric data shall be encoded as a BEJ integer if it contains neither a decimal point nor
1739 an exponentiation marker ('e' or 'E') and as a BEJ real otherwise.

1740 8.4.1.4 JSON strings

1741 When converting JSON strings to BEJ format, a null terminator shall be appended to the string.

1742 8.4.1.5 JSON Boolean

1743 In JSON, Boolean data consists of one of the two sentinels “true” or “false”. These sentinels shall be
1744 encoded as BEJ Boolean data with an appropriate value field.

1745 8.4.1.6 JSON null

1746 In JSON, null data consists of the sentinel “null”. This sentinel shall be encoded as BEJ Null data only if
1747 the datatype for the property in the schema is null. For a nullable property (identified via the third tag bit

1748 from the dictionary entry or by the schema), null data shall be encoded as its standard type (from the
1749 dictionary) with length zero and no value tuple element.

1750 8.4.2 Resource links

1751 Most Redfish schemas contain links to other schemas within their properties, formatted as @odata.id
1752 annotations. When encoding these links in BEJ, the URI may be encoded as any of bejString,
1753 bejResourceLink, or bejResourceLinkExpansion. If encoded as a bejString, deferred binding substitutions
1754 may be employed as needed to complete the reference.

1755 When encoding a BEJ payload as part of an RDE create, update or action operation, the MC should use
1756 the following criteria when encoding resource links:

- 1757 1. If the resource link can be mapped to a Redfish resource PDR, the MC should encode the link
1758 using the bejResourceLink data type.
- 1759 2. If the resource link cannot be mapped to a Redfish resource PDR, and is a well-formed link, the
1760 MC should use a deferred binding bejString to encode the link.
- 1761 3. For a malformed link, the MC should reject the operation using a HTTP status code of 400 Bad
1762 Request.

1763 8.4.3 Registry items

1764 Redfish messages contain items from collated collections called registries. When encoding Redfish
1765 message registries in BEJ, the string may be encoded as either bejString or bejRegistryItem.

1766 8.4.4 Annotations

1767 Redfish annotations may be recognized as properties with a name string containing the “at” sign (@’).
1768 Several annotations are defined in Redfish, including some that are mandatory for inclusion with any
1769 Redfish GET Operation. The RDE Device is responsible for ensuring that these mandatory annotations
1770 are included in the results of an RDE read Operation.

1771 Annotations in Redfish have two forms:

- 1772 • Standalone form annotations have the form “@annotation_class.annotation_name” :
1773 annotation_value.
 - 1774 – Example: “@odata.id”: “/redfish/v1/Systems/1/”
 - 1775 – Standalone annotations shall be encoded with the BEJ data type listed in the annotation
1776 dictionary in the row matching the annotation name string
- 1777 • Property annotation form annotations have the form
1778 “property@annotation_class.annotation_name” : annotation_value.
 - 1779 – Example: “ResetType@Redfish.AllowableValues” : [“On”, “PushPowerButton”]
 - 1780 – Property annotation form annotations shall be encoded with the BEJ Property Annotation
1781 data type; the annotation value shall be encoded as a dependent child of the annotation
1782 entry. See clause 5.3.20.

1783 NOTE Unlike major schema resource properties, annotations have a flat namespace from which sequence
1784 numbers are drawn. To identify the sequence number for an annotation, an encoder should start at the root
1785 of the annotation dictionary and then find the string matching the annotation name (including the ‘@’ sign
1786 and the annotation source) within this set. In particular, the sequence number for an annotation is
1787 independent of the current encoding context.

1788 Special handling is required when the RDE Device sends a message annotation to the MC. The related
1789 properties property inside the annotation’s data structure is formatted as an array of strings, but the RDE
1790 Device has only sequence numbers to work with: the RDE Device may not be able to supply the property

1791 name for the sequence number. If the RDE Device knows the name of the related property that is
 1792 relevant for the message annotation, it may supply the name directly as an array element. Otherwise, it
 1793 shall encode into the array element a BEJ locator by concatenating the following string components:

1794

Table 43 – Message annotation related property BEJ locator encoding

Description
Delimiter Shall be ‘:’
ComponentCount The number N of sequence numbers in the fields below, stringified
Delimiter Shall be ‘:’
Locator Component [0] Sequence number [0], stringified
Delimiter Shall be ‘:’
Locator Component [1] Sequence number [1], stringified
Delimiter Shall be ‘:’
Locator Component [2] Sequence number [2], stringified
Delimiter Shall be ‘:’
...
Delimiter Shall be ‘:’
Locator Component [N – 1] Sequence number [N – 1], stringified

1795 **8.4.4.1 Nested Annotations**
 1796 The data format for an annotation may be a simple property such as a string or an integer, but it may also
 1797 be a compound property such as a set. In this latter case it is further possible that the set can itself
 1798 contain an annotation. To distinguish the case where the sequence number for annotation data refers
 1799 anew to a top-level annotation instead of to a property within the set for an annotation, the format byte of
 1800 the BEJ tuple for that annotation shall have the `read_only_property_and_top_level_annotation` bit set to
 1801 1b. When encoding nested annotations, the BEJ encoding version shall be set to 1.1.0 (0xF1F1F000).

1802 **8.4.5 Choice encoding for properties that support multiple data types**
 1803 If the encoder finds a property that is listed in the dictionary as being of type BEJ choice, it shall encode
 1804 the property with type `bejChoice` in the BEJ format tuple element. The actual value and selected data type
 1805 shall be encoded as a dependent child of the tuple containing the `bejChoice` element. See clauses 5.3.19
 1806 and 7.2.3.3.

1807 **8.4.6 Properties with invalid values**

1808 If the MC is encoding an update request from a client that includes a property value that does not match a
1809 required data type according to the dictionary it is translating from, the MC shall in accordance with the
1810 Redfish standard [DSP0266](#) respond to the client with HTTP status code 400 and a
1811 @Message.ExtendedInfo annotation specifying the property with the value format error (see
1812 PropertyValueFormatError, PropertyValueTypeError in the Redfish base message registry). Similarly, if
1813 the value supplied for a property such as an enumeration does not match any required values, the MC
1814 shall in accordance with the Redfish standard [DSP0266](#) respond to the client with HTTP status code 400
1815 and a @Message.ExtendedInfo annotation specifying the property with a value not in the accepted list
1816 (see PropertyValueNotInList in the Redfish base message registry).

1817 **8.4.7 Properties missing from dictionaries**

1818 When encoding JSON data, an encoder may find that the name of a property does not correspond to a
1819 string found in the dictionary. If the encoder is the RDE Device, this should never happen as the RDE
1820 Device is responsible for the dictionary. This situation therefore represents a non-compliant RDE
1821 implementation.

1822 If the MC finds that a property does not correspond to a string found in the dictionary from an RDE
1823 Device, it should in accordance with the Redfish standard [DSP0266](#) respond to the client with HTTP
1824 status code 200 or 400 and an annotation specifying the property as unsupported (see PropertyUnknown
1825 in the Redfish base message registry). The MC may continue to process the client request.

1826 **8.5 BEJ decoding**

1827 This clause presents implementation considerations for the BEJ decoding process.

1828 Properties in BEJ data may be encoded in any order. Decoders must therefore be prepared to accept
1829 data in whatever order it was encoded.

1830 **8.5.1 Conversion of BEJ data types to JSON**

1831 When decoding from BEJ to JSON, the following rules shall be followed. In each of the following,
1832 "property_name" shall be taken to mean the name of the property or annotation as decoded from the
1833 relevant dictionary. For all data types, if the length tuple field is zero, the data shall be decoded as
1834 follows:

1835 "property_name" : null

1836 When multiple properties appear sequentially within a set, they shall be delimited with commas.

1837 **8.5.1.1 BEJ Set**

1838 A BEJ Set shall be decoded to the following format, with the text inside angle brackets ('<', '>') replaced as
1839 indicated:

1840 "property_name" : { <set dependent children decoded individually as a comma-separated list> }

1841 **8.5.1.2 BEJ Array**

1842 A BEJ Array shall be decoded to the following format, with the text inside angle brackets ('<', '>') replaced
1843 as indicated:

1844 "property_name" : [<array dependent children decoded individually as a comma-separated list>]

1845 **8.5.1.3 BEJ Integer and BEJ Real**

1846 BEJ Integers and BEJ Reals shall be decoded to the following format, with the text inside angle brackets
1847 ('‘, ’’) replaced as indicated:

1848 “property_name” : “<decoded numeric value>”

1849 **8.5.1.4 BEJ String**

1850 BEJ Strings shall be decoded to the following format, with the text inside angle brackets ('‘, ’’) replaced
1851 as indicated. When converting BEJ strings to JSON format, the null terminator shall be dropped as JSON
1852 string encodings do not include null terminators.

1853 “property_name” : “<decoded string value>”

1854 **8.5.1.5 BEJ Boolean**

1855 BEJ Booleans shall be decoded to the following format, with the text inside angle brackets ('‘, ’’) replaced
1856 as indicated (note that the “true” and “false” sentinels are not encased in quote marks):

1857 “property_name” : <true or false, depending on the decoded value>

1858 **8.5.1.6 BEJ Null**

1859 BEJ Null shall be decoded to the following format:

1860 “property_name” : null

1861 **8.5.1.7 BEJ Resource Link**

1862 A BEJ Resource Link shall be decoded to the following format, with the text inside angle brackets ('‘, ’’) replaced as indicated.

1864 “property_name” : “<URI for the resource corresponding the Redfish Resource PDR with the
1865 supplied ResourcID>”

1866 MCs shall be aware that either a BEJ Resource Link or a BEJ Resource Link Expansion may be encoded
1867 for a dictionary entry that lists its type as BEJ Resource Link.

1868 **8.5.1.8 BEJ Resource Link expansion**

1869 A BEJ Resource Link Expansion shall be decoded to the following format, with the text inside angle
1870 brackets ('‘, ’’) replaced as indicated.

1871 <full resource data for the Redfish Resource PDR corresponding to the supplied ResourcID>

1872 NOTE property_name is not included in the decoded JSON output in this case.

1873 If the supplied ResourcID is zero and the parent resource is a collection, the MC shall use the
1874 COLLECTION_MEMBER_TYPE schema dictionary obtained from the collection resource (rather than
1875 trying to use a dictionary from the members) to decode resource data.

1876 MCs shall be aware that either a BEJ Resource Link or a BEJ Resource Link Expansion may be encoded
1877 for a dictionary entry that lists its type as BEJ Resource Link.

1878 **8.5.2 Annotations**

1879 This clause documents the approach for decoding the two types of Redfish annotations to JSON text.

1880 **8.5.2.1 Standalone annotations**

1881 Standalone annotations (data from decoded from the annotation dictionary) shall be decoded to the
1882 following format, with the bit inside angle brackets ('<', '>') replaced as indicated:

1883 "@annotation_class.annotation_name" : "<decoded annotation value>"

1884 **8.5.2.2 BEJ property annotations**

1885 BEJ Property Annotations shall be decoded to the following format, with the bit inside angle brackets ('<',
1886 '>') replaced as indicated:

1887 "property_name@annotation_class.annotation_name" : "<decoded annotation value from the
1888 annotation's dependent child node>"

1889 **8.5.2.3 [MC] Related Properties in message annotations**

1890 When a message annotation is sent from the RDE Device to the MC, the related properties field of
1891 message annotations requires special handling in RDE. Specifically, the array element string values are
1892 BEJ locators to individual properties, may be encoded as a colon-delimited string (see clause **Error!**
1893 **Reference source not found.**). When decoding, the MC shall check the first character of the supplied
1894 string. If it is a colon (:), the MC shall extract the individual sequence numbers for the BEJ locator, and
1895 then use them to identify the property name to send back to the client for the annotation. If the first
1896 character of the supplied string is not a colon, the MC shall return the supplied string unmodified.

1897 **8.5.3 Sequence numbers missing from dictionaries**

1898 It may transpire that when decoding BEJ data, a decoder finds a sequence number not in its dictionary.
1899 The handling of this case differs between the RDE Device and the MC.

1900 If the RDE Device finds an unrecognized sequence number as part of the payload for a put, patch, or
1901 create operation, the RDE Device shall in accordance with the Redfish standard [DSP0266](#) respond with
1902 an annotation specifying the sequence number as an unsupported property (see `PropertyUnknown` in the
1903 Redfish base message registry). The RDE Device may continue to decode the remainder of the payload
1904 and perform the requested Operation upon the portion it understands.

1905 If the MC finds an unrecognized sequence number as part of the response payload for a get or action
1906 Operation, or as part of a `@Message.ExtendedInfo` annotation response for any other Operation, it shall
1907 treat this as a failure on the part of the RDE Device and respond to the client with HTTP status code 500,
1908 Internal Server Error.

1909 **8.5.4 Sequence numbers for read-only properties in modification Operations**

1910 If the RDE Device is performing a modification operation (create, put, patch, or some actions), and it finds
1911 a sequence number corresponding to a property that is read-only, the RDE Device should in accordance
1912 with the Redfish standard [DSP0266](#) respond with an annotation specifying the sequence number as a
1913 non-updateable property (see `PropertyNotWritable` in the Redfish base message registry). The RDE
1914 Device may continue to decode and update with the remainder of the payload.

1915 **8.6 Example encoding and decoding**

1916 The following examples demonstrate the BEJ encoding and decoding processes. For illustrative
1917 purposes, we show the data collected in an XML form that happens to align with the schema; however,
1918 there is no requirement that data be stored in this form. Indeed, it is very unlikely that any RDE Device
1919 would do so.

1920 The examples in this clause use the example dictionary from clause 8.6.1.

1921 8.6.1 Example dictionary

1922 The example dictionary is based on the DummySimple JSON schema presented in Figure 5:

```

1923 {
1924     "$ref": "#/definitions/DummySimple",
1925     "$schema": "http://json-schema.org/draft-04/schema#",
1926     "copyright": "Copyright 2018 DMTF. For
1927         the full DMTF copyright policy, see http://www.dmtf.org/about/policies/copyright",
1928     "definitions": {
1929         "LinkStatus": {
1930             "enum": [
1931                 "NoLink",
1932                 "LinkDown",
1933                 "LinkUp"
1934             ],
1935             "type": "string"
1936         },
1937         "DummySimple" : {
1938             "additionalProperties": false,
1939             "description": "The DummySimple schema represents a very simple schema used to
1940                             demonstrate the BEJ dictionary format.",
1941             "longDescription": "This resource shall not be used except for illustrative
1942                             purposes. It does not correspond to any real hardware or software.",
1943             "patternProperties": {
1944                 "^( [a-zA-Z][a-zA-Z0-9_]* )? @odata | Redfish | Message | Privileges ) \\ . [a-zA-
1945 Z0-9_.]+ $": {
1946                     "description": "This property shall specify a valid odata or Redfish
1947                         property.",
1948                     "type": [
1949                         "array",
1950                         "boolean",
1951                         "number",
1952                         "null",
1953                         "object",
1954                         "string"
1955                     ]
1956                 }
1957             },
1958             "properties": {
1959                 "@odata.context": {
1960                     "$ref":
1961                         "http://redfish.dmtf.org/schemas/v1/odata.v4_0_1.json#/definitions/context"
1962                 },
1963                 "@odata.id": {
1964                     "$ref":
1965                         "http://redfish.dmtf.org/schemas/v1/odata.v4_0_1.json#/definitions/id"
1966                 },
1967                 "@odata.type": {
1968                     "$ref":
1969                         "http://redfish.dmtf.org/schemas/v1/odata.v4_0_1.json#/definitions/type"
1970                 },
1971                 "ChildArrayProperty": {
1972                     "items": {
1973                         "additionalProperties": false,
1974                         "type": "object",
1975                         "properties": {
1976                             "LinkStatus": {
1977                                 "anyOf": [
1978                                     {
1979                                         "$ref": "#/definitions/LinkStatus"
1980                                     },
1981                                     {
1982                                         "type": "null"
1983                                     }
1984                                 ],
1985                                 "readOnly": true
1986                             },
1987                             "AnotherBoolean": {
1988                                 "type": "boolean"
1989                             }
1990                         }
1991                     }
1992                 }
1993             }
1994         }
1995     }
1996 }
1997 
```

```

1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
    }
},
"SampleIntegerProperty": {
    "type": "integer"
},
"Id": {
    "type": "string",
    "readOnly": true
},
"SampleEnabledProperty": {
    "type": "boolean"
}
}
},
"title": "#DummySimple.v1_0_0.DummySimple"
}
}

```

Figure 5 – DummySimple schema

2009 NOTE This is not a published DMTF Redfish schema.

2010 In tabular form, the dictionary for DummySimple appears as shown in Table 44:

Table 44 – DummySimple dictionary (tabular form)

Row	Sequence Number	Format	Name	Child Pointer	Child Count
0	0	set	DummySimple	1	4
1	0	array	ChildArrayProperty	5	1
2	1	string	Id	null	0
3	2	boolean	SampleEnabledProperty	null	0
4	3	integer	SampleIntegerProperty	null	0
5	0	set	null (anonymous array elements)	6	2
6	0	boolean	AnotherBoolean	null	0
7	1	enum	LinkStatus	8	3
8	0	string	LinkDown	null	0
9	1	string	LinkUp	null	0
10	2	string	NoLink	null	0

2011 Finally, in binary form, the dictionary appears as shown in Figure 6. (Colors in this example match those used in Figure 4.)

0x00	0x00	0x0B	0x00	0x00	0xF0	0xF0	0xF1
0x12	0x01	0x00	0x00	0x00	0x00	0x00	0x16
0x00	0x04	0x00	0x0C	0x7A	0x00	0x14	0x00
0x00	0x3E	0x00	0x01	0x00	0x13	0x86	0x00
0x56	0x01	0x00	0x00	0x00	0x00	0x00	0x03
0x99	0x00	0x74	0x02	0x00	0x00	0x00	0x00

2021	0x00 0x16 0x9C 0x00 0x34 0x03 0x00 0x00 0x00
2022	0x00 0x00 0x00 0x16 0xB2 0x00 0x00 0x00 0x00
2023	0x00 0x48 0x00 0x02 0x00 0x00 0x00 0x00 0x00
2024	0x74 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x0F
2025	0xC8 0x00 0x46 0x01 0x00 0x5C 0x00 0x03 0x00
2026	0x00 0x0B 0xD7 0x00 0x50 0x00 0x00 0x00 0x00
2027	0x00 0x00 0x00 0x09 0xE2 0x00 0x50 0x01 0x00
2028	0x00 0x00 0x00 0x00 0x00 0x07 0xEB 0x00 0x00
2029	0x50 0x02 0x00 0x00 0x00 0x00 0x00 0x00 0x07
2030	0xF2 0x00 0x44 0x75 0x6D 0x6D 0x79 0x53 0x00
2031	0x69 0x6D 0x70 0x6C 0x65 0x00 0x43 0x68 0x00
2032	0x69 0x6C 0x64 0x41 0x72 0x72 0x61 0x79 0x00
2033	0x50 0x72 0x6F 0x70 0x65 0x72 0x74 0x79 0x00
2034	0x00 0x49 0x64 0x00 0x53 0x61 0x6D 0x70 0x00
2035	0x6C 0x65 0x45 0x6E 0x61 0x62 0x6C 0x65 0x00
2036	0x64 0x50 0x72 0x6F 0x70 0x65 0x72 0x74 0x00
2037	0x79 0x00 0x53 0x61 0x6D 0x70 0x6C 0x65 0x00
2038	0x49 0x6E 0x74 0x65 0x67 0x65 0x72 0x50 0x00
2039	0x72 0x6F 0x70 0x65 0x72 0x74 0x79 0x00 0x00
2040	0x41 0x6E 0x6F 0x74 0x68 0x65 0x72 0x42 0x00
2041	0x6F 0x6F 0x6C 0x65 0x61 0x6E 0x00 0x4C 0x00
2042	0x69 0x6E 0x6B 0x53 0x74 0x61 0x74 0x75 0x00
2043	0x73 0x00 0x4C 0x69 0x6E 0x6B 0x44 0x6F 0x00
2044	0x77 0x6E 0x00 0x4C 0x69 0x6E 0x6B 0x55 0x00
2045	0x70 0x00 0x4E 0x6F 0x4C 0x69 0x6E 0x6B 0x00
2046	0x00 0x18 0x43 0x6F 0x70 0x79 0x72 0x69 0x00
2047	0x67 0x68 0x74 0x20 0x28 0x63 0x29 0x20 0x00
2048	0x32 0x30 0x31 0x38 0x20 0x44 0x4D 0x54 0x00
2049	0x46 0x00 0x00 0x00 0x00 0x00 0x00 0x00 0x00

Figure 6 – DummySimple dictionary – binary form

8.6.2 Example encoding

For this example, we start with the following data (shown here in an XML representation).

NOTE The names assigned to array elements are fictitious and inserted for illustrative purposes only. Also, the encoding sequence presented here is only one possible approach; any sequence that generates the same result is acceptable. The value of the @odata.id annotation shown here is a deferred binding (see clause 8.3) that assumes the DummySimple resource corresponds to a Redfish Resource PDR with resource ID 10. Finally, for illustrative purposes we omit here the header bytes contained within the bejEncoding type that are not part of the bejTuple PLDM type.

```

2060 <Item name="DummySimple" type="set">
2061   <Item name="@odata.id" type="string" value="%L10">
2062     <Item name="ChildArrayProperty" type="array">
2063       <Item name="array element 0">
2064         <Item name="AnotherBoolean" type="boolean" value="true"/>
2065         <Item name="LinkStatus" type="enum" enumtype="String">
2066           <Enumeration value="NoLink"/>
2067         </Item>
2068       </Item>
2069       <Item name="array element 1">
2070         <Item name="LinkStatus" type="enum" enumtype="String">
2071           <Enumeration value="LinkDown"/>
2072         </Item>
2073       </Item>
2074     </Item>
2075     <Item name="Id" type="string" value="Dummy ID"/>
2076     <Item name="SampleIntegerProperty" type="number" value="12"/>
2077   </Item>
```

2078 The first step of the encoding process is to insert sequence numbers, which can be retrieved from the
 2079 relevant dictionary. (For purposes of this example, we are assuming that the @odata.id annotation is
 2080 sequence number 16 in the annotation dictionary.) Sequence numbers for array elements correspond to
 2081 their zero-based index within the array.

```

2082 <Item name="DummySimple" type="set" seqno="major/0">
2083   <Item name="@odata.id" type="string" value="%L10" seqno="annotation/16">
2084     <Item name="ChildArrayProperty" type="array" seqno="major/0">
2085       <Item name="array element 0" seqno="major/0">
2086         <Item name="AnotherBoolean" type="boolean" value="true" seqno="major/0"/>
2087           <Item name="LinkStatus" type="enum" enumtype="String" seqno="major/1">
2088             <Enumeration value="NoLink" seqno="major/2"/>
2089           </Item>
2090         </Item>
2091       <Item name="array element 1" seqno="major/1">
2092         <Item name="LinkStatus" type="enum" enumtype="String" seqno="major/1">
2093           <Enumeration value="LinkDown" seqno="major/0"/>
2094         </Item>
2095       </Item>
2096     </Item>
2097     <Item name="Id" type="string" value="Dummy ID" seqno="major/1"/>
2098     <Item name="SampleIntegerProperty" type="integer" value="12" seqno="major/3"/>
2099   </Item>
```

2100 After the sequence numbers are fully characterized, they can be encoded. We encode which dictionary
 2101 these sequence numbers came by shifting them left one bit to insert 0b (major dictionary) or 1b
 2102 (annotation dictionary) as the low order bit per clause 8.2.1. As the sequence numbers are now assigned,
 2103 names of properties and enumeration values are no longer needed:

```

2104
2105 <Item type="set" seqno="0">
2106   <Item seqno="33" type="string" value="%L10" seqno="annotation/16">
2107     <Item type="array" seqno="0">
2108       <Item seqno="0">
2109         <Item type="boolean" value="true" seqno="0"/>
2110           <Item type="enum" enumtype="String" seqno="2">
2111             <Enumeration seqno="4"/>
2112           </Item>
2113         </Item>
2114       <Item seqno="2">
2115         <Item type="enum" enumtype="String" seqno="2">
2116           <Enumeration seqno="0"/>
2117         </Item>
2118       </Item>
2119     </Item>
2120     <Item type="string" value="Dummy ID" seqno="2"/>
2121     <Item type="integer" value="12" seqno="6"/>
2122   </Item>
```

2123 The next step is to convert everything into BEJ SFLV Tuples. Per clause 5.3.12, the value of an
 2124 enumeration is the sequence number for the selected option.
 2125

```

2126 {0x01 0x00, set, [length placeholder], value={count=3,
2127     {0x01 0x21, string, [length placeholder], value="%L10"}
2128     {0x01 0x00, array, [length placeholder], value={count=2,
2129         {0x01 0x00, set, [length placeholder], value={count=2,
2130             {0x01 0x00, boolean, [length placeholder], value=true}
2131             {0x01 0x02, enum, [length placeholder], value=2}
2132         }
2133         {0x01 0x02, set, [length placeholder], value={count=1,
2134             {0x01 0x02, enum, [length placeholder], value=0}
2135         }
2136     }
2137     {0x01 0x02, string, [length placeholder], value="Dummy ID"}
2138     {0x01 0x06, integer, [length placeholder], value=12}
2139   }
}

```

2140 We now encode the formats and the leaf nodes, following **Table 9**. For sets and arrays, the value
 2141 encoding count prefix is a nonnegative Integer; we can encode that now as well per **Table 4**. Note the null
 2142 terminator for the string. The encoded sequence numbers for enumeration values do not need a
 2143 dictionary selector inserted as the LSB as the dictionary was already indicated with the sequence number
 2144 for the enumeration itself in the format tuple field. The @odata.id annotation string value contains a
 2145 deferred binding, so we set that bit in the format tuple field.

```

2146
2147 {0x01 0x00, 0x00, [length placeholder], {0x01 0x04,
2148     {0x01 0x21, 0x51, [length placeholder], 0x25 0x4C 0x31 0x30}
2149     {0x01 0x00, 0x10, [length placeholder], {0x01 0x02,
2150         {0x01 0x00, 0x00, [length placeholder], {0x01 0x02,
2151             {0x01 0x00, 0x70, [length placeholder], 0xFF}
2152             {0x01 0x02, 0x40, [length placeholder], 0x01 0x02}
2153         }
2154         {0x01 0x02, 0x00, [length placeholder], {0x01 0x01,
2155             {0x01 0x02, 0x40, [length placeholder], 0x01 0x00}
2156         }
2157     }
2158     {0x01 0x02, 0x50, [length placeholder],
2159         0x44 0x75 0x6D 0x6D 0x79 0x20 0x49 0x44 0x00}
2160     {0x01 0x06, 0x30, [length placeholder], 0x0C}
2161   }
}

```

2162 All that remains is to fill in the length values. We begin at the leaves:

```

2163
2164 {0x01 0x00, 0x00, [length placeholder], {0x01 0x04,
2165     {0x01 0x21, 0x51, 0x01 0x04, 0x25 0x4C 0x31 0x30}
2166     {0x01 0x00, 0x10, [length placeholder], {0x01 0x02,
2167         {0x01 0x00, 0x00, [length placeholder], {0x01 0x02,
2168             {0x01 0x00, 0x70, 0x01 0x01, 0xFF}
2169             {0x01 0x02, 0x40, 0x01 0x02, 0x01 0x02}
2170         }
2171         {0x01 0x02, 0x00, [length placeholder], {0x01 0x01,
2172             {0x01 0x02, 0x40, 0x01 0x02, 0x01 0x00}
2173         }
2174     }
2175     {0x01 0x02, 0x50, 0x01 0x09,
2176         0x44 0x75 0x6D 0x6D 0x79 0x20 0x49 0x44 0x00}
2177     {0x01 0x06, 0x30, 0x01 0x01, 0x0C}
2178   }
}

```

2179 We then work our way from the leaves towards the outermost enclosing tuples. First, the array element
 2180 sets:
 2181

```

2182 {0x01 0x00, 0x00, [length placeholder], {0x01 0x04,
2183   {0x01 0x21, 0x51, 0x01 0x04, 0x25 0x4C 0x31 0x30}
2184   {0x01 0x00, 0x10, [length placeholder], {0x01 0x02,
2185     {0x00, 0x00, 0x01 0x0F, {0x01 0x02,
2186       {0x01 0x00, 0x07, 0x01 0x01, 0xFF}
2187       {0x01 0x20, 0x04, 0x01 0x02, 0x01 0x02}
2188     }
2189     {0x01 0x02, 0x00, 0x01 0x09, {0x01 0x01,
2190       {0x01 0x02, 0x40, 0x01 0x02, 0x01 0x00}
2191     }
2192   }
2193   {0x01 0x02, 0x50, 0x01 0x09,
2194     0x44 0x75 0x6D 0x6D 0x79 0x20 0x49 0x44 0x00}
2195   {0x01 0x06, 0x30, 0x01 0x01, 0x0C}
2196 }

```

2197 Next, the array itself:

```

2198
2199 {0x01 0x00, 0x00, [length placeholder], {0x01 0x04,
2200   {0x01 0x21, 0x51, 0x01 0x04, 0x25 0x4C 0x31 0x30}
2201   {0x01 0x00, 0x10, 0x01 0x24, {0x01 0x02,
2202     {0x01 0x00, 0x00, 0x01 0x0F, {0x01 0x02,
2203       {0x01 0x00, 0x70, 0x01 0x01, 0xFF}
2204       {0x01 0x02, 0x40, 0x01 0x02, 0x01 0x02}
2205     }
2206     {0x01 0x02, 0x00, 0x01 0x09, {0x01 0x01,
2207       {0x01 0x02, 0x40, 0x01 0x02, 0x01 0x00}
2208     }
2209   }
2210   {0x01 0x02, 0x50, 0x01 0x09,
2211     0x44 0x75 0x6D 0x6D 0x79 0x20 0x49 0x44 0x00}
2212   {0x01 0x06, 0x30, 0x01 0x01, 0x0C}
2213 }

```

2214 Finally, the outermost set:

```

2215
2216 {0x01 0x00, 0x00, 0x01 0x48, {0x01 0x04,
2217   {0x01 0x21, 0x51, 0x01 0x04, 0x25 0x4C 0x31 0x30}
2218   {0x01 0x00, 0x10, 0x01 0x24, {0x01 0x02,
2219     {0x01 0x00, 0x00, 0x01 0x0F, {0x01 0x02,
2220       {0x01 0x00, 0x70, 0x01 0x01, 0xFF}
2221       {0x01 0x02, 0x40, 0x01 0x02, 0x01 0x02}
2222     }
2223     {0x01 0x02, 0x00, 0x01 0x09, {0x01 0x01,
2224       {0x01 0x02, 0x40, 0x01 0x02, 0x01 0x00}
2225     }
2226   }
2227   {0x01 0x02, 0x50, 0x01 0x09,
2228     0x44 0x75 0x6D 0x6D 0x79 0x20 0x49 0x44 0x00}
2229   {0x01 0x06, 0x30, 0x01 0x01, 0x0C}
2230 }

```

2231 The encoded bytes may now be read off, and the inner encoding is complete:
2232

```

2233 0x01 0x00 0x00 0x01 : 0x48 0x01 0x04 0x01
2234 0x21 0x51 0x01 0x04 : 0x25 0x4C 0x31 0x30
2235 0x01 0x00 0x10 0x01 : 0x24 0x01 0x02 0x01
2236 0x00 0x00 0x01 0x0F : 0x01 0x02 0x01 0x00
2237 0x70 0x01 0x01 0xFF : 0x01 0x02 0x40 0x01
2238 0x02 0x01 0x02 0x01 : 0x02 0x00 0x01 0x09
2239 0x01 0x01 0x01 0x02 : 0x40 0x01 0x02 0x01
2240 0x00 0x01 0x02 0x50 : 0x01 0x09 0x44 0x75
2241 0x6D 0x6D 0x79 0x20 : 0x49 0x44 0x00 0x01
2242 0x06 0x30 0x01 0x01 : 0x0C

```

2243 8.6.3 Example decoding

2244 The decoding process is largely the inverse of the encoding process. For this example, we start with the
2245 final encoded data from clause 8.6.1:

```

2246
2247 0x01 0x00 0x00 0x01 : 0x48 0x01 0x04 0x01
2248 0x21 0x51 0x01 0x04 : 0x25 0x4C 0x31 0x30
2249 0x01 0x00 0x10 0x01 : 0x24 0x01 0x02 0x01
2250 0x00 0x00 0x01 0x0F : 0x01 0x02 0x01 0x00
2251 0x70 0x01 0x01 0xFF : 0x01 0x02 0x40 0x01
2252 0x02 0x01 0x02 0x01 : 0x02 0x00 0x01 0x09
2253 0x01 0x01 0x01 0x02 : 0x40 0x01 0x02 0x01
2254 0x00 0x01 0x02 0x50 : 0x01 0x09 0x44 0x75
2255 0x6D 0x6D 0x79 0x20 : 0x49 0x44 0x00 0x01
2256 0x06 0x30 0x01 0x01 : 0x0C

```

2257 The first step of the decoding process is to map the byte data to {SFLV} tuples, using the length bytes and
2258 set/array counts to identify tuple boundaries:

```

2259
2260 {S=0x01 0x00, F=0x00, L=0x01 0x3F, V={0x01 0x04,
2261     {S=0x01 0x21, F=0x51, L=0x01 0x04, V=0x25 0x4C 0x31 0x30}
2262     {S=0x01 0x00, F=0x10, L=0x01 0x24, V={0x01 0x02,
2263         {S=0x01 0x00, F=0x00, L=0x01 0x0F, V={0x01 0x02,
2264             {S=0x01 0x00, F=0x70, L=0x01 0x01, V=0xFF}
2265             {S=0x01 0x02, F=0x40, L=0x01 0x02, V=0x01 0x02}
2266         }
2267         {S=0x01 0x02, F=0x00, L=0x01 0x09, V={0x01 0x01,
2268             {S=0x01 0x02, F=0x40, L=0x01 0x02, V=0x01 0x00}
2269         }
2270     }
2271     {S=0x01 0x02, F=0x50, L=0x01 0x09,
2272         V=0x44 0x75 0x6D 0x6D 0x79 0x20 0x49 0x44 0x00}
2273     {0x01 S=0x06, F=0x30, L=0x01 0x01, V=0x0C}
2274 }

```

2275 After the tuple boundaries are understood, the length and count data are no longer needed:

```

2276
2277 {S=0x01 0x00, F=0x00, V={
2278     {S=0x01 0x21, F=0x51, V=0x25 0x4C 0x31 0x30}
2279     {S=0x01 0x00, F=0x10, V={
2280         {S=0x01 0x00, F=0x00, V={
2281             {S=0x01 0x00, F=0x70, V=0xFF}
2282             {S=0x01 0x02, F=0x40, V=0x01 0x02}
2283         }
2284         {S=0x01 0x02, F=0x00, V={
2285             {S=0x01 0x02, F=0x40, V=0x01 0x00}
2286         }
2287     }
2288     {S=0x01 0x02, F=0x50, V=0x44 0x75 0x6D 0x6D 0x79 0x20 0x49 0x44 0x00}

```

```
2289     {S=0x01 0x06, F=0x30, V=0x0C}
2290 }
```

2291 The next step is to decode format tuple bytes using **Table 9**. This will tell us how to decode the value
2292 data:

```
2293
2294 {S=0x01 0x00, set, V={}
2295   {S=0x01 0x21, string with deferred binding, V=0x25 0x4C 0x31 0x30}
2296   {S=0x01 0x00, array, V={}
2297     {S=0x01 0x00, set, V={}
2298       {S=0x01 0x00, boolean, V=0xFF}
2299       {S=0x01 0x02, enum, V=0x01 0x02}
2300     }
2301     {S=0x01 0x02, set, V={}
2302       {S=0x01 0x02, enum, V=0x01 0x00}
2303     }
2304   }
2305   {S=0x01 0x02, string, V=0x44 0x75 0x6D 0x6D 0x79 0x20 0x49 0x44 0x00}
2306   {S=0x01 0x06, integer, V=0x0C}
2307 }
```

2308 We now decode value data. The deferred binding for the @odata.id property can now be processed,
2309 translating from "%L10" to "/redfish/v1/systems/1/DummySimples/1", an instance in a collection of
2310 resources of type DummySimple:

```
2311
2312 {S=0x01 0x00, set, {
2313   {S=0x01 0x21, string, "/redfish/v1/systems/1/DummySimples/1"}
2314   {S=0x01 0x00, array, {
2315     {S=0x01 0x00, set, {
2316       {S=0x01 0x00, boolean, true}
2317       {S=0x01 0x02, enum, <value 2>}
2318     }
2319     {S=0x01 0x02, set, {
2320       {S=0x01 0x02, enum, <value 0>}
2321     }
2322   }
2323   {S=0x01 0x02, string, "Dummy ID"}
2324   {S=0x01 0x06, integer, 12}
2325 }
```

2326 Next, we decode the sequence numbers to identify which dictionary they select:

```
2327
2328 {S=major/0, set, {
2329   {S=annotation/16, string, "/redfish/v1/systems/1/DummySimples/1"}
2330   {S=major/0, array, {
2331     {S=major/0, set, {
2332       {S=major/0, boolean, true}
2333       {S=major/1, enum, <value 2>}
2334     }
2335     {S=major/1, set, {
2336       {S=major/1, enum, <value 0>}
2337     }
2338   }
2339   {S=major/1, string, "Dummy ID"}
2340   {S=major/3, integer, 12}
2341 }
```

2342 Next, we use the selected dictionary to replace decoded sequence numbers with the strings they
2343 represent:

```

2345 {"DummySimple", set, {
2346     "@odata.id", string, "/redfish/v1/systems/1/DummySimples/1"
2347     "ChildArrayProperty", array, {
2348         {<Array element 0>, set, {
2349             "AnotherBoolean", boolean, true
2350             {"LinkStatus", enum, "NoLink"}
2351         }}
2352         {<Array element 1>, set, {
2353             "LinkStatus", enum, "LinkDown"
2354         }}
2355     }}
2356     {"Id", string, "Dummy ID"}
2357     {"SampleIntegerProperty", integer, 12}
2358 }
}

```

2359 We can now write out the decoded BEJ data in JSON format if desired (an MC will need to do this to
 2360 forward an RDE Device's response to a client, but an RDE Device may not need this step):

```

2361 {
2362     "DummySimple" : {
2363         "@odata.id" : "/redfish/v1/systems/1/DummySimples/1",
2364         "ChildArrayProperty" : [
2365             {
2366                 "AnotherBoolean" : true,
2367                 "LinkStatus" : "NoLink"
2368             },
2369             {
2370                 "LinkStatus" : "LinkDown"
2371             }
2372         ],
2373         "Id" : "Dummy ID",
2374         "SampleIntegerProperty" : 12
2375     }
2376 }
2377

```

2378 8.7 BEJ locators

2379 A BEJ locator represents a particular location within a resource at which some operation is to take place.
 2380 The locator itself consists of a list of sequence numbers for the series of nodes representing the traversal
 2381 from the root of the schema tree down to the point of interest. The list of schema nodes is concatenated
 2382 together to form the locator. A locator with no sequence numbers targets the root of the schema.

2383 NOTE The sequence numbers are absolute as they are relative to the schema, not to the subset of the schema for
 2384 which the RDE Device supports data. This enables a locator to be unambiguous.

2385 As an example, consider a locator, encoded for the example dictionary of clause 8.6.1:

2386 0x01 0x08 0x01 0x00 0x01 0x00 0x01 0x06 0x01 0x02

2387 Decoding this locator, begins with decoding the length in bytes of the locator. In this case, the first two
 2388 bytes specify that the remainder of the locator is 8 bytes long. The next step is to decode the bejTupleS-
 2389 formatted sequence numbers. The low-order bit of each sequence number references the schema to
 2390 which it refers; in this case, the pattern 0b indicates the major schema. Decoding produces the following
 2391 list:

2392 0, 0, 3, 1

2393 Now, referring to the dictionary enables identification of the target location. Remember that all indices are
 2394 zero-based:

- 2395 • The first zero points to DummySimple

- 2396 • The second zero points to the first child of DummySimple, or ChildArrayProperty
2397 • The three points to the fourth element in the ChildArrayProperty array, an anonymous instance
2398 of the array type (array instances are not reflected in the dictionary, but are implicitly the
2399 immediate children of any array)
2400 • The one points to the second child inside the ChildArray element type, or LinkStatus

2401 **9 Operational behaviors**

2402 This clause describes the operational behavior for initialization, Operations/Tasks, and Events.

2403 **9.1 Initialization (MC perspective)**

2404 The following clauses present initialization of RDE Devices with MCs.

2405 **9.1.1 Sample initialization ladder diagram**

2406 Figure 7 presents the ladder diagram for an example initialization sequence.

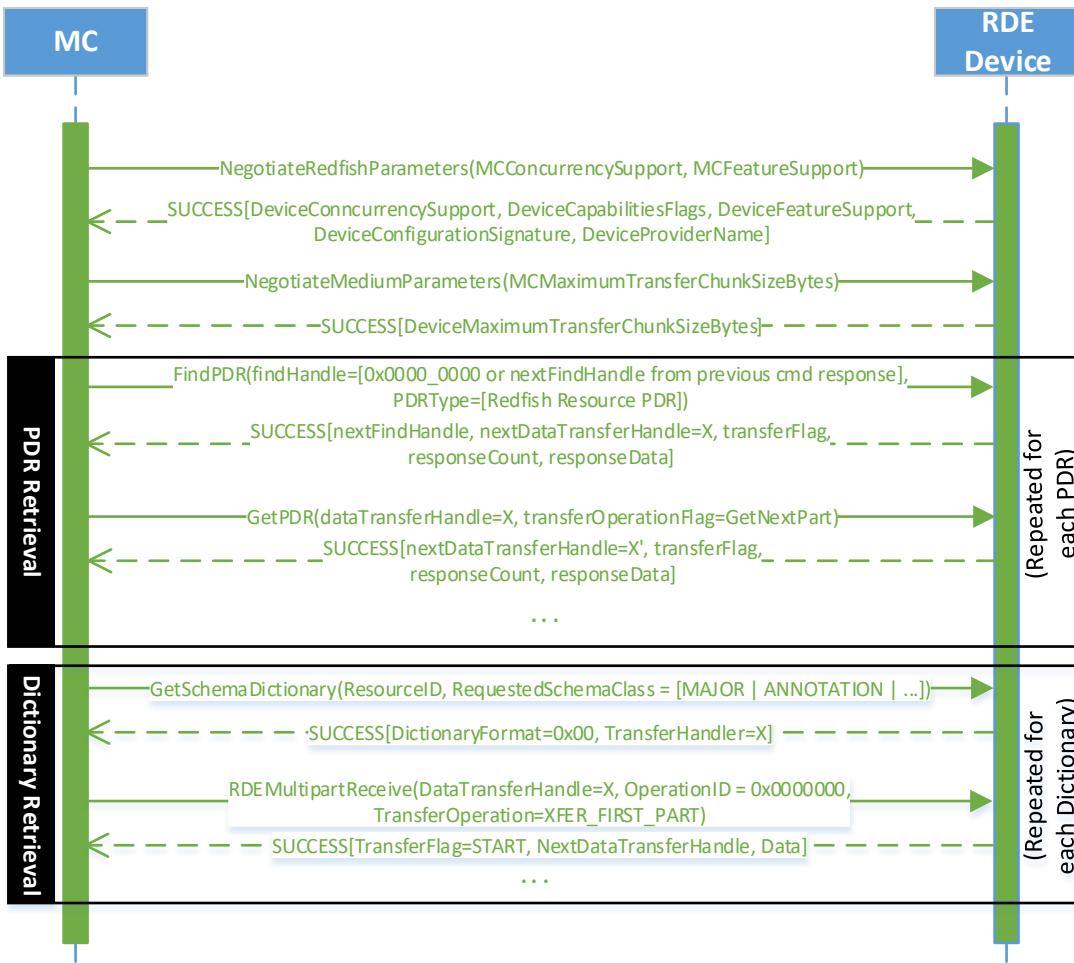
2407 Once the MC detects the RDE Device, it begins the discovery process by invoking the
2408 NegotiateRedfishParameters command to determine the concurrency and feature support for the RDE
2409 Device. It then uses the NegotiateMediumParameters command to determine the maximum message
2410 size that the MC and the RDE Device can both support. This finishes the RDE discovery process.

2411 After discovery comes the RDE registration process. It consists of two parts, PDR retrieval and dictionary
2412 retrieval. To retrieve the RDE PDRs, the MC utilizes the PLDM for Platform Monitoring and Control
2413 FindPDR command to locate PDRs that are specific to RDE⁴. For each such PDR located, the MC then
2414 retrieves it via one or more message sequences in the PLDM for Platform Monitoring and Control
2415 GetPDR command.

2416 After all the PDRs are retrieved, the next step is to retrieve dictionaries. For each Redfish Resource PDR
2417 that the MC retrieved, it retrieves the relevant dictionaries via a standardized process in which it first
2418 executes the GetSchemaDictionary command to obtain a transfer handle for the dictionary. It then uses
2419 the transfer handle with the RDEMultipartReceive command to retrieve the corresponding dictionary.

2420 Multiple initialization variants are possible; for example, it is conceivable that retrieval of some or all
2421 dictionaries could be postponed until such time as the MC needs to translate BEJ and/or JSON code for
2422 the relevant schema. Further, the MC may be able to determine that one or more of the dictionaries it has
2423 already retrieved is adequate to support a PDR and thus skip retrieving that dictionary anew. Finally, if the
2424 DeviceConfigurationSignature from the NegotiateRedfishParameters command matches the one for data
2425 that the MC has already cached for the RDE Device, it may skip the retrieval altogether.

⁴ Note: FindPDR is an optional command. If the RDE Device does not support it, the MC may achieve equivalent functionality by using GetPDR to transfer of each PDR one at a time, discarding any that are not RDE PDRs.



2426

2427

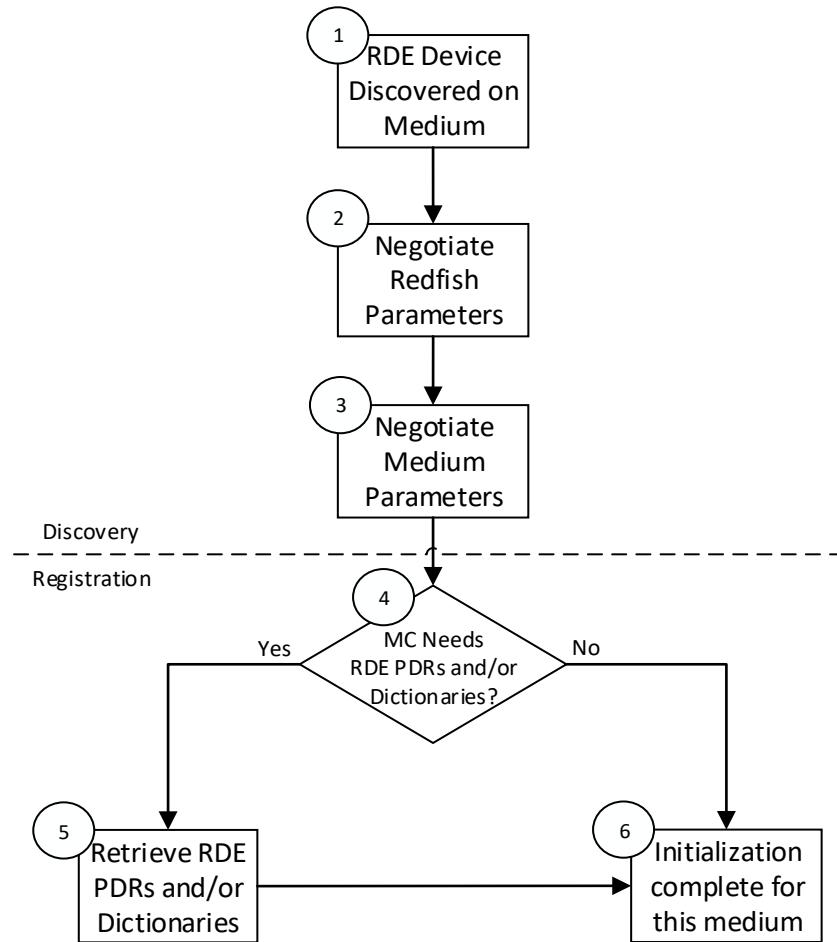
Figure 7 – Example Initialization ladder diagram2428 **9.1.2 Initialization workflow diagram**

2429 Table 45 details the information presented visually in Figure 8.

2430

Table 45 – Initialization Workflow

Step	Description	Condition	Next Step
1 – DISCOVERY	The MC discovers the presence of the RDE Device through either a medium-specific or other out-of-band mechanism	None	2
2 – NEG_REDISH	The MC issues the NegotiateRedfishParameters command to the device in order to learn basic information about it	Successful command completion	3
3 – NEG_MEDIUM	The MC issues the NegotiateMediumParameters command to the RDE Device to learn how the RDE Device intends to behave with this medium	Successful command completion	4
4 –NEED_PDR / DICTIONARY_CHECK	The MC may already have dictionaries and PDRs for the RDE Device cached, such as if this is not the first medium the RDE Device has been discovered on. The MC may choose not to retrieve a fresh copy if the DeviceConfigurationSignature from the NegotiateRedfishParameters command's response message matches what was previously received.	MC does not need to retrieve PDRs or dictionaries for this RDE Device	6
		Otherwise	5
5 – RETRIEVE_PDR / DICTIONARY	The MC retrieves PDRs and/or dictionaries from the RDE Device	Retrieval complete	6
6 – INIT_COMPLETE	The MC has finished discovery and registration for this device	None	None



2432 **Figure 8 – Typical RDE Device discovery and registration**

2433 **9.2 Operation/Task lifecycle**

2434 The following clauses present the Task lifecycle from two perspectives, first from an Operation-centric
 2435 viewpoint and then from the RDE Device perspective. MC and RDE Device implementations of RDE shall
 2436 comply with the sequences presented here.

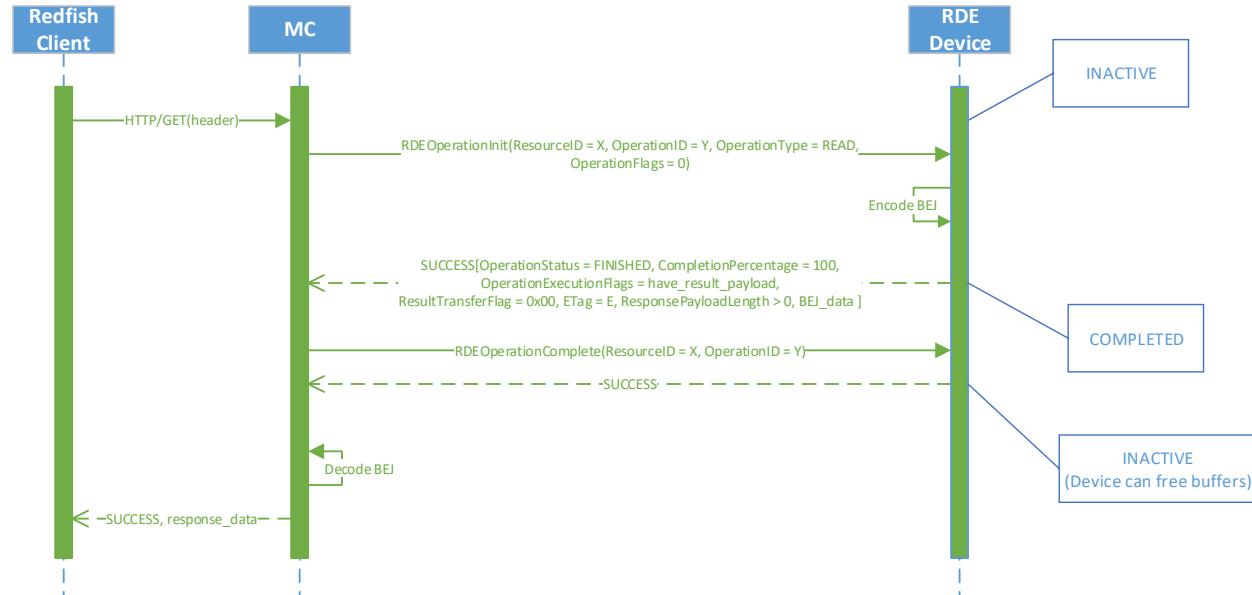
2437 **9.2.1 Example Operation command sequence diagrams**

2438 This clause presents request/response messaging sequences for common Operations.

2439 **9.2.1.1 Simple read Operation ladder diagram**

2440 Figure 9 presents the ladder diagram for a simple read Operation. The Operation begins when the
 2441 Redfish client sends a GET request over an HTTP connection to the MC. The MC decodes the URI
 2442 targeted by the GET operation to pin it down to a specific resource and PDR and sends the
 2443 RDEOperationInit command to the RDE Device that owns the PDR, with OperationType set to READ.
 2444 The RDE Device now has everything it needs for the Operation, so it performs a BEJ encoding of the
 2445 schema data for the requested resource and sends it as an inline payload back to the MC. Sending inline
 2446 is possible in this case because the read data is small enough to not cause the response message to
 2447 exceed the maximum transfer size that was previously negotiated in the NegotiateMediumParameters
 2448 command. The MC in turn has all of the results for the Operation, so it sends RDEOperationComplete to
 2449 finalize the Operation. The RDE Device can now throw away the BEJ encoded read result, so it does so

2450 and responds to the MC with success. Finally, the MC uses the dictionary it previously retrieved from the
 2451 RDE Device to decode the BEJ payload for the read command into JSON data and the MC sends the
 2452 JSON data back to the client.



2453
2454

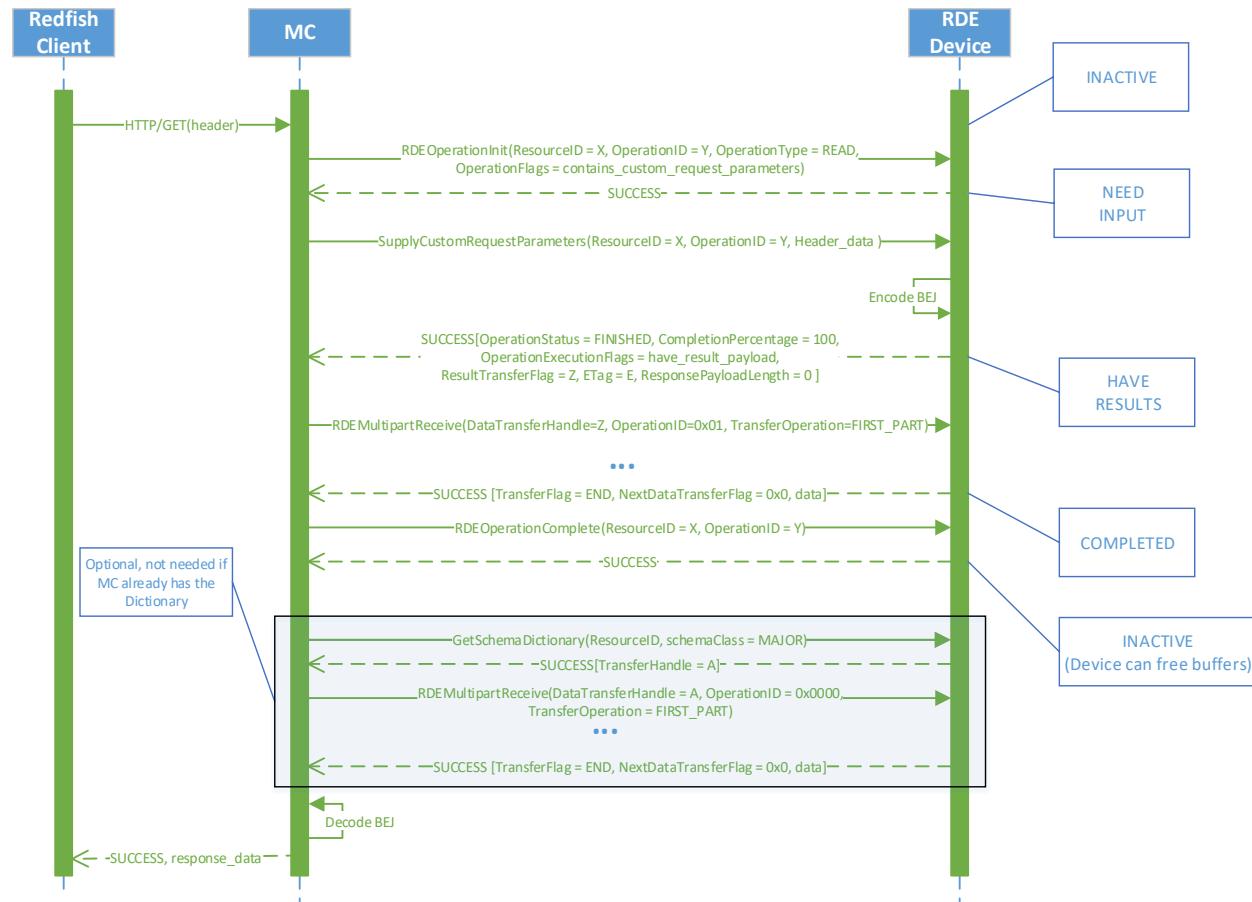
2455 **Figure 9 – Simple read Operation ladder diagram**

2456 **9.2.1.2 Complex read Operation diagram**

2457 Figure 10 presents the ladder diagram for a more complex read Operation. As with the simple read case,
 2458 the Operation begins when the Redfish client sends a GET request over an HTTP connection to the MC.
 2459 The MC again decodes the URI targeted by the GET operation to pin it down to a specific resource and
 2460 PDR and sends the RDEOperationInit command to the RDE Device that owns the PDR, with
 2461 OperationType set to READ. In this case, however, the OperationFlags that the MC sent with the
 2462 RDEOperationInit command indicate that there are supplemental parameters to be sent to the RDE
 2463 Device, so the RDE Device must wait for these before beginning work on the Operation. The MC sends
 2464 these supplemental parameters to the RDE Device via the SupplyCustomRequestParameters command.

2465 At this point, the RDE Device has everything it needs for the Operation, so just as before, the RDE
 2466 Device performs a BEJ encoding of the schema data for the requested resource. As opposed to the
 2467 previous example, in this case the BEJ-encoded payload is too large to fit within the response message,
 2468 so the RDE Device instead supplied a transfer handle that the MC can use to retrieve the BEJ payload
 2469 separately. The MC, seeing this, performs a series of RDEMultipartReceive commands to retrieve the
 2470 payload. Once it is all transferred, the MC has everything it needs. Whether it needed to retrieve a
 2471 dictionary or it already had one, the MC now sends the RDEOperationComplete command to finalize the
 2472 Operation and allow the RDE Device to throw away the BEJ encoded read result. If the MC needs a
 2473 dictionary to decode the BEJ payload, it may retrieve one via the GetSchemaDictionary command
 2474 followed by one or more RDEMultipartReceive commands to retrieve the binary dictionary data.
 2475 (Normally, the MC would have retrieved the dictionary during initialization; however, if the MC has limited
 2476 storage space to cache dictionaries, it may have been forced to evict it.) Finally, the MC uses the
 2477 dictionary to decode the BEJ payload for the read command into JSON data and then the MC sends the
 2478 JSON data back to the client.
 2479

2480

2481
2482

2483

Figure 10 – Complex Read Operation ladder diagram2484 **9.2.1.3 Write (update) Operation ladder diagram**

2485 Figure 11 presents the ladder diagram for a write operation. As with the read cases, the operation begins
 2486 when the Redfish client sends a request over an HTTP connection to the MC, in this case, an UPDATE.
 2487 Once again, the MC decodes the URI targeted by the UPDATE operation to pin it down to a specific
 2488 resource and PDR. Before it can send the RDEOperationInit command to the RDE Device that owns the
 2489 PDR, the MC must perform a BEJ encoding of the JSON payload it received from the Redfish client. If the
 2490 BEJ encoded payload were small enough to fit within the maximum transfer chunk, the MC could inline it
 2491 with the RDEOperationInit command; however, in this example, that is not the case. The MC therefore
 2492 sends RDEOperationInit with the OperationType set to UPDATE and a nonzero transfer handle. Seeing
 2493 this, the RDE Device knows to expect a larger payload via RDEMultipartSend.

2494 The MC uses the RDEMultipartSend command to transfer the encoded payload to the RDE Device in one
 2495 or more chunks. The contains_request_parameters Operation flag is not set, so the RDE Device will not
 2496 expect supplemental parameters as part of this operation. Having everything it needs to execute, the
 2497 RDE Device moves to the TRIGGERED state. The MC now sends the RDEOperationStatus command to
 2498 the RDE Device to have it execute the operation. (In practice, the RDE Device is allowed to begin
 2499 executing the operation as soon as it has received the request payload, so it may choose not to wait for
 2500 the RDEOperationStatus command to do so.) The RDE Device executes the operation and sends the

2501 results to the MC as the response to the RDEOperationStatus command. As before, the MC finalizes the
 2502 Operation via RDEOperationComplete and then sends the results back to the client.

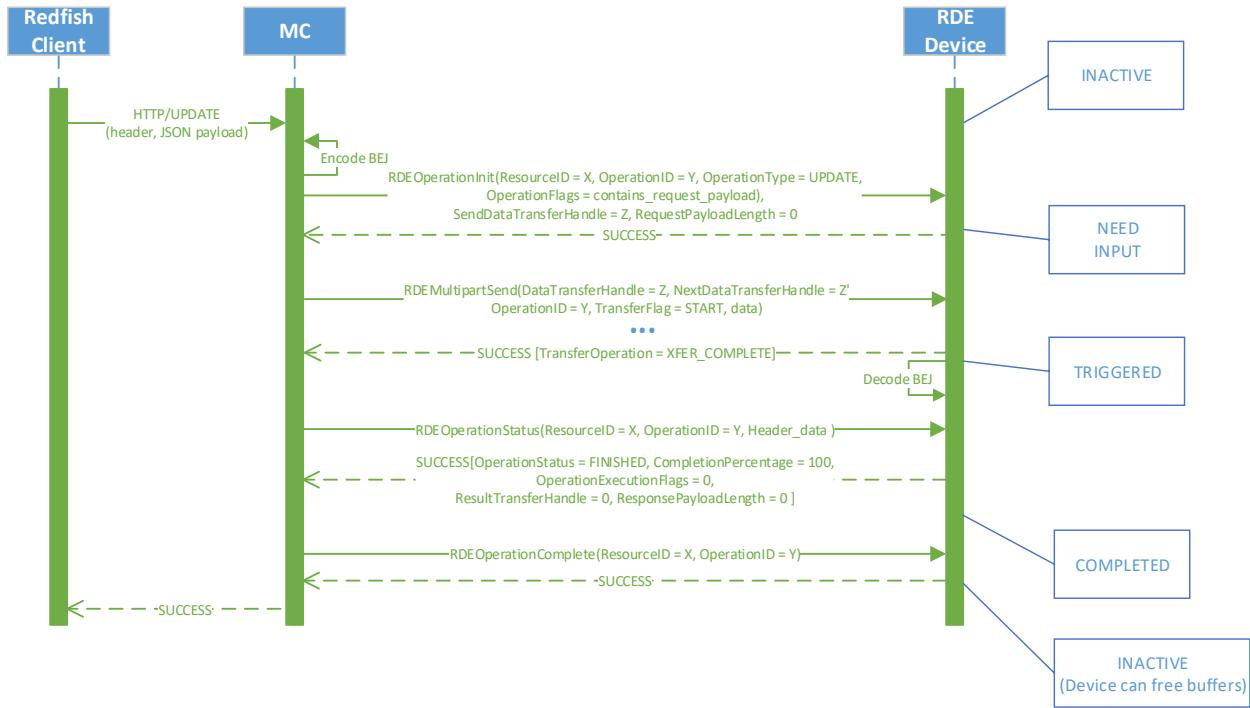


Figure 11 – Write Operation ladder diagram

9.2.1.4 Write (update) with Long-running Task Operation Ladder Diagram

2503
 2504

2508 Figure 12 presents the ladder diagram for a write Operation that spawns a long-running Task. As with the
2509 previous case, the Operation begins when the Redfish client sends an UPDATE request over an HTTP
2510 connection to the MC, and the MC decodes the URI targeted by the UPDATE Operation to pin it down to
2511 a specific resource and PDR. Before it can send the RDEOperationInit command to the RDE Device that
2512 owns the PDR, the MC must perform a BEJ encoding of the JSON payload it received from the Redfish
2513 client. Unlike the previous example, the BEJ encoded payload here is small enough to fit in the maximum
2514 transfer chunk, so the MC inlines it into the RDEOperationInit request command. Again, the
2515 contains_request_parameters Operation flag is not set, so the RDE Device will not expect supplemental
2516 parameters as part of this Operation.

2517 When the RDE Device receives the RDEOperationInit request command, it has everything it needs to
2518 begin work on the Operation. In this case, the RDE Device determines that performing the write will take
2519 longer than PT1, so the RDE Device spawns a long-running Task to process the write asynchronously
2520 and sends TaskSpawned in the OperationExecutionFlags to inform the MC.

2521 When it discovers that the RDE Device spawned a long-running Task, the MC adds a member to the
2522 Task collection it maintains and synthesizes a TaskMonitor URI to send back to the client in a location
2523 response header. At this point, the client can issue an HTTP GET to retrieve a status update on the Task;
2524 when it does so, the MC sends RDEOperationStatus to the RDE Device to get the status update and
2525 sends it back to the client as the result of the GET operation.

2526 At some point, the asynchronous Task finishes executing. When this happens, the RDE Device issues a
2527 PlatformEventMessage to send a TaskCompletion event to the MC. (This presupposes that the RDE
2528 Device and the MC both support asynchronous eventing. Were this not the case, the RDE Device would
2529 still generate the TaskCompletion event, but would wait for the MC to invoke the
2530 PollForPlatformEventMessage command to report the event.) Regardless of which way the MC gets the
2531 event, it then sends the RDEOperationStatus command one last time in order to retrieve the final results
2532 from the Operation. The next time the client performs a GET on the TaskMonitor, the MC can send back
2533 the final results of the Operation. Finally, the MC finalizes the Operation via RDEOperationComplete at
2534 which point the MC can delete the Task collection member and the TaskMonitor URI and the RDE Device
2535 can free up any buffers associated with the Operation and/or Task.

2536

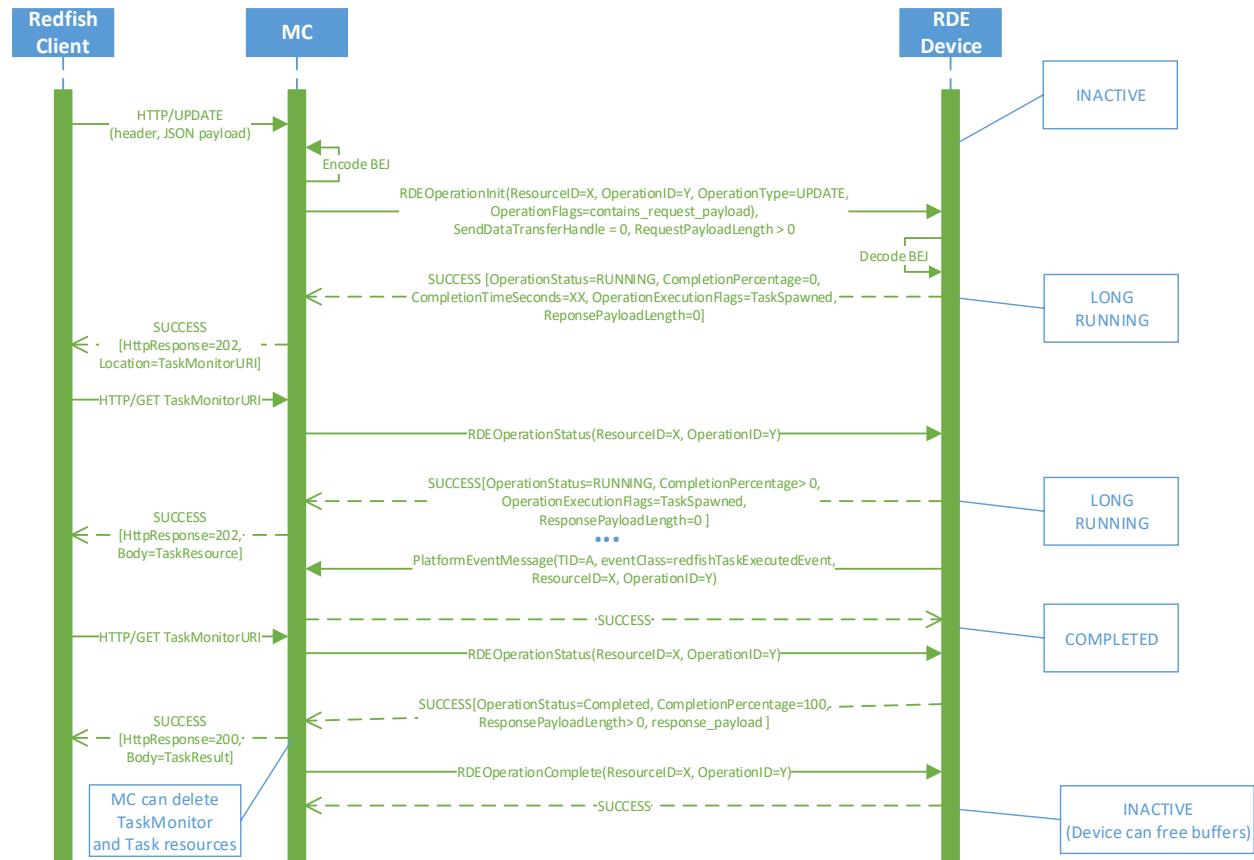


Figure 12 – Write Operation with long-running Task ladder diagram

9.2.2 Operation/Task overview workflow diagrams (Operation perspective)

This clause describes the operating behavior for MCs and RDE Devices over the lifecycle of Operations from an Operation-centric perspective. The workflow diagrams are split between simpler, short-lived Operations and those that spawn a Task to be processed asynchronously. These workflow diagrams are intended to capture the standard flow for the execution of most Operations, but do not cover every possible error condition. For full precision, refer to clause 9.2.3.

9.2.2.1 Operation overview workflow diagram

Table 46 details the information presented visually in Figure 13.

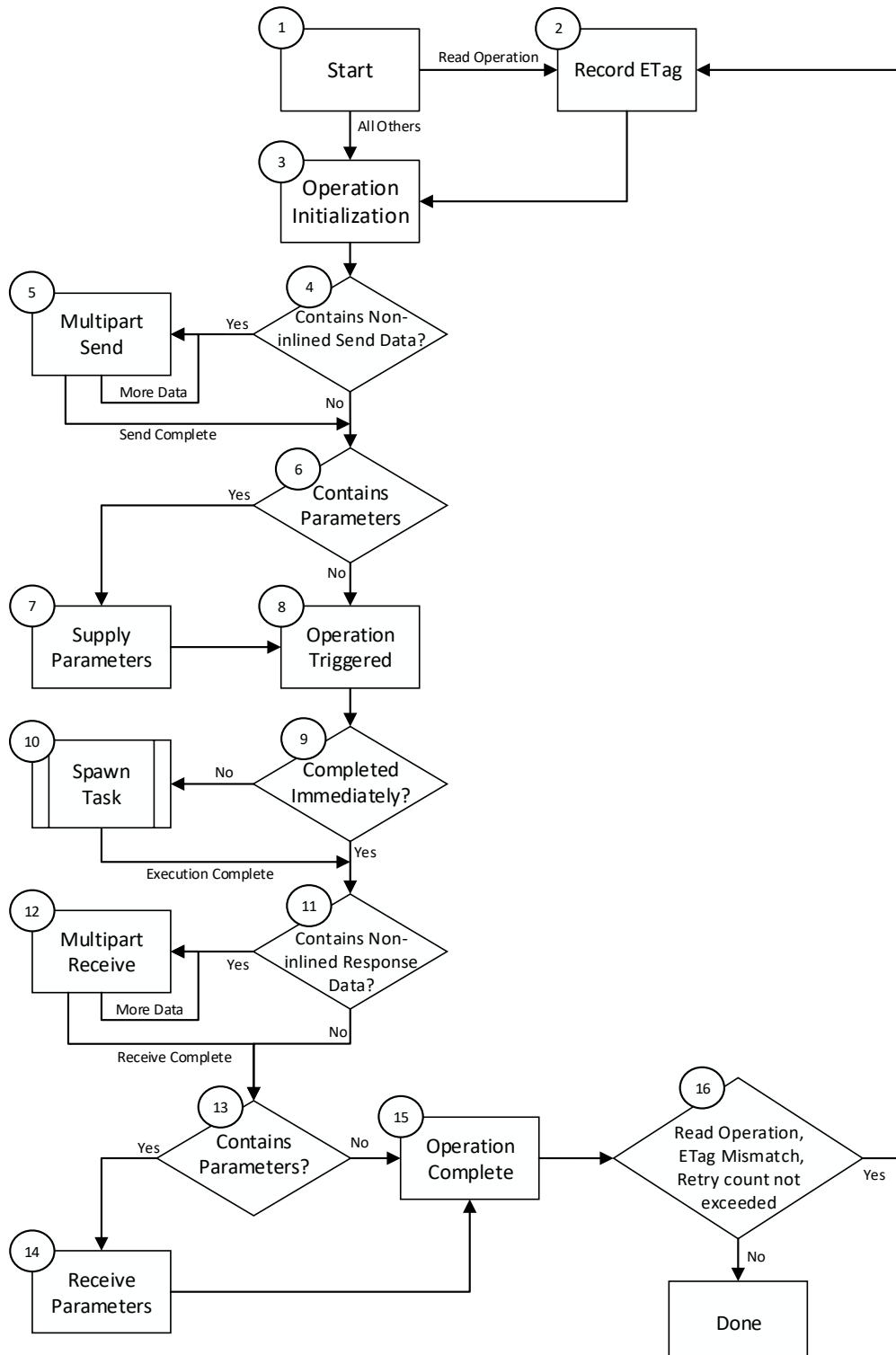
Table 46 – Operation lifecycle overview

Step	Description	Condition	Next Step
1 – START	The lifecycle of an Operation begins when the MC receives an HTTP/HTTPS operation from the client	For any Redfish Read (HTTP/HTTPS GET) operations	2
		For any other operation	3
2 – GET_DIGEST	For Read operations, the MC may use the GetResourceETag command to record a digest snapshot. If the RDE Device advertised that it is capable of	Unconditional	3

Step	Description	Condition	Next Step
	reading a resource atomically in the NegotiateRedfishParameters command (see clause 11.1), the MC may skip this step if the read does not span multiple resources (such as through the \$expand request header)		
3 – INITIALIZE_OP	The MC checks the HTTP/HTTPS operation to see if it contains JSON payload data to be transferred to the RDE Device. If so, it performs a BEJ encoding of this data. It then uses the RDEOperationInit command to begin the Operation with the RDE Device	Unconditional	4
4 – SEND_PAYLOAD_CHK	If the RDE Operation contains BEJ payload data, it needs to be sent to the RDE Device. The payload data may be inlined in the RDEOperationInit request message if the resulting message fits within the negotiated transfer chunk limit.	If the Operation contains a non-inlined payload (that did not fit in the RDEOperationInit request message)	5
		Otherwise	6
5 – SEND_PAYLOAD	The MC uses the RDEMultipartSend command to send BEJ-encoded payload data to the RDE Device	The last chunk of payload data has been sent	6
		More data remains to be sent	5
6 – SEND_PARAMS_C HK	If the RDE Operation contains uncommon request parameters or headers that need to be transferred to the RDE Device, they need to be sent to the RDE Device.	If the Operation contains supplemental request parameters	7
		Otherwise	8
7 – SEND_PARAMS	The MC uses the SupplyCustomRequestParameters command to submit the supplemental request parameters to the RDE Device	Unconditional	8
8 – TRIGGERED	The RDE Device begins executing the Operation as soon as it has all the information it needs for it	Unconditional	9
9 – COMPLETION_CH K	The RDE Device must respond to the triggering command (that provided the last bit of information needed to execute the Operation or a follow-up call to RDEOperationStatus if the last data was sent via RDEMultipartSend) within PT1 time. If it can complete the Operation within that timeframe, it does not need to spawn a Task to run the Operation asynchronously.	If the RDE Device is able to complete the Operation “quickly”	11
		Otherwise	10
10 – LONG_RUN	If the RDE Device was not able to complete the Operation quickly enough it spawns a Task to execute asynchronously. See Figure 14 for details of the Task sublifecycle.	Once the Task finishes executing	11

Step	Description	Condition	Next Step
11 – RCV_PAYLOAD_C HK	<p>If the Operation contains a response payload, the RDE Device encodes it in BEJ format. If the response payload is small enough to inline and have the response message fit within the negotiated maximum transfer chunk, the RDE Device appends it to the response message of:</p> <ul style="list-style-type: none"> • RDEOperationInit, if this was the triggering command • SupplyCustomRequestParameters, if this was the triggering command • The first RDEOperationStatus after a triggering RDEMultipartSend command, if the Operation could be completed “quickly” • The first RDEOperationStatus after asynchronous Task execution finishes, otherwise 	If there is no payload or if the payload is small enough to be inlined into the response message of the appropriate command	13
		Otherwise	12
12 – RCV_PAYLOAD	The MC uses the RDEMultipartReceive command to retrieve the BEJ-encoded payload from the RDE Device	The last chunk of payload data has been sent	13
		More data remains to be sent	12
13 – RCV_PARAMS_CH K	The MC checks to see if the Operation result contains supplemental response parameters	If the Operation contains response parameters	14
		Otherwise	15
14 – RCV_PARAMS	The MC uses the RetrieveCustomResponseParameters command to obtain the supplemental response parameters. NOTE The transfer of a non-inlined response payload and supplemental response parameters may be performed in either order. For simplicity, the flow shown assumes that a response payload would be transferred before supplemental response parameters; however, the opposite assumption could be made by swapping the positions of blocks 11/12 with blocks 13/14 in the figure.	Unconditional	15
15 – COMPLETE	The MC sends the RDEOperationComplete command to finalize the Operation	n/a	n/a

Step	Description	Condition	Next Step
16 – CMP_DIGEST	If the Operation was a read and the MC collected an ETag in step 2, the MC compares the response ETag with the one it collected in step 2 to check for a consistency violation. If it finds one, it may retry the operation or give up. The MC may skip the consistency check (treat it as successful without checking) if the RDE Device advertised that it has the capability to read a resource atomically in its response to the NegotiateRedfishParameters command (see clause 11.1).	Read operation and mismatched ETags and retry count not exceeded	2
		Not a read, no ETag collected, the ETAGs match, or retry count exceeded	n/a: Done



2548

2549

Figure 13 – RDE Operation lifecycle overview (holistic perspective)2550 **9.2.2.2 Task overview workflow diagram**

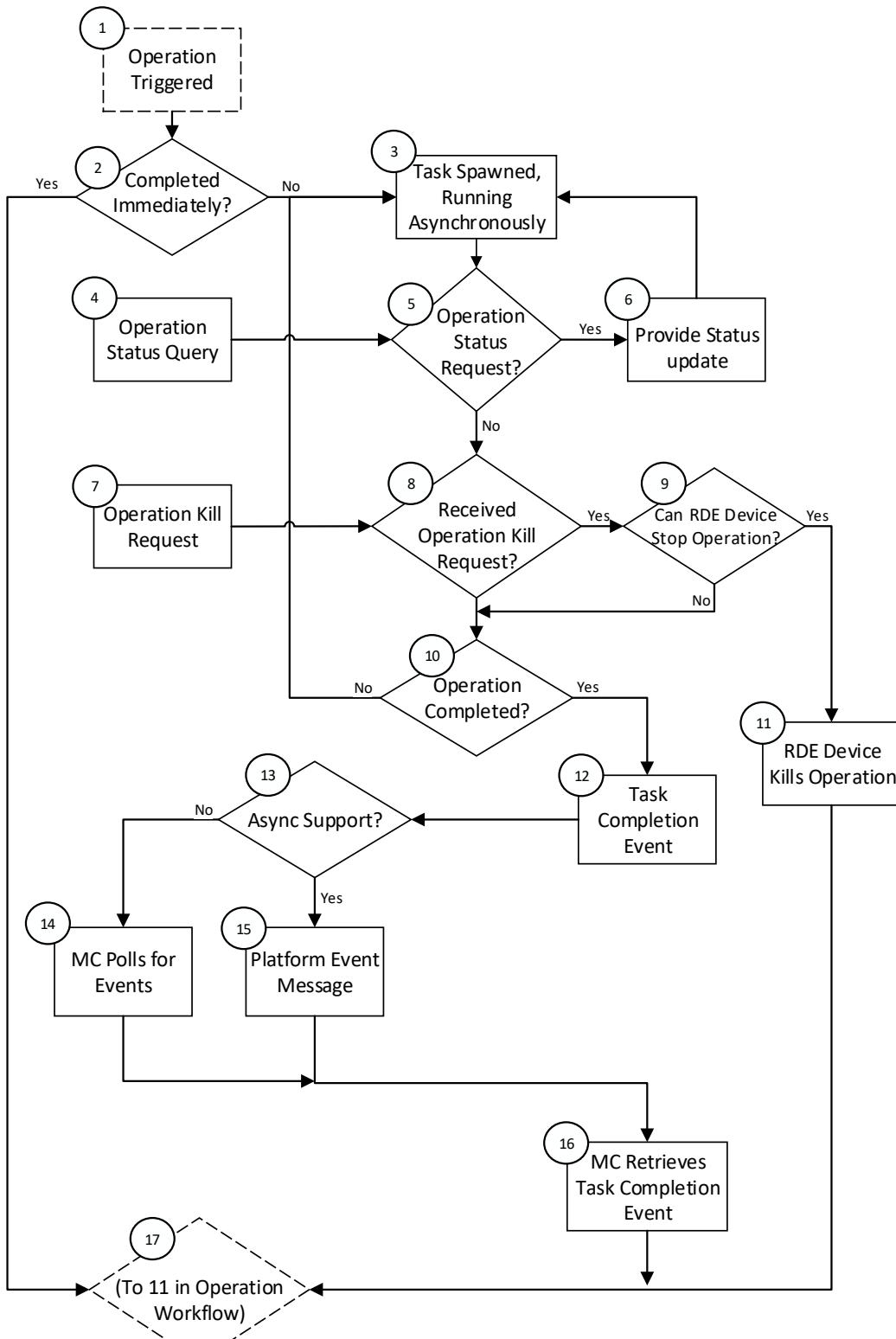
2551 Table 47 details the information presented visually in Figure 14.

Table 47 – Task lifecycle overview

Current Step	Description	Condition	Next Step
1 – TRIGGERED	The sublifecycle of a Task begins when the RDE Device receives all the data it needs to perform an Operation. (This corresponds to Step 8 in Table 46.)	Unconditional	2
2 – COMPLETION_CHK	The RDE Device must respond to the triggering command (that provided the last bit of information needed to execute the Operation) within PT1 time. If it cannot complete the Operation within that timeframe, it spawns a Task to run the Operation asynchronously.	If the RDE Device is able to complete the Operation quickly (not a Task)	17
		Otherwise	3
3 – LONG_RUN	The RDE Device runs the Task asynchronously	Unconditional	5
4 – REQ_STATUS	The MC may issue an RDEOperationStatus command at any time to the RDE Device.	If issued	5
5 – STATUS_CHK	The RDE Device must be ready to respond to an RDEOperationStatus command while running a Task asynchronously	Status request received	6
		No status request received	8
6 – PROCESS_STATUS	The RDE Device sends a response to the RDEOperationStatus command to provide a status update	Unconditional	3
7 – REQ_KILL	The MC may issue an RDEOperationKill command at any time to the RDE Device	Unconditional	8
8 – KILL_CHK	The RDE Device must be ready to respond to an RDEOperationKill command while running a Task asynchronously	Kill request received	9
		No kill request received	10
9 – PROCESS_KILL	If the RDE Device receives a kill request, it may or may not be able to abort the Task. This is an RDE Device-specific decision about whether the Task has crossed a critical boundary and must be completed	RDE Device cannot stop the Task	10
		RDE Device can stop the Task	11
10 – ASYNC_EXECUTE_FINISHED_CHK	The RDE Device should eventually complete the Task	If the Task has been completed	12
		If the Task has not been completed	3
11 – PERFORM_ABORT	The RDE Device aborts the Task in response to a request from the MC	Unconditional	17

Current Step	Description	Condition	Next Step
12 – COMPLETION_EVENT	After the Task is complete, the RDE Device generates a Task Completion Event	Unconditional	13
13 – ASYNC_CHK	The mechanism by which the Task completion Event reaches the MC depends on how the MC configured the RDE Device for Events via the PLDM for Platform Monitoring and Control SetEventReceiver command	Asynchronous Events	14
		Polled Events	15
14 – PEM_POLL	The MC uses the PollForPlatformEventMessage command to check for Events and finds the Task Completion Event	Unconditional	16
15 – PEM_SEND	The RDE Devices sends the Task Completion Event to the MC asynchronously via the PlatformEventMessage command	Unconditional	16
16 – GET_TASK_FOLLOWUP	After receiving the Task completion Event, the MC uses the RDEOperationStatus command to retrieve the outcome of the Task's execution	Unconditional	17
17 – TASK_DONE	The MC checks the response message to the RDEOperationStatus command to see if there is a response payload (This corresponds to Step 11 in Table 46.)	See Step 11 in Table 48	See Step 11 in Table 48

2553



2554

2555

2556

Figure 14 – RDE Task lifecycle overview (holistic perspective)

2557 9.2.3 RDE Operation state machine (RDE Device perspective)

2558 The following clauses describe the operating behavior for the lifecycle of Operations and Tasks from an
2559 RDE Device-centric perspective. Table 48 details the information presented visually in Figure 15. The
2560 states presented in this state machine are not (collectively) the total state for the RDE Device, but rather
2561 the state for the Operation. The total state for the RDE Device would involve separate instances of the
2562 Task/Operation state machine replicated once for each of the concurrent Operations that the RDE Device
2563 and the MC negotiated to support at registration time.

2564 9.2.3.1 State definitions

2565 The following states shall be implemented by the RDE Device for each Operation it is supporting:

- 2566 • INACTIVE
 - 2567 – INACTIVE is the default Operation state in which the RDE Device shall start after
2568 initialization. In this state, the RDE Device is not processing an Operation as it has not
2569 received an RDEOperationInit command from the MC.
- 2570 • NEED_INPUT
 - 2571 – After receiving the RDEOperationInit command, the RDE Device moves to this state if it is
2572 expecting additional Operation-specific parameters or a payload that was not inlined in the
2573 RDEOperationInit command.
- 2574 • TRIGGERED
 - 2575 – Once the RDE Device receives everything it needs to execute an Operation, it begins
2576 executing it immediately. If the triggering command – the command that supplied the last
2577 bit of data needed to execute the Operation – was RDEOperationInit or
2578 SupplyCustomRequestParameters, the response message to the triggering command
2579 reflects the initial results for the Operation. However, if the triggering command was a
2580 RDEMultipartSend, initial results are deferred until the MC invokes the
2581 RDEOperationStatus command. This state captures the case where the Operation was
2582 triggered by a RDEMultipartSend and the MC has not yet sent an RDEOperationStatus
2583 command to get initial results. In this state, the RDE Device may execute the Operation;
2584 alternatively, it may wait to receive RDEOperationStatus to begin execution.
- 2585 • TASK_RUNNING
 - 2586 – If the RDE Device cannot complete the Operation within the timeframe needed for the
2587 response to the command that triggered it, the RDE Device spawns a Task in which to
2588 execute the Operation asynchronously.
- 2589 • HAVE_RESULTS
 - 2590 – When execution of the Operation produces a response parameters or a response payload
2591 that does not fit in the response message for the command that triggered the Operation (or
2592 detected its completion, if a Task was spawned or if there was a payload but no custom
2593 request parameters), the RDE Device remains in this state until the MC has collected all of
2594 these results.
- 2595 • COMPLETED
 - 2596 – The RDE Device has completed processing of the Operation and awaits acknowledgment
2597 from the MC that it has received all Operation response data. This acknowledgment is
2598 done by the MC issuing the RDEOperationComplete command. When the RDE Device
2599 receives this command, it may discard any internal records or state it has maintained for
2600 the Operation.
- 2601 • FAILED

- 2602 – The MC has explicitly killed the Operation or an error prevented execution of the
 2603 Operation.
- 2604 • ABANDONED
- 2605 – If MC fails to progress the Operation through this state machine, the RDE Device may
 2606 abort the Operation and mark it as abandoned.

2607 **9.2.3.2 Operation lifecycle state machine**

2608 Figure 15 illustrates the state transitions the RDE Device shall implement. Each bubble represents a
 2609 particular state as defined in the previous clause. Upon initialization, system reboot, or an RDE Device
 2610 reset the RDE Device shall enter the INACTIVE state.

2611

Table 48 – Task lifecycle state machine

Current State	Trigger	Response	Next State
0 - INACTIVE	RDEOperationInit - RDE Device not ready - RDE Device does not wish to specify a deferral timeframe	ERROR_NOT_READY, HaveCustomResponseParameter's bit in OperationExecutionFlags not set	INACTIVE
	RDEOperationInit - RDE Device not ready - RDE Device does wish to specify a deferral timeframe	ERROR_NOT_READY, HaveCustomResponseParameter's bit in OperationExecutionFlags set	HAVE_RESULTS
	RDEOperationInit, SupplyCustomRequestParameters, RDEOperationStatus, RDEOperationKill, or RDEOperationComplete - Resource ID does not correspond to any active Operation	ERROR_NO_SUCH_RESOURCE	INACTIVE
	RDEOperationInit, wrong resource type for POST Operation in request (e.g., Action sent to a collection)	ERROR_WRONG_LOCATION_TYPE	INACTIVE
	RDEOperationInit, RDE Device does not allow the requested Operation	ERROR_NOT_ALLOWED	INACTIVE
	RDEOperationInit, RDE Device does not support the requested Operation	ERROR_UNSUPPORTED	INACTIVE
	RDEOperationInit, Operation ID has MSBit clear (indicating that the MC is attempting to initiate an Operation with an ID reserved for the RDE Device)	ERROR_INVALID_DATA	INACTIVE
	RDEOperationInit, request contains any other error	Various, depending on the specific error encountered	INACTIVE
	RDEOperationStatus	OPERATION_INACTIVE	INACTIVE

Current State	Trigger	Response	Next State
	RDEOperationInit; - valid request - Operation Flags indicate request non-inlined payload or parameters to be sent from MC to RDE Device	Success	NEED_INPUT
	RDEOperationInit; - valid request - Operation Flags indicate no request payload to be sent from MC to RDE Device (or request payload inlined in RDEOperationInit request message) - request flags indicate no supplemental parameters needed - RDE Device cannot complete Operation within PT1	Success	TASK_RUNNING
	RDEOperationInit; - valid request - Operation Flags indicate no request payload to be sent from MC to RDE Device (or request payload inlined in RDEOperationInit request message) - request flags indicate no supplemental parameters needed - RDE Device completes Operation within PT1 - response flags indicate response parameters or a non-inlined response payload to be retrieved from RDE Device	Success	HAVE_RESULTS
	RDEOperationInit; - valid request - Operation Flags indicate no request payload to be sent from MC to RDE Device (or request payload inlined in RDEOperationInit request message) - request flags indicate no supplemental parameters needed - RDE Device completes Operation within PT1 - no payload to be retrieved	Success	COMPLETED

Current State	Trigger	Response	Next State
	from RDE Device or response payload fits within response message such that total response message size is within negotiated maximum transfer chunk - no response parameters		
	RDEOperationKill (any combination of flags)	ERROR_UNEXPECTED	INACTIVE
	Any other Operation command	ERROR	INACTIVE
1- NEED_INPUT	RDEOperationInit, same rdeOpID	ERROR_OPERATION_EXISTS; no disruption to existing Operation	NEED_INPUT
	RDEOperationInit, different rdeOpID	Success or ERROR_CANNOT_CREATE_OPERATION, depending on whether the RDE Device has another slot to execute an Operation	The new Operation is tracked in a separate copy of the state machine; this Operation remains in NEED_INPUT
	RDEOperationInit request flags indicated supplemental parameters and or payload data to be sent; $T_{abandon}$ timeout waiting for RDEMultipartSend/SupplyCustomRequestParameterscommand	None	ABANDONED
	RDEOperationKill; - neither run_to_completion nor discard_record flag set	Success	FAILED
	RDEOperationKill; - run_to_completion flag not set - discard_record flag set	Success	INACTIVE
	RDEOperationKill; - run_to_completion flag set - discard_record flag not set	ERROR_INVALID_DATA	NEED_INPUT
	RDEOperationKill; - both run_to_completion and discard_record flags both set	ERROR_UNEXPECTED (can't run to completion without further input from MC, so the request is contradictory)	NEED_INPUT
	RDEOperationStatus	OPERATION_NEED_INPUT	NEED_INPUT
	RDEMultipartSend; - data inlined or Operation flags indicate no payload data	ERROR_UNEXPECTED	NEED_INPUT
	RDEMultipartSend; - transfer error	Error specific to type of transfer failure encountered	NEED_INPUT (MC may retry send or use

Current State	Trigger	Response	Next State
			RDEOperationKill to abort Operation)
	RDEMultipartSend; - more data to be sent from the MC to the RDE Device after this chunk	Success	NEED_INPUT
	RDEMultipartSend; - no more data to be sent from the MC to the RDE Device after this chunk - RDEOperationInit request flags indicated supplemental parameters needed - params not yet sent	Success	NEED_INPUT
	RDEMultipartSend; - no more data to be sent after this chunk - RDEOperationInit request flags indicated supplemental parameters not needed or parameters already sent	Success	TRIGGERED
	RDEMultipartSend; - data already transferred	ERROR_UNEXPECTED	NEED_INPUT
	SupplyCustomRequestParameters; - Operation includes unsupported ETag operation or query option	ERROR_UNSUPPORTED	FAILED
	SupplyCustomRequestParameters; - Operation flags indicated supplemental parameters not needed or payload data remaining to be sent	ERROR_UNEXPECTED	NEED_INPUT
	SupplyCustomRequestParameters; - no payload data remaining to be sent - ETagOperation is ETAG_IF_MATCH and no ETag matches or ETagOperation is ETAG_IF_NONE_MATCH and an ETAG matches	ERROR_ETAG_MATCH	FAILED
	SupplyCustomRequestParameters; - request contains unsupported RDE custom header	ERROR_UNRECOGNIZED_CUS TOM_HEADER	FAILED
	SupplyCustomRequestParameters; - no payload data remaining	Error specific to type of failure encountered	FAILED

Current State	Trigger	Response	Next State
1 - IDLE	to be sent - Error occurs in processing of Operation		
	SupplyCustomRequestParameters; - no payload data remaining to be sent - RDE Device cannot complete Operation within PT1	Success	LONG_RUNNING
	SupplyCustomRequestParameters; - no payload data remaining to be sent - RDE Device completes Operation within PT1 - response flags indicate response parameters or a non-inlined response payload to be retrieved from RDE Device	Success	HAVE_RESULTS
	SupplyCustomRequestParameters; - no payload data remaining to be sent - RDE Device completes Operation within PT1 - no payload to be retrieved from RDE Device or response payload fits within response message such that total response message size is within negotiated maximum transfer chunk - no response parameters	Success	COMPLETED
	RDEMultipartReceive, RDEOperationComplete	ERROR_UNEXPECTED	NEED_INPUT
	Any other Operation command	ERROR	NEED_INPUT
	RDEOperationInit, same rdeOpID	ERROR_OPERATION_EXISTS; no disruption to existing Operation	TRIGGERED
2 - TRIGGERED	RDEOperationInit, different rdeOpID	Success or ERROR_CANNOT_CREATE_OPERATION, depending on whether the RDE Device has another slot to execute an Operation	The new Operation is tracked in a separate copy of the state machine; this Operation remains in TRIGGERED
	T _{abandon} timeout waiting for RDEOperationStatus command	None	ABANDONED
	RDEOperationStatus; error occurs in processing of Operation	Error specific to type of failure encountered	FAILED

Current State	Trigger	Response	Next State
	RDEOperationKill - discard_results flag set - any other flag set	ERROR_INVALID_DATA	TRIGGERED
	RDEOperationKill - discard_results flag set - no other flag set	ERROR_UNEXPECTED	TRIGGERED
	RDEOperationKill - run_to_completion flag set - discard_record flag not set	ERROR_INVALID_DATA	TRIGGERED
	RDEOperationKill; - Operation executing; Operation can be killed - neither run_to_completion nor discard_record flag set	Success	FAILED
	RDEOperationKill - Operation executing - Operation can be killed - run_to_completion flag not set - discard_record flag set	Success	INACTIVE
	RDEOperationKill - Operation executing - Operation can be killed - both run_to_completion and discard_record flags set	ERROR_UNEXPECTED (can't run to completion without further input from MC to move it to TASK_RUNNING, so the request is contradictory)	TRIGGERED
	RDEOperationKill - Operation executing - Operation cannot be killed or Operation execution finished - any combination of run_to_completion and discard_record flags set	ERROR_OPERATION_UNKILLABLE	TRIGGERED
	RDEOperationStatus; - RDE Device cannot complete Operation within PT1	OPERATION_TASK_RUNNING	TASK_RUNNING
	RDEOperationStatus; - RDE Device completes Operation within PT1 - payload to be retrieved from RDE Device or response parameters present	Success	HAVE_RESULTS
	RDEOperationStatus; - RDE Device completes	Success	COMPLETED

Current State	Trigger	Response	Next State
	<ul style="list-style-type: none"> - Operation within PT1 - no payload or payload fits within response message such that total response message size is within negotiated maximum transfer chunk - no response parameters 		
	RDEMultipartSend, RDEMultipartReceive, SupplyCustomRequestParameters, RetrieveCustomResponseParameters, RDEOperationComplete	ERROR_UNEXPECTED	TRIGGERED
	Any other Operation command	ERROR	TRIGGERED
3 - TASK_RUNNING	RDEOperationInit, same rdeOpID	ERROR_OPERATION_EXISTS; no disruption to existing Operation	TASK_RUNNING
	RDEOperationInit, different rdeOpID	Success or ERROR_CANNOT_CREATE_OPERATION, depending on whether the RDE Device has another slot to execute an Operation	The new Operation is tracked in a separate copy of the state machine; this Operation remains in TASK_RUNNING
	Error occurs in processing of Operation	None	FAILED
	RDEOperationKill <ul style="list-style-type: none"> - discard_results flag set - any other flag set 	ERROR_INVALID_DATA	TASK_RUNNING
	RDEOperationKill <ul style="list-style-type: none"> - discard_results flag set - no other flag set 	ERROR_UNEXPECTED	TASK_RUNNING
	RDEOperationKill <ul style="list-style-type: none"> - run_to_completion flag set - discard_record flag not set 	ERROR_INVALID_DATA	TASK_RUNNING
	RDEOperationKill; <ul style="list-style-type: none"> - Operation can be aborted - neither run_to_completion nor discard_record flag set 	Success	FAILED
	RDEOperationKill <ul style="list-style-type: none"> - Operation executing - Operation can be killed - run_to_completion flag not set - discard_record flag set 	Success	INACTIVE
	RDEOperationKill <ul style="list-style-type: none"> - Operation executing 	Success	TASK_RUNNING

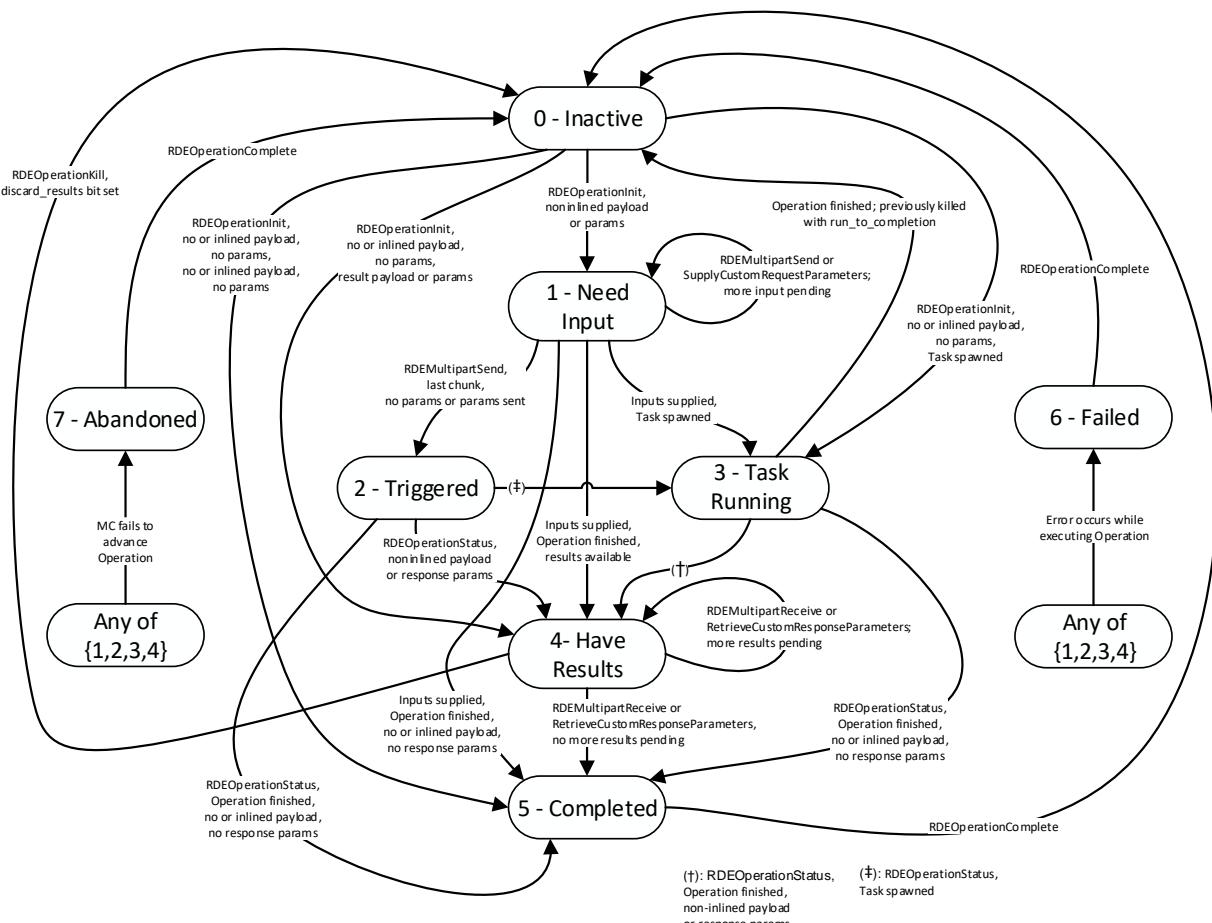
Current State	Trigger	Response	Next State
	- Operation can be killed - both run_to_completion and discard_record flags set		
	RDEOperationKill; - Operation cannot be aborted or has finished execution - any combination of run_to_completion and discard_record flags set	ERROR_OPERATION_UNKILLABLE	TASK RUNNING
	Execution finishes; - Operation not killed	Generate Task Completion Event (only once per Operation). Send to MC via PlatformEventMessage if MC configured the RDE Device to use asynchronous Events via SetEventReceiver; otherwise, MC will retrieve Event via PollForPlatformEventMessage. See Event lifecycle in clause 9.3 for further details	TASK_RUNNING
	Execution finishes; - Operation killed	None	INACTIVE
	Execution finished; - Task Completion Event received by MC; - $T_{abandon}$ timeout waiting for RDEOperationStatus command	None	ABANDONED
	RDEOperationStatus; - execution not yet finished	OPERATION_TASK_RUNNING	TASK RUNNING
	RDEOperationStatus; - execution finished - payload to be retrieved from RDE Device or response parameters present	OPERATION_HAVE_RESULTS	HAVE_RESULTS
	RDEOperationStatus; - execution finished - no payload or payload fits in response message such that total response message size is within negotiated maximum transfer chunk - no response parameters	OPERATION_COMPLETED	COMPLETED
	RDEMultipartSend, RDEMultipartReceive, RDEOperationComplete	ERROR_UNEXPECTED	TASK_RUNNING
	Any other Operation command	ERROR	TASK_RUNNING

Current State	Trigger	Response	Next State
4 - HAVE_RESULTS	RDEOperationInit, same rdeOpID	ERROR_OPERATION_EXISTS; no disruption to existing Operation	HAVE_RESULTS
	RDEOperationInit, different rdeOpID	Success or ERROR_CANNOT_CREATE_OPERATION, depending on whether the RDE Device has another slot to execute an Operation	The new Operation is tracked in a separate copy of the state machine; this Operation remains in HAVE_RESULTS
	RDEOperationKill - discard_results flag set - any other flag set	ERROR_INVALID_DATA	HAVE_RESULTS
	RDEOperationKill - discard_results flag set - no other flag set	SUCCESS	INACTIVE
	RDEOperationKill - run_to_completion flag set - discard_record flag not set	ERROR_INVALID_DATA	HAVE_RESULTS
	RDEOperationKill; - any other combination of run_to_completion and discard_record flags set	ERROR_OPERATION_UNKILLABLE	HAVE_RESULTS
	RDEOperationStatus	OPERATION_HAVE_RESULTS	HAVE_RESULTS
	RDEMultipartReceive; - MC aborts transfer	Do not send data; Success; Prepare to restart transfer with next RDEMultipartReceive command	HAVE_RESULTS
	RDEMultipartReceive; - transfer error	Error specific to type of transfer failure encountered	HAVE_RESULTS (MC may retry receive or abandon Operation)
	RDEMultipartReceive; - more data to transfer from the RDE Device to the MC after this chunk	Send data; Success	HAVE_RESULTS
	RDEMultipartReceive; - no more data to transfer from the RDE Device to the MC after this chunk - response parameters to send	Send data; Success	HAVE_RESULTS
	RDEMultipartReceive; - no more data to transfer from the RDE Device to the MC after this chunk - no response parameters	Send data; Success	COMPLETED

Current State	Trigger	Response	Next State
4 - ABANDONED	present		
	Tabandon timeout waiting for RDEMultipartReceive and/or RetrieveCustomResponseParameters commands (depending on type of results still to be retrieved)	None	ABANDONED
	ReceiveCustomResponseParameters <ul style="list-style-type: none"> - RDE Device was not ready when RDEOperationInit command was sent and wished to specify a deferral timeframe 	Deferral Timeframe; Success	FAILED
	ReceiveCustomResponseParameters <ul style="list-style-type: none"> - response payload data not yet transferred 	Success	HAVE_RESULTS
	ReceiveCustomResponseParameters <ul style="list-style-type: none"> - response payload data partially transferred 	ERROR_UNEXPECTED	HAVE_RESULTS
	ReceiveCustomResponseParameters <ul style="list-style-type: none"> - no response payload or all response payload data transferred 	Success	COMPLETED
	Any other Operation or transfer command	Error	HAVE_RESULTS
5 - COMPLETED	RDEOperationInit, same rdeOpID	ERROR_OPERATION_EXISTS; no disruption to existing Operation	COMPLETED
	RDEOperationInit, different rdeOpID	Success or ERROR_CANNOT_CREATE_OPERATION, depending on whether the RDE Device has another slot to execute an Operation	The new Operation is tracked in a separate copy of the state machine; this Operation remains in COMPLETED
	RDEOperationKill <ul style="list-style-type: none"> - discard_results flag set - any other flag set 	ERROR_INVALID_DATA	COMPLETED
	RDEOperationKill <ul style="list-style-type: none"> - discard_results flag set - no other flag set 	ERROR_UNEXPECTED	COMPLETED
	RDEOperationKill <ul style="list-style-type: none"> - run_to_completion flag set - discard_record flag not set 	ERROR_INVALID_DATA	COMPLETED
	RDEOperationKill;	ERROR_OPERATION_UNKILLAB	COMPLETED

Current State	Trigger	Response	Next State
	- any other combination of run_to_completion and discard_record flags set	LE	
	RDEOperationStatus	OPERATION_COMPLETED	COMPLETED
	RDEOperationComplete	Success	INACTIVE
	Any other Operation command	Error	COMPLETED
6 - FAILED	RDEOperationInit, same rdeOpID	ERROR_OPERATION_EXISTS Operation	FAILED
	RDEOperationInit, different rdeOpID	Success or ERROR_CANNOT_CREATE_OPERATION, depending on whether the RDE Device has another slot to execute an Operation	The new Operation is tracked in a separate copy of the state machine; this Operation remains in FAILED
	RDEOperationKill - discard_results flag set - any other flag set	ERROR_INVALID_DATA	FAILED
	RDEOperationKill - discard_results flag set - no other flag set	ERROR_UNEXPECTED	FAILED
	RDEOperationKill - run_to_completion flag set - discard_record flag not set	ERROR_INVALID_DATA	FAILED
	RDEOperationKill - any other combination of run_to_completion and discard_record flags set	ERROR_OPERATION_FAILED	FAILED
	RDEOperationStatus	OPERATION_FAILED	FAILED
	RDEOperationComplete	Success	INACTIVE
	Any other Operation command	ERROR_OPERATION_FAILED	FAILED
7 - ABANDONED	RDEOperationInit, same rdeOpID	ERROR_OPERATION_EXISTS Operation	ABANDONED
	RDEOperationInit, different rdeOpID	Success or ERROR_CANNOT_CREATE_OPERATION, depending on whether the RDE Device has another slot to execute an Operation	The new Operation is tracked in a separate copy of the state machine; this Operation remains in ABANDONED
	RDEOperationKill - discard_results flag set - any other flag set	ERROR_INVALID_DATA	ABANDONED

Current State	Trigger	Response	Next State
	RDEOperationKill <ul style="list-style-type: none"> - discard_results flag set - no other flag set 	ERROR_UNEXPECTED	ABANDONED
	RDEOperationKill <ul style="list-style-type: none"> - run_to_completion flag set - discard_record flag not set 	ERROR_INVALID_DATA	ABANDONED
	RDEOperationKill; <ul style="list-style-type: none"> - any other combination of run_to_completion and discard_record flags set 	ERROR_OPERATION_ABANDONED	ABANDONED
	RDEOperationStatus	OPERATION_ABANDONED	ABANDONED
	RDEOperationComplete	Success	INACTIVE
	Any other Operation command	ERROR_OPERATION_ABANDONED	ABANDONED

2612
2613

2614

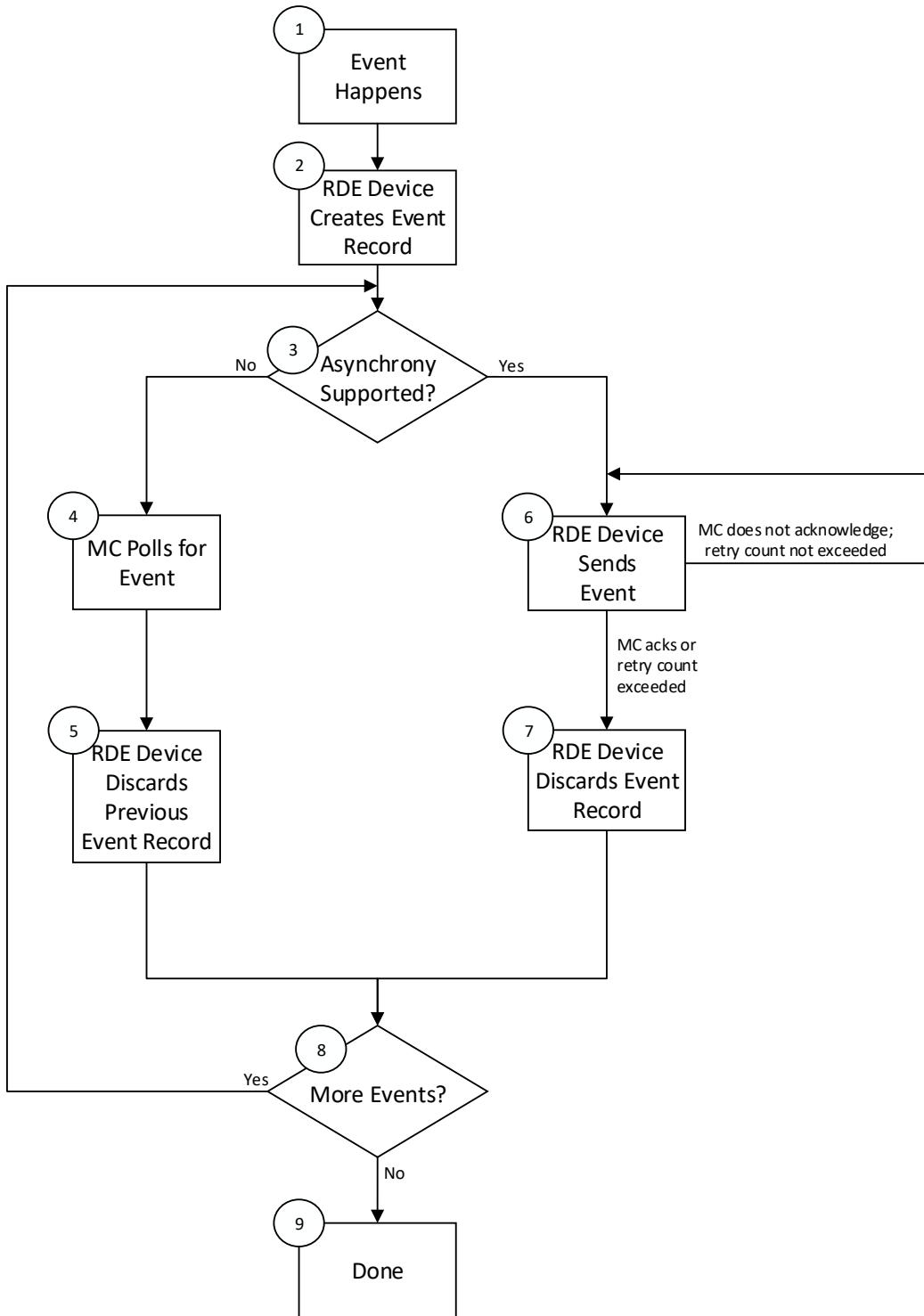
Figure 15 – Operation lifecycle state machine (RDE Device perspective)

2615 9.3 Event lifecycle

2616 Table 49 describes the operating behavior for MCs and RDE Devices over the lifecycle of Events
 2617 depicted visually in Figure 16. This sequence applies to both Task completion Events and schema-based
 2618 Events. MC and RDE Device implementations of RDE shall comply with the sequences presented here.

2619 **Table 49 – Event lifecycle overview**

Current State	Description	Condition	Next Step
1 – OCCURS	The lifecycle of an Event begins when the Event occurs.	Unconditional	2
2 – RECORD	The RDE Device creates an Event record.	Unconditional	3
3 – ASYNC_CHK	The MC used the SetEventReceiver command to configure the RDE Device either to use asynchronous Events or to be polled for Events.	Asynchronous Events	6
		Polling	4
4 – EVT_POLL	The MC polls for Events using the PollForPlatformEventMessage command and discovers the Event.	Unconditional	5
5 – DISC_PREV	If the PollForPlatformEventMessage command request message reflected a previous Event to be acknowledged, the RDE Device discards the record for that previous Event.	Unconditional	8
6 – EVT_SEND	The RDE Device issues a PlatformEventMessage command to the MC to notify it of the Event.	MC acknowledges the Event	7
		MC does not acknowledge the Event and retry count (PN1, see DSP0240) not exceeded	6
		MC does not acknowledge the Event and retry count exceeded	7
7 – DISC_RCRD	The RDE Device discards its Event record.	Unconditional	8
8 – MORE_CHK	Are there more Events (in the asynchronous case) or there was an Event to acknowledge (in the synchronous case)?	Yes	3
		No	9
9 – DONE	Event processing is complete.	n/a	-



2620

2621

Figure 16 – Redfish event lifecycle overview

2622 10 PLDM commands for Redfish Device Enablement

2623 This clause provides the list of command codes that are used by MCs and RDE Devices that implement
 2624 PLDM Redfish Device Enablement as defined in this specification. The command codes for the PLDM
 2625 messages are given in Table 50. RDE Devices and MCs shall implement all commands where the entry
 2626 in the “Command Requirement for RDE Device” or “Command Requirement for MC”, respectively, is
 2627 listed as Mandatory. RDE Devices and MCs may optionally implement any commands where the entry in
 2628 the “Command Requirement for RDE Device” or “Command Requirement for MC”, respectively, is listed
 2629 as Optional.

2630 **Table 50 – PLDM for Redfish Device Enablement command codes**

Command	Command Code	Command Requirement for RDE Device	Command Requirement for MC	Command Requestor (Initiator)	Reference
Discovery and Schema Management Commands					
NegotiateRedfishParameters	0x01	Mandatory	Mandatory	MC	See 11.1
NegotiateMediumParameters	0x02	Mandatory	Mandatory	MC	See 11.2
GetSchemaDictionary	0x03	Mandatory	Mandatory	MC	See 11.3
GetSchemaURI	0x04	Mandatory	Mandatory	MC	See 11.4
GetResourceETag	0x05	Mandatory	Mandatory	MC	See 11.5
GetOEMCount	0x06	Optional	Optional	MC	See 11.6
GetOEMName	0x07	Optional	Optional	MC	See 11.7
GetRegistryCount	0x08	Optional	Optional	MC	See 11.8
GetRegistryDetails	0x09	Optional	Optional	MC	See 11.9
SelectRegistryVersion	0x0A	Optional	Optional	MC	See 11.10
GetMessageRegistry	0x0B	Optional	Optional	MC	See 11.11
GetSchemaFile	0x0C	Optional	Optional	MC	See 11.12
Reserved	0x0D-0x0F				
RDE Operation and Task Commands					
RDEOperationInit	0x10	Mandatory	Mandatory	MC	See 12.1
SupplyCustomRequestParameters	0x11	Mandatory	Mandatory	MC	See 12.2
RetrieveCustomResponseParameters	0x12	Conditional ⁴	Mandatory	MC	See 12.3
RDEOperationComplete	0x13	Mandatory	Mandatory	MC	See 12.4
RDEOperationStatus	0x14	Mandatory	Mandatory	MC	See 12.5
RDEOperationKill	0x15	Optional	Optional	MC	See 12.6
RDEOperationEnumerate	0x16	Mandatory	Optional	MC	See 12.7
Reserved	0x17-0x2F				
Multipart Transfer Commands					
RDEMultipartSend	0x30	Conditional ¹	Conditional ¹	MC	See 13.1
RDEMultipartReceive	0x31	Mandatory	Mandatory	MC	See 13.2
Reserved	0x32-0x3F				

Command	Command Code	Command Requirement for RDE Device	Command Requirement for MC	Command Requestor (Initiator)	Reference
Reserved For Future Use					
Reserved	0x40-0xFF				
Referenced PLDM Base Commands (PLDM Type 0)					
NegotiateTransferSize	See DSP0240	Conditional ₁	Conditional ₁	MC	See DSP0240
MultipartSend	See DSP0240	Conditional ₁	Conditional ₁	MC	See DSP0240
MultipartReceive	See DSP0240	Conditional ₁	Conditional ₁	MC	See DSP0240

Command	Command Code	Command Requirement for RDE Device	Command Requirement for MC	Command Requestor (Initiator)	Reference
Referenced PLDM for Monitoring and Control Commands (PLDM Type 2)					
GetPDRRepositoryInfo	See DSP0248	Mandatory	Mandatory	MC	See DSP0248
GetPDR	See DSP0248	Mandatory	Mandatory	MC	See DSP0248
SetEventReceiver	See DSP0248	Conditional ₂	Conditional ₂	MC	See DSP0248
PlatformEventMessage	See DSP0248	Optional ₃	Conditional ₃	RDE Device	See DSP0248
PollForPlatformEventMessage	See DSP0248	Optional ₂	Conditional ₃	MC	See DSP0248

2631 Notes:

- 2632 1) Either RDEMultipartSend or PLDM common MultipartSend is required if the RDE Device intends to support write Operations. RDE versions of bulk transfer commands shall be used if either the RDE Device or the MC does not support PLDM common versions; if both the RDE Device and the MC advertise support for PLDM common versions of bulk transfer commands (via the PLDM Base NegotiateTransferSize command), the RDE versions shall not be used.
- 2637 2) SetEventReceiver is mandatory if the RDE Device intends to support asynchronous messaging for Events via PlatformEventMessage.
- 2639 3) RDE Devices and MCs must support either PlatformEventMessage or PollForPlatformEventMessage in order to enable Event support.
- 2641 4) SupplyCustomResponseParameter is required if the RDE Device ever sets the HaveCustomResponseParameters flag in the OperationExecutionFlags field of the response message for a triggering command.

2644 11 PLDM for Redfish Device Enablement – Discovery and schema commands

2646 This clause describes the commands that are used by RDE Devices and MCs that implement the discovery and schema management commands defined in this specification. The command codes for the PLDM messages are given in Table 50.

2649 11.1 NegotiateRedfishParameters command (0x01) format

2650 This command enables the MC to negotiate general Redfish parameters with an RDE Device. The MC shall send this command to the RDE Device prior to any other RDE command. An RDE Device that supports multiple mediums shall provide the same response to this command independent of the medium on which this command was issued.

2654 When the RDE Device receives a request with data formatted per the Request Data section below, it shall respond with data formatted per the Response Data section. For a non-SUCCESS CompletionCode, only the CompletionCode field of the Response Data shall be returned.

2657

Table 51 – NegotiateRedfishParameters command format

Type	Request data
uint8	<p>MCConcurrencySupport</p> <p>The maximum number of concurrent outstanding Operations the MC can support for this RDE Device. Must be > 0; a value of 1 indicates no support for concurrency. A value of 255 (0xFF) shall be interpreted to indicate that no such limit exists. Upon completion of this command, the RDE Device shall not initiate an Operation if MCConcurrencySupport (or DeviceConcurrencySupport whichever is lower) Operations are already active.</p>
bitfield16	<p>MCFeatureSupport</p> <p>Operations and functionality supported by the MC; for each, 1b indicates supported, 0b not:</p> <ul style="list-style-type: none"> [15:9] - reserved [8] - BEJ v1.1 encoding and decoding supported; 1b = yes [7] - events_supported; 1b = yes. Must be 1b if MC supports Redfish Events or Long-running Tasks. [6] - action_supported; 1b = yes [5] - replace_supported; 1b = yes [4] - update_supported; 1b = yes [3] - delete_supported; 1b = yes [2] - create_supported; 1b = yes [1] - read_supported; 1b = yes. All MCs that implement PLDM for Redfish Device Enablement shall support read Operations [0] - head_supported; 1b = yes
Type	Response data
enum8	<p>CompletionCode</p> <p>value: { PLDM_BASE_CODES }</p>
uint8	<p>DeviceConcurrencySupport</p> <p>The maximum number of concurrent outstanding Operations the RDE Device can support. Must be > 0; a value of 1 indicates no support for concurrency. A value of 255 (0xFF) shall be interpreted to indicate that no such limit exists. Regardless of the RDE Device's level of support for concurrency, it shall not initiate an Operation if a limit indicated by MCConcurrencySupport has already been reached.</p>
bitfield8	<p>DeviceCapabilitiesFlags</p> <p>Capabilities for this RDE Device; for each, 1b indicates the RDE Device has the capability, 0b not:</p> <ul style="list-style-type: none"> [7:3] - reserved [2] - bej_1_1_support: the RDE Device supported encoding and decoding BEJ version 1.1 [1] - expand_support: the RDE Device can process a \$expand request query parameter (expressed via the LinkExpand field of the SupplyCustomRequestParameters command) [0] - atomic_resource_read: the RDE Device can respond to a read of an entire resource atomically, guaranteeing consistency of the read

Type	Response data (continued)
bitfield16	<p>DeviceFeatureSupport</p> <p>Operations and functionality supported by this RDE Device; for each, 1b indicates supported, 0b not:</p> <ul style="list-style-type: none"> [15:8] - reserved [7] - events_supported; 1b = yes. Must be 1b if RDE Device supports Redfish Events or Long-running Tasks. Shall match PLDM Event support indicated via support for PLDM for Platform Monitoring and Control (DSP0248) SetEventReceiver command [6] - action_supported; 1b = yes [5] - replace_supported; 1b = yes [4] - update_supported; 1b = yes [3] - delete_supported; 1b = yes [2] - create_supported; 1b = yes [1] - read_supported; 1b = yes. All RDE Devices shall support read Operations [0] - head_supported; 1b = yes
uint32	<p>DeviceConfigurationSignature</p> <p>A signature (such as a CRC-32) calculated across all RDE PDRs and dictionaries that the RDE Device supports. This calculation should be performed as if all of the RDE PDRs and dictionaries were concatenated together into a single block of memory. The RDE Device may order the RDE PDRs and dictionaries in any sequence it chooses; however, it should be consistent in this ordering across invocations of the NegotiateRedfishParameters command. The RDE Device may use any method to generate the signature so long as it guarantees that a change to one or more RDE PDRs and/or dictionaries will not result in the same signature being generated.</p> <p>The RDE Device may generate the signature in any manner it sees fit; however, the signature generated for any given set of PDRs and dictionaries shall match any previous signature generated for the same set of PDRs and dictionaries. If a nonzero result from an RDE Device signature matches the result from a previous invocation of this command, the MC may generally assume that any RDE PDRs and/or dictionaries it has stored for the RDE Device remain unchanged and can be reused. However, MCs must be aware that any hashing algorithm risks a false positive match in result between hashes of two distinct sets of data. To mitigate this risk, MCs should utilize a secondary check, such as comparing the updateTime field in the PLDM for Platform Monitoring and Control GetPDRRepositoryInfo command response message to that from when PDRs were previously retrieved.</p>
varstring	<p>DeviceProviderName</p> <p>An informal name for the RDE Device</p>

2658 11.2 NegotiateMediumParameters command (0x02) format

2659 This command enables the MC to negotiate medium-specific parameters with an RDE Device. The MC
 2660 should invoke this command on each communication medium (e.g., RBT, SMBus, PCIe VDM) on which it
 2661 intends to interface with the RDE Device. The MC shall send this command over the transport for a
 2662 particular medium to negotiate parameters for that medium. When the RDE Device receives a request
 2663 with data formatted per the Request Data section below, it shall respond with data formatted per the
 2664 Response Data section. For a non-SUCCESS CompletionCode, only the CompletionCode field of the
 2665 Response Data shall be returned.

2666

Table 52 – NegotiateMediumParameters command format

Type	Request data
uint32	<p>MCMmaximumTransferChunkSizeBytes</p> <p>An indication of the maximum amount of data the MC can support for a single message transfer. This value represents the size of the PLDM header and PLDM payload; medium specific header information shall not be included in this calculation. For cases of larger messages, a protocol-specific multipart transfer shall be utilized.</p> <p>All MC implementations shall support a transfer size of at least 64 bytes.</p> <p>NOTE For MCTP-based mediums, this is relative to the message size, not the packet size.</p>
Type	Response data
enum8	<p>CompletionCode</p> <p>value: { PLDM_BASE_CODES }</p> <p>If the MC reports a maximum transfer size of less than 64 bytes, the RDE Device shall respond with completion code ERROR_INVALID_DATA.</p>
uint32	<p>DeviceMaximumTransferChunkSizeBytes</p> <p>The maximum number of bytes that the RDE Device can support in a chunk for a single message transfer. This value represents the size of the PLDM header and PLDM payload; medium specific header information shall not be included in this calculation. If this value is greater than MCMmaximumTransferChunkSizeBytes, the RDE Device shall “throttle down” to using the smaller value. If this value is smaller, the MC shall not attempt a transfer exceeding it.</p> <p>All RDE Device implementations shall support a transfer size of at least 64 bytes.</p> <p>NOTE For MCTP-based mediums, this is relative to the message size, not the packet size.</p>

2667

11.3 GetSchemaDictionary command (0x03) format

2668
2669
2670
2671
2672
2673
2674
2675
2676

This command enables the MC to retrieve a dictionary (full or truncated; see clause 7.2.3) associated with a Redfish Resource PDR. After invoking the GetSchemaDictionary command, the MC shall, upon receipt of a successful completion code and a valid read transfer handle, invoke one or more RDEMultipartReceive commands (clause 13.2) to transfer data for the dictionary from the RDE Device. The MC shall only have one dictionary, schema, or message registry retrieval in process from a given RDE Device at any time. In the event that the MC begins a dictionary, schema, or message registry retrieval when a previous retrieval has not yet completed (i.e., more chunks of dictionary or schema data remain to be retrieved), the previous retrieval is implicitly aborted and the RDE Device may discard any data associated with the transfer.

2677
2678
2679
2680
2681
2682
2683

MCs are discouraged from invoking the GetSchemaDictionary command in the middle of processing an RDE Operation (excluding when it is running asynchronously as a long-running task). Instead, whenever possible, they should run the Operation back to the INACTIVE state and only then retrieve dictionaries needed to finalize processing of Operation results. (Ideally, these dictionaries would have been cached before the Operation was initialized.) Neither the GetSchemaDictionary command nor any RDEMultipartReceive commands used to retrieve a dictionary shall be construed as resetting the abandonment timer ($T_{abandon}$, see clause 7.6).

2684
2685
2686

When the RDE Device receives a request with data formatted per the Request Data section below, it shall respond with data formatted per the Response Data section if it supports the command. For a non-SUCCESS CompletionCode, only the CompletionCode field of the Response Data shall be returned.

2687

Table 53 – GetSchemaDictionary command format

Type	Request data
uint32	<p>ResouceID The ResouceID of any resource in the Redfish Resource PDR from which to retrieve the dictionary. A ResouceID of 0xFFFF FFFF may be supplied to retrieve dictionaries common to all RDE Device resources (such as the event or annotation dictionary) without referring to an individual resource.</p>
schemaClass	<p>RequestedSchemaClass The class of schema being requested</p>
Type	Response data
enum8	<p>CompletionCode value: { PLDM_BASE_CODES, ERROR_UNSUPPORTED, ERROR_NO SUCH RESOURCE } If the RDE Device does not support a schema of the type requested, it shall return CompletionCode ERROR_UNSUPPORTED. If the supplied Resource ID does not correspond to a collection, but the RequestedSchemaClass is COLLECTION_MEMBER_TYPE, the RDE Device shall return ERROR_INVALID_DATA.</p>
uint8	<p>DictionaryFormat The format of the dictionary as specified in the dictionary's VersionTag, defined in clause 7.2.3.2.</p>
uint32	<p>TransferHandle A data transfer handle that the MC shall use to retrieve the dictionary data via one or more RDEMultipartReceive commands (see clause 13.2). In conjunction with a non-failed CompletionCode, the RDE Device shall return a valid transfer handle.</p>

2688 11.4 GetSchemaURI command (0x04) format

2689 This command enables the MC to retrieve the formal URI for one of the RDE Device's schemas.

2690 When the RDE Device receives a request with data formatted per the Request Data section below, it shall
2691 respond with data formatted per the Response Data section if it supports the command. For a non-
2692 SUCCESS CompletionCode, only the CompletionCode field of the Response Data shall be returned.

2693

Table 54 – GetSchemaURI command format

Type	Request data
uint32	<p>ResouceID The ResouceID of a resource in a Redfish Resource PDR from which to retrieve the URI. A ResouceID of 0xFFFF FFFF may be supplied to retrieve URIs for schemas common to all RDE Device resources (such as for the annotation schema) without referring to an individual resource.</p>
schemaClass	<p>RequestedSchemaClass The class of schema being requested</p>
uint8	<p>OEMExtensionNumber Shall be zero for a standard DMTF-published schema, or the one-based OEM extension to a standard schema</p>

Type	Response data
enum8	<p>CompletionCode value: { PLDM_BASE_CODES, ERROR_UNSUPPORTED, ERROR_NO SUCH RESOURCE }</p> <p>For an out-of-range OEMExtensionNumber, the RDE Device shall return ERROR_INVALID_DATA. If the RDE Device does not support a schema of the type requested, it shall return CompletionCode ERROR_UNSUPPORTED.</p>
uint8	<p>StringFragmentCount</p> <p>The number of fragments N into which the URI string is broken; shall be greater than zero. The MC shall concatenate these together to reassemble the final string.</p>
varstring	<p>SchemaURI [0]</p> <p>URI string fragment for the schema. The reassembled string shall be the canonical URI for the JSON Schema used by the RDE Device.</p>
...	...
varstring	<p>SchemaURI [N - 1]</p> <p>URI string fragment for the schema. The reassembled string shall be the canonical URI for the JSON Schema used by the RDE Device.</p>

2694 11.5 GetResourceETag command (0x05) format

2695 This command enables the MC to retrieve a hashed summary of the data contained immediately within a
2696 resource, including all OEM extensions to it, or of all data within an RDE Device. The retrieved ETag shall
2697 reflect the underlying data as specified in the Redfish specification ([DSP0266](#)).

2698 When the RDE Device receives a request with data formatted per the Request Data section below, it shall
2699 respond with data formatted per the Response Data section if it supports the command. For a non-
2700 SUCCESS CompletionCode, only the CompletionCode field of the Response Data shall be returned.

2701 In the event that the RDE Device cannot provide a response to this command within the PT1 time period
2702 (defined in [DSP0240](#)), the RDE Device may provide completion code ETAG_CALCULATION_ONGOING
2703 and continue the process of generating the ETag. The MC may then poll for the completed ETag by
2704 repeating the same GetResourceETag command that it gave that previously yielded this result. The RDE
2705 Device in turn shall signal whether it has completed the calculation by responding with a completion code
2706 of either SUCCESS (the calculation is done) or ETAG_CALCULATION_ONGOING (otherwise). It is
2707 recommended that the MC delay for an integer multiple of PT1 between retry attempts.

2708 Following an invocation of this command that results in a completion code of
2709 ETAG_CALCULATION_ONGOING, any other RDE command, including an invocation of
2710 GetResourceETag with a different request message, shall be interpreted by the RDE Device as implicitly
2711 canceling the pending GetResourceETag command and cause it to stop generating the ETag. The RDE
2712 Device shall then proceed to respond to the newly arrived command normally.

2713 NOTE ETags provided via this command are not escaped for inclusion in JSON data. MCs should be aware that
2714 performing a raw comparison of an ETag retrieved from this command with one received as part of BEJ-
2715 encoded JSON data will result in a mismatch as the ETag format requires characters that must be escaped
2716 in JSON data.

2717

Table 55 – GetResourceETag command format

Type	Request data
uint32	ResouceID The ResouceID of a resource in the the Redfish Resource PDR for the instance from which to get an ETag digest; or 0xFFFF FFFF to get a global digest of all resource-based data within the RDE Device
Type	Response data
enum8	CompletionCode value: { PLDM_BASE_CODES, ERROR_NO SUCH_RESOURCE, ETAG_CALCULATION_ONGOING }
varstring	ETag The RFC7232 -compliant ETag string data; the string text format shall be UTF-8. Either a strong or a weak etag may be returned. This field shall be omitted if the CompletionCode is not SUCCESS.

2718 11.6 GetOEMCount command (0x06) format

2719 This command enables the MC to retrieve the number of OEM extensions for a schema.

2720 When the RDE Device receives a request with data formatted per the Request Data section below, it shall
 2721 respond with data formatted per the Response Data section if it supports the command. For a non-
 2722 SUCCESS CompletionCode, only the CompletionCode field of the Response Data shall be returned.

2723 **Table 56 – GetOEMCount command format**

Type	Request data
uint32	ResouceID The ResouceID of the resource in the Redfish Resource PDR from which to retrieve the OEM count. A ResouceID of 0xFFFF FFFF may be supplied to retrieve OEM counts for schemas common to all RDE Device resources (such as the event dictionary) without referring to an individual resource.
schemaClass	RequestedSchemaClass The class of schema being requested. <small>NOTE Redfish does not allow OEM extensions to Annotation and Registry schemas.</small>
Type	Response data
enum8	CompletionCode value: { PLDM_BASE_CODES, ERROR_NO SUCH_RESOURCE }
uint8	OEMCount The number of OEM extensions associated with the schema. For schema classes that do not support OEM extensions this value shall be zero.

2724 11.7 GetOEMName command (0x07) format

2725 This command enables the MC to retrieve information about the name associated with an OEM extension
 2726 to a schema (including schemas for which OEM information is available in a Redfish Resource PDR).

2727 RDE Devices shall enumerate OEM extensions in lexicographic order.

2728 When the RDE Device receives a request with data formatted per the Request Data section below, it shall
 2729 respond with data formatted per the Response Data section if it supports the command. For a non-
 2730 SUCCESS CompletionCode, only the CompletionCode field of the Response Data shall be returned.

2731

Table 57 – GetOEMName command format

Type	Request data
uint32	ResourceId The ResouceID of any resource in the Redfish Resource PDR from which to retrieve an OEM name. A ResouceID of 0xFFFF FFFF may be supplied to retrieve OEM names for extensions to schemas common to all RDE Device resources (such as the event dictionary) without referring to an individual resource.
schemaClass	RequestedSchemaClass The class of schema being requested
uint8	OEMIndex The zero-based index of the OEM extension about which information is to be retrieved. The total number of OEM extensions supported by an RDE Device for a given schema may be retrieved via the GetOEMCount command; the index supplied here should be less than that count.
Type	Response data
enum8	CompletionCode value: { PLDM_BASE_CODES, ERROR_NO SUCH_RESOURCE } A response code of ERROR_INVALID_DATA shall be used to indicate when the supplied index does not exist in the schema or when the schema class does not support OEM schemas.
varstring	OEMName The OEM name associated with the extension

2732 **11.8 GetRegistryCount command (0x08) format**

2733 This command enables the MC to retrieve the number of message registries supported by an RDE
 2734 Device.

2735 When the RDE Device receives a request with data formatted per the Request Data section below, it shall
 2736 respond with data formatted per the Response Data section if it supports the command. For a non-
 2737 SUCCESS CompletionCode, only the CompletionCode field of the Response Data shall be returned.

2738

Table 58 – GetRegistryCount command format

Type	Request data
--	None
Type	Response data
enum8	CompletionCode value: { PLDM_BASE_CODES }
uint8	RegistryCount The number of registries supported by the Device

2739 **11.9 GetRegistryDetails command (0x09) format**

2740 This command enables the MC to retrieve information about a message registry an RDE Device supports.

- 2741 RDE Devices shall enumerate message registries in lexicographic order and return message registry
 2742 versions in reverse numeric order (most recent versions listed first). The RDE Device shall truncate the
 2743 list and decrease the count as needed to ensure that the response message fits within the negotiated
 2744 message size, thereby omitting mention of support for older versions.
- 2745 When the RDE Device receives a request with data formatted per the Request Data section below, it shall
 2746 respond with data formatted per the Response Data section if it supports the command. For a non-
 2747 SUCCESS CompletionCode, only the CompletionCode field of the Response Data shall be returned.

2748

Table 59 – GetRegistryDetails command format

Type	Request data
uint8	RegistryIndex The zero-based index of the message registry about which information is to be retrieved. The total number of registries supported by an RDE Device may be retrieved via the GetRegistryCount command; the index supplied here should not exceed that count.
Type	Response data
enum8	CompletionCode value: { PLDM_BASE_CODES } ERROR_INVALID_DATA: The supplied index does not correspond to a supported registry
varstring	RegistryPrefix The Redfish prefix (name without version information) associated with the registry
varstring	RegistryURI URI at which the registry schema is published
uint8[2]	RegistryLanguage Language in which the registry is published, as an ISO 639-1 two-letter code
uint8	VersionCount The number N of registry versions the RDE Device supports for this registry
ver32	Version [0] First (newest) version of the registry supported
...	...
ver32	Version [N - 1] Last (oldest) version of the registry supported

2749 **11.10 SelectRegistryVersion command (0x0A) format**

- 2750 This command enables the MC to specify the version of a supported Redfish message registry that the
 2751 RDE device should use. By default, the RDE Device shall utilize the latest version of the registry that it
 2752 supports.
- 2753 When the RDE Device receives a request with data formatted per the Request Data section below, it shall
 2754 respond with data formatted per the Response Data section if it supports the command.

2755

Table 60 – SelectRegistryVersion command format

Type	Request data
uint8	<p>RegistryIndex</p> <p>The zero-based index of the message registry for which the registry is to be selected. The total number of registries supported by an RDE Device may be retrieved via the GetRegistryCount command; the index supplied here should be less than that count.</p>
ver32	<p>RegistryVersion</p> <p>Version of the registry to be used</p>
Type	Response data
enum8	<p>CompletionCode</p> <p>value: { PLDM_BASE_CODES }</p> <p>ERROR_INVALID_DATA: The supplied index does not correspond to a supported registry or the supplied version is not supported</p>

2756 11.11 GetMessageRegistry command (0x0B) format

2757 This command enables the MC to retrieve the formal JSON registry for a Redfish message registry
 2758 supported by the RDE device. After invoking the GetMessageRegistry command, the MC shall, upon
 2759 receipt of a successful completion code and a valid read transfer handle, invoke one or more
 2760 RDEMultipartReceive commands (clause 13.2) to transfer data for the registry from the RDE Device. The
 2761 MC shall only have one dictionary, schema, or message registry retrieval in process from a given RDE
 2762 Device at any time. In the event that the MC begins a dictionary, schema, or message registry retrieval
 2763 when a previous retrieval has not yet completed (i.e., more chunks of dictionary or schema data remain to
 2764 be retrieved), the previous retrieval is implicitly aborted and the RDE Device may discard any data
 2765 associated with the transfer.

2766 When the RDE Device receives a request with data formatted per the Request Data section below, it shall
 2767 respond with data formatted per the Response Data section if it supports the command. For a non-
 2768 SUCCESS CompletionCode, only the CompletionCode field of the Response Data shall be returned.

2769

Table 61 – GetMessageRegistry command format

Type	Request data
uint8	<p>RegistryIndex</p> <p>The zero-based index of the message registry to be retrieved. The total number of registries supported by an RDE Device may be retrieved via the GetRegistryCount command; the index supplied here should not exceed that count.</p>

Type	Response data
enum8	<p>CompletionCode value: { PLDM_BASE_CODES } ERROR_INVALID_DATA: The supplied index does not correspond to a supported registry</p>
uint8	<p>SchemaFormat Bitwise OR of two values: Text format: { RAW_UTF8 = 0; GZIP_UTF8 = 1 } Schema format: { JSON = 0x10; CSDL = 0x20; YAML = 0x30 } In most cases, a message registry would be supplied as a GZIP'd UTF-8 JSON document, the value supplied would be 0x10.</p>
uint32	<p>TransferHandle A data transfer handle that the MC shall use to retrieve the registry data via one or more RDEMultipartReceive commands (see clause 13.2). In conjunction with a non-failed CompletionCode, the RDE Device shall return a valid transfer handle.</p>

2770 11.12 GetSchemaFile command (0x0C) format

2771 This command enables the MC to retrieve the formal schema for a Redfish resource supported by the
2772 RDE device. After invoking the GetSchemaFile command, the MC shall, upon receipt of a successful
2773 completion code and a valid read transfer handle, invoke one or more RDEMultipartReceive commands
2774 (clause 13.2) to transfer data for the schema from the RDE Device. The MC shall only have one
2775 dictionary, schema, or message registry retrieval in process from a given RDE Device at any time. In the
2776 event that the MC begins a dictionary, schema, or message registry retrieval when a previous retrieval
2777 has not yet completed (i.e., more chunks of dictionary or schema data remain to be retrieved), the
2778 previous retrieval is implicitly aborted and the RDE Device may discard any data associated with the
2779 transfer. MCs should reference the version and signature of schemas, as documented in Redfish
2780 Resource PDRs, wherever possible to avoid duplicate download of schema files.

2781 When the RDE Device receives a request with data formatted per the Request Data section below, it shall
2782 respond with data formatted per the Response Data section if it supports the command. For a non-
2783 SUCCESS CompletionCode, only the CompletionCode field of the Response Data shall be returned.

2784 **Table 62 – GetSchemaFile command format**

Type	Request data
uint32	<p>ResourceID The ResourceID of a Redfish Resource PDR from which to retrieve the schema for an associated resource. A ResourceID of 0xFFFF FFFF may be supplied to retrieve a schema common to all RDE Device resources (such as the event or annotation dictionary) without referring to an individual resource.</p>
schemaClass	<p>RequestedSchemaClass The class of schema being requested</p>
uint8	<p>OEMOffset The offset for an OEM extension schema (see 11.6). A value of 0xFF shall be interpreted as requesting the base (standard) schema, including for schemas that do not support OEM extensions.</p>

Type	Response data
enum8	<p>CompletionCode value: { PLDM_BASE_CODES, ERROR_NO SUCH_RESOURCE } ERROR_INVALID_DATA: The supplied OEMOffset is not valid</p>
uint8	<p>SchemaFormat Bitwise OR of two values: Text format: { RAW_UTF8 = 0; GZIP_UTF8 = 1 } Schema format: { JSON = 0x10; CSDL = 0x20; YAML = 0x30 } For example, for a CSDL (XML) format schema supplied as GZIP'd UTF-8 text, the value supplied would be 0x21.</p>
uint32	<p>TransferHandle A data transfer handle that the MC shall use to retrieve the registry data via one or more RDEMultipartReceive commands (see clause 13.2). In conjunction with a non-failed CompletionCode, the RDE Device shall return a valid transfer handle.</p>

12 PLDM for Redfish Device Enablement – RDE Operation and Task commands

This clause describes the Task commands that are used by RDE Devices and MCs that implement Redfish Device Enablement as defined in this specification. The command numbers for the PLDM messages are given in Table 50.

12.1 RDEOperationInit command (0x10) format

This command enables the MC to initiate a Redfish Operation with an RDE Device on behalf of a client. After invoking the RDEOperationInit command, the MC may, upon receipt of a successful completion code, invoke one or more RDEMultipartSend commands (clause 13.1) to transfer payload data of type `bejEncoding` to the RDE Device. The MC shall only use RDEMultipartSend to transfer the payload data if that data cannot fit in the request message of the RDEOperationInit command. After any payload has been transferred, the MC may invoke the SupplyCustomRequestParameters command if additional parameters are required. See clause 9 for more details on the Operation lifecycle.

After the RDE Device receives the RDEOperationInit command, if flags are not set to indicate that it should expect either payload data or custom request parameters, the RDE Device is triggered and shall begin execution of the Operation. Similarly, if the flags are set to expect a payload but not parameters, and the payload is contained inline in the request message, the RDE Device is implicitly triggered and shall begin execution of the Operation.

If triggered, the RDE Device shall respond with results if it is able to complete the Operation within the time period required for a response to this message. If there is a response payload that fits within the ResponsePayload field while maintaining a message size compatible with the negotiated maximum chunk size (see NegotiateMediumParameters, clause 11.2), the RDE Device shall include it within this response. Only if including a response payload would cause the message to exceed the negotiated chunk size may the RDE Device flag it for transfer via RDEMultipartReceive.

When the RDE Device receives a request with data formatted per the Request Data section below, it shall respond with data formatted per the Response Data section. Even with a non-SUCCESS CompletionCode, all fields of the Response Data shall be returned.

Table 63 – RDEOperationInit command format

Type	Request data
uint32	<p>ResourceID The resourceID of a resource in the Redfish Resource PDR for the data that is the target of this operation</p>
rdeOpID	<p>OperationID Identification number for this Operation; must match the one used for all commands relating to this Operation.</p> <p>NOTE Operation IDs with the most significant bit cleared are reserved for use by the RDE Device; it is an error for the MC to supply such an ID.</p>
enum8	<p>OperationType The type of Redfish Operation being performed. values: { OPERATION_HEAD = 0; OPERATION_READ = 1; OPERATION_CREATE = 2; OPERATION_DELETE = 3; OPERATION_UPDATE = 4; OPERATION_REPLACE = 5; OPERATION_ACTION = 6 }</p>
bitfield8	<p>OperationFlags Flags associated with this Operation:</p> <ul style="list-style-type: none"> [7:4] - reserved for future use [3] - excerpt_flag; if 1b, the RDE Device should perform an excerpt read (see 7.2.4.3.6) [2] - contains_custom_request_parameters; if 1b, the RDE Device should expect to receive a SupplyCustomRequestParameters command request before it may trigger the Operation [1] - contains_request_payload; if 0b, the Operation does not require data to be sent [0] - locator_valid; if 0b, the locator in the OperationLocator field shall be ignored
uint32	<p>SendDataTransferHandle Handle to be used with the first RDEMultipartSend command transferring BEJ formatted data for the operation. If no data is to be sent for this operation or if the request payload fits entirely within this request message, then it shall be zero (0x00000000) (see the RequestPayloadLength and RequestPayload fields below).</p>
uint8	<p>OperationLocatorLength Length in bytes of the OperationLocator for this Operation. This field shall be zero (0x00) if the locator_valid bit in the OperationFlags field above is set to 0b or if the OperationType field above is not one of OPERATION_UPDATE and OPERATION_ACTION.</p>
uint32	<p>RequestPayloadLength Length in bytes of the request payload in this message. This value shall be zero (0x00000000) under either of the following conditions:</p> <ul style="list-style-type: none"> • There is no request payload as indicated by contains_request_payload bit of the OperationFlags parameter above • The entire payload cannot fit within this message, subject to the maximum transfer chunk size as determined at registration time via the NegotiateMediumParameters command
bejLocator	<p>OperationLocator BEJ locator indicating where the new Operation is to take place within the resource specified in ResourceID.</p> <p>When the OperationType is set to OPERATION_ACTION, this field shall be set to the location of an action within a particular resource dictionary.</p> <p>This field may not be supported for other operation types and shall be omitted if the OperationLocatorLength field above is set to zero.</p>

null or bejEncoding	RequestPayload The request payload. The format of this parameter shall be null (consisting of zero bytes) if the RequestPayloadLength above is zero; it shall be bejEncoding otherwise.
Type	Response data
enum8	CompletionCode value: { PLDM_BASE_CODES, ERROR_CANNOT_CREATE_OPERATION, ERROR_NOT_ALLOWED, ERROR_WRONG_LOCATION_TYPE, ERROR_OPERATION_EXISTS, ERROR_UNSUPPORTED, ERROR_NO_SUCH_RESOURCE } Response codes ERROR_CANNOT_CREATE_OPERATION, ERROR_NOT_ALLOWED, ERROR_WRONG_LOCATION_TYPE, ERROR_OPERATION_EXISTS, ERROR_UNSUPPORTED, and ERROR_NO_SUCH_RESOURCE shall be interpreted to represent an operational failure, not a command failure.
enum8	OperationStatus values: { OPERATION_INACTIVE = 0; OPERATION_NEEDS_INPUT = 1; OPERATION_TRIGGERED= 2; OPERATION_RUNNING = 3; OPERATION_HAVE_RESULTS = 4; OPERATION_COMPLETED = 5, OPERATION_FAILED = 6, OPERATION_ABANDONED = 7 }
uint8	CompletionPercentage 0..100: percentage complete; 101-253: reserved for future use; 254: not supported or otherwise unable to estimate (but a valid Operation) 255: invalid Operation This value shall be zero if the Operation has not yet been triggered or if the Operation has failed.
uint32	CompletionTimeSeconds An estimate of the number of seconds remaining before the Operation is completed, or 0xFFFF FFFF if such an estimate cannot be provided. This value shall be 0xFFFF FFFF if the Operation has not yet been triggered or if the Operation has failed.
bitfield8	OperationExecutionFlags [7:4] - Reserved [3] - CacheAllowed – 1b = yes; shall be 0b for Operations other than read, head. Shall be 0b unless Operation has finished. Referring to RFC 7234 , a value of yes shall be considered as equivalent to Cache-Control response header value “public” and a value of no shall be considered as equivalent to Cache-Control response header value “no-store”. Other cache directives are not supported. The decision of whether to allow caching of data is up to the RDE Device. Typically, static data is allowed to be cached unless, for example, it represents sensitive data such as login credentials; data that changes over time is generally not marked as cacheable. To process the CacheAllowed flag, the MC shall behave as described in clause 7.2.4.2.7 [2] - HaveResultPayload – 1b = yes. Shall be 0b if Operation has not finished [1] - HaveCustomResponseParameters – 1b = yes. Shall be 0b if Operation has not finished [0] - TaskSpawned – 1b = yes For a failed Operation, this field shall be 0b for all flags other than HaveResultPayload, which may be 1b if a @Message.ExtendedInfo annotation is available to explain the result.
uint32	ResultTransferHandle A data transfer handle that the MC may use to retrieve a larger response payload via one or more RDEMultipartReceive commands (see clause 13.2). The RDE Device shall return a transfer handle of 0xFFFFFFFF if Operation execution has not finished or if the Operation has not yet been triggered. In the event of a failed Operation, or if the data fits entirely within the payload of this command response, or if there is no data to retrieve, the RDE Device shall return a null transfer handle, 0x00000000.

Type	Response data (continued)
bitfield8	<p>PermissionFlags</p> <p>Indicates the access level (types of Operations; see Table 33) granted to the resource targeted by the Operation.</p> <ul style="list-style-type: none"> [7: 6] - reserved for future use [5] - head access; 1b = access allowed [4] - delete access; 1b = access allowed [3] - create access; 1b = access allowed [2] - replace access; 1b = access allowed [1] - update access; 1b = access allowed [0] - read access; 1b = access allowed <p>Additional notes on processing PermissionFlags may be found in clause 7.2.4.2.8.</p>
uint32	<p>ResponsePayloadLength</p> <p>Length in bytes of the response payload in this message. This value shall be zero under any of the following conditions:</p> <ul style="list-style-type: none"> • The Operation has not yet been triggered. • The Operation status is not completed or failed, as indicated by the OperationStatus parameter above. For a failed Operation, a @Message.ExtendedInfo annotation may be supplied in the response payload. • There is no response payload as indicated by Bit 2 of the OperationExecutionFlags parameter above. • The entire payload cannot fit within this message, subject to the maximum transfer chunk size as determined at registration time via the NegotiateMediumParameters command.
varstring	<p>ETag</p> <p>String data for an ETag digest of the target resource; the string text format shall be UTF-8. The ETag shall be skipped (a string consisting of just the null terminator returned in this field) for any of the following actions: Action, Delete, Replace, and Update. The ETag shall also be skipped (a string consisting of just the null terminator returned in this field) if execution of the Operation has failed or not yet finished.</p> <p>Additional notes on processing ETAGs may be found in clause 7.2.4.2.4.</p> <p>NOTE ETAGs provided via this field are not escaped for inclusion in JSON data as they are primarily intended to be used for the ETag HTML header. MCs should be aware that performing a raw comparison of an ETag retrieved from this command with one received as part of BEJ-encoded JSON data will result in a mismatch as the ETag format requires characters that must be escaped in JSON data.</p>
null or bejEncoding	<p>ResponsePayload</p> <p>The response payload. The format of this parameter shall be null (consisting of zero bytes) if the ResponsePayloadLength above is zero; it shall be bejEncoding otherwise.</p>

2813 12.2 SupplyCustomRequestParameters command (0x11) format

2814 This command enables the MC to send custom HTTP/HTTPS X- headers and other uncommon request
 2815 parameters to an RDE Device to be applied to an Operation if the client's HTTP operation contains any
 2816 such parameters. The MC must not use this command to submit any headers for which a standard
 2817 handling is defined in either this specification or [DSP0266](#). If the client's HTTP operation does not contain
 2818 the parameters conveyed in this command, the MC shall not send this command as part of its processing
 2819 of the Operation.

2820 The MC shall only invoke this command in the event that at least one custom header or uncommon
 2821 request parameter needs to be transferred to the RDE Device. When sent, the

2822 **SupplyCustomRequestParameters** command shall be invoked after the MC sends the
2823 RDEOperationInit command.

2824 After the RDE Device receives the SupplyCustomRequestParameters command, if flags from the original
2825 RDEOperationInit command (see clause 12.1) were not set to indicate that it should expect payload data
2826 or if the RDE Device has already received payload data, the RDE Device shall consider itself triggered
2827 and begin execution of the Operation.

If triggered, the RDE Device shall respond with results if it is able to complete the Operation within the time period required for a response to this message. If there is a response payload that fits within the ResponsePayload field while maintaining a message size compatible with the negotiated maximum chunk size (see clause 11.2), the RDE Device shall include it within this response. Only if including a response payload would cause the message to exceed the negotiated chunk size may the RDE Device flag it for transfer via RDEMultipartReceive.

2834 The size of the request message is limited to the negotiated maximum chunk size (see clause 11.2). If the
2835 client supplied sufficiently many custom request headers and/or ETags that the request message would
2836 exceed this negotiated size, the MC shall abort the request and perform the following steps:

- 2837 1) Use the RDEOperationKill (see clause 12.6) and then RDEOperationComplete (see clause
2838 12.4) commands to abort and finalize the Operation if it had already been initiated via
2839 RDEOperationInit (see clause 12.1).

2840 2) Return to the client HTTP/HTTPS error code 431, Request Header Fields Too Large.

2841 3) Cease processing of the client request.

When the RDE Device receives a request with data formatted per the Request Data section below, it shall respond with data formatted per the Response Data section. Even with a non-SUCCESS CompletionCode, all fields of the Response Data shall be returned.

Table 64 – SupplyCustomRequestParameters command format

Type	Request data
uint32	<p>ResourceID The resourceID of a resource in the Redfish Resource PDR for the instance to which custom headers should be supplied</p>
rdeOpID	<p>OperationID Identification number for this Operation; must match the one used for all commands relating to this Operation.</p>
uint16	<p>LinkExpand The value of a \$levels qualifier to a \$expand query option if supplied as part of an HTTP/HTTPS GET operation. The MC shall supply a value of zero if the query option was not supplied. This integer indicates the number of levels of links to expand when reading data from a resource. The MC shall supply a value of zero if the \$expand query option was not supplied. See DSP0266 for more details.</p> <p>This value should be ignored by the RDE Device if it did not set expand_support in the DeviceCapabilitiesFlags response parameter to the NegotiateRedfishParameters command.</p> <p>To process the LinkExpand parameters, the MC and RDE Device shall behave as described in clause 7.2.4.3.3. In particular, when supporting this command, an RDE Device shall encode pages expanded into with the <code>bejResourceLinkExpansion</code> format specification.</p>

Type	Request data (continued)
uint16	<p>CollectionSkip</p> <p>The value of a \$skip query option if supplied as part of an HTTP/HTTPS GET operation. The MC shall supply a value of zero if the \$skip query option was not supplied. This integer indicates the number of Members in a resource collection to skip before retrieving the first resource. See DSP0266 for more details.</p> <p>Additional notes on processing the \$skip query option may be found in clause 0.</p>
uint16	<p>CollectionTop</p> <p>The value of a \$top query option if supplied as part of an HTTP/HTTPS GET operation. The MC shall supply a value of 0xFFFF (to be treated by the RDE Device as unlimited) if the query option was not supplied. This indicates the number of Members of a resource collection to include in a response. See DSP0266 for more details.</p> <p>Additional notes on processing the \$top query option may be found in clause 7.2.4.3.2.</p>
uint16	<p>PaginationOffset</p> <p>The page offset for paginated response data that the RDE Device supplied in conjunction with an @odata.nextlink annotation and decoded from a pagination URI. Shall be 0 if no pagination has taken place. See clause 14.2.8 for more details on RDE Device-selected dynamic pagination.</p> <p>Additional notes on pagination may be found in clause 14.2.8.</p>
enum8	<p>ETagOperation</p> <p>To process an ETagOperation, the RDE Device shall respond as described in clauses 7.2.4.2.1 and 7.2.4.2.2.</p> <p>values: { ETAG_IGNORE = 0; ETAG_IF_MATCH = 1; ETAG_IF_NONE_MATCH = 2 }</p>
uint8	<p>ETagCount</p> <p>Number of ETags supplied in this message; should be zero if ETagOperation above is ETAG_IGNORE and nonzero otherwise.</p>
varstring	<p>ETag [0]</p> <p>String data for first ETag, if ETagCount > 0. This string shall be UTF-8 format.</p> <p>Additional notes on processing ETags may be found in clause 7.2.4.2.4.</p>
...	Additional ETags
uint8	<p>HeaderCount</p> <p>The number of RDE custom headers being supplied in this operation.</p> <p>Additional notes on processing RDE custom headers may be found in clause 7.2.4.2.3.</p>
varstring	<p>HeaderName [0]</p> <p>The name of the header, including the PLDM-RDE- prefix</p>
varstring	<p>HeaderParameter [0]</p> <p>The parameter or parameters associated with the header. The MC may preprocess these – though any such preprocessing is outside the scope of this specification – or convey them exactly as received.</p>
...	...
Type	Response data
enum8	<p>CompletionCode</p> <p>value: { PLDM_BASE_CODES, ERROR_OPERATION_ABANDONED, ERROR_OPERATION_FAILED, ERROR_UNSUPPORTED, ERROR_UNEXPECTED, ERROR_UNRECOGNIZED_CUSTOM_HEADER, ERROR_ETAG_MATCH, ERROR_NO_SUCH_RESOURCE }</p> <p>Response codes ERROR_UNSUPPORTED and ERROR_UNRECOGNIZED_CUSTOM_HEADER shall be used to indicate that an unsupported request parameter was sent. These responses represent an Operational failure, not a command failure.</p>

Type	Response data (continued)
enum8	<p>OperationStatus</p> <p>values: { OPERATION_INACTIVE = 0; OPERATION_NEEDS_INPUT = 1; OPERATION_TRIGGERED= 2; OPERATION_RUNNING = 3; OPERATION_HAVE_RESULTS = 4; OPERATION_COMPLETED = 5, OPERATION_FAILED = 6, OPERATION_ABANDONED = 7 }</p>
uint8	<p>CompletionPercentage</p> <p>0..100: percentage complete; 101-253: reserved for future use; 254: not supported or otherwise unable to estimate (but a valid Operation) 255: invalid Operation</p> <p>This value shall be zero if the Operation has not yet been triggered or if the Operation has failed.</p>
uint32	<p>CompletionTimeSeconds</p> <p>An estimate of the number of seconds remaining before the Operation is completed, or 0xFFFF FFFF if such an estimate cannot be provided.</p> <p>This value shall be 0xFFFF FFFF if the Operation has not yet been triggered or if the Operation has failed.</p>
bitfield8	<p>OperationExecutionFlags</p> <p>[7:4] - Reserved</p> <p>[3] - CacheAllowed – 1b = yes; shall be 0b for Operations other than read, head. Shall be 0b unless Operation has finished. Referring to RFC 7234, a value of yes shall be considered as equivalent to Cache-Control response header value “public” and a value of no shall be considered as equivalent to Cache-Control response header value “no-store”. Other cache directives are not supported. The decision of whether to allow caching of data is up to the RDE Device. Typically, static data is allowed to be cached unless, for example, it represents sensitive data such as login credentials; data that changes over time is generally not marked as cacheable</p> <p>To process the CacheAllowed flag, the MC shall behave as described in clause 7.2.4.2.7</p> <p>[2] - HaveResultPayload – 1b = yes. Shall be 0b if Operation has not finished</p> <p>[1] - HaveCustomResponseParameters – 1b = yes. Shall be 0b if Operation has not finished</p> <p>[0] - TaskSpawned – 1b = yes</p> <p>For a failed Operation, this field shall be 0b for all flags other than HaveResultPayload, which may be 1b if a @Message.ExtendedInfo annotation is available to explain the result.</p>
uint32	<p>ResultTransferHandle</p> <p>A data transfer handle that the MC may use to retrieve a larger response payload via one or more RDEMultipartReceive commands (see clause 13.2). The RDE Device shall return a transfer handle of 0xFFFFFFFF if Operation execution has not finished or if the Operation has not yet been triggered. In the event of a failed Operation, or if the data fits entirely within the payload of this command response, or if there is no data to retrieve, the RDE Device shall return a null transfer handle, 0x00000000.</p>

Type	Response data (continued)
bitfield8	<p>PermissionFlags</p> <p>Indicates the access level (types of Operations; see Table 33) granted to the resource targeted by the Operation.</p> <ul style="list-style-type: none"> [7:6] - reserved for future use [5] - head access; 1b = access allowed [4] - delete access; 1b = access allowed [3] - create access; 1b = access allowed [2] - replace access; 1b = access allowed [1] - update access; 1b = access allowed [0] - read access; 1b = access allowed <p>The MC and RDE Device shall process PermissionFlags as described in clause 7.2.4.2.8. NOTE: The bit mapping for the PermissionFlags field was changed in version 1.0.1 of this specification to match that from the RDEOperationInit command, thereby making the entire response message identical for both of these commands.</p>
uint32	<p>ResponsePayloadLength</p> <p>Length in bytes of the response payload in this message. This value shall be zero under any of the following conditions:</p> <ul style="list-style-type: none"> • The Operation has not yet been triggered • The Operation status is not completed or failed, as indicated by the OperationStatus parameter above. For a failed Operation, a @Message.ExtendedInfo annotation may be supplied in the response payload. • There is no response payload as indicated by Bit 2 of the OperationExecutionFlags parameter above • The entire payload cannot fit within this message, subject to the maximum transfer chunk size as determined at registration time via the NegotiateMediumParameters command
varstring	<p>ETag</p> <p>String data for an ETag digest of the target resource; the string text format shall be UTF-8. The ETag may be skipped (an empty string returned in this field) for any of the following actions: Action, Delete, Replace, and Update. The ETag shall also be skipped (an empty string returned in this field) if execution of the Operation has not yet finished.</p> <p>This field supports the ETag Response header. Additional notes on processing ETags may be found in clause 7.2.4.2.4.</p> <p>NOTE ETags provided via this field are not escaped for inclusion in JSON data as they are primarily intended to be used for the ETag HTML header. MCs should be aware that performing a raw comparison of an ETag retrieved from this command with one received as part of BEJ-encoded JSON data will result in a mismatch as the ETag format requires characters that must be escaped in JSON data.</p>
null or bejEncoding	<p>ResponsePayload</p> <p>The response payload. The format of this parameter shall be null (consisting of zero bytes) if the ResponsePayloadLength above is zero; it shall be bejEncoding otherwise.</p>

2846 12.3 RetrieveCustomResponseParameters command (0x12) format

2847 This command enables the MC to retrieve custom HTTP/HTTPS headers or other uncommon response
 2848 parameters from an RDE Device to be forwarded to the client that initiated a Redfish operation. The MC
 2849 shall only invoke this command when the **HaveCustomResponseParameters** flag in the response
 2850 message for a triggered RDE command indicates that it is needed.

2851 The RDE Device shall not supply more response headers than would allow the response message to fit in
 2852 the negotiated maximum transfer chunk size (see clause 11.2).

2853 When the RDE Device receives a request with data formatted per the Request Data section below, it shall
 2854 respond with data formatted per the Response Data section. For a non-SUCCESS CompletionCode, only
 2855 the CompletionCode field of the Response Data shall be returned.

2856

Table 65 – RetrieveCustomResponseParameters command format

Type	Request data
uint32	ResourceID The resourceId of a resource in the Redfish Resource PDR for the instance from which custom headers should be reported
rdeOpID	OperationID Identification number for this Operation; must match the one used for all commands relating to this Operation
Type	Response data
enum8	CompletionCode value: { PLDM_BASE_CODES, ERROR_OPERATION_ABANDONED, ERROR_OPERATION_FAILED, ERROR_UNEXPECTED, ERROR_NO_SUCH_RESOURCE }
uint32	DeferralTimeframe The expected length of time in seconds before the RDE Device will be able to respond to a request to start an Operation, or 0xFF if unknown. The MC shall ignore this field except when the completion code of the previous RDEOperationInit was ERROR_NOT_READY. This field supports the Retry-After response header. Additional notes on processing the Retry-After response header may be found in clause 7.2.4.2.9.
uint32	NewResourceId Resource ID for a newly created collection entry; this value shall be 0 and ignored if the Operation is not a Redfish Create or if the Operation has failed or not yet completed. This field supports the Location Response header. Additional notes on processing the Location response header may be found in clause 7.2.4.2.6.
uint8	ResponseHeaderCount Number of custom response headers contained in the remainder of this message
varstring	HeaderName [0] The name of the header, including the X- prefix This field shall be omitted if ResponseHeaderCount above is zero
varstring	HeaderParameter [0] The parameter or parameters associated with the header. The MC may preprocess these – though any such preprocessing is outside the scope of this specification – or convey them exactly as received This field shall be omitted if ResponseHeaderCount above is zero
...	...

2857 12.4 RDEOperationComplete command (0x13) format

2858 This command enables the MC to inform an RDE Device that it considers an Operation to be complete,
 2859 including failed and abandoned Operations. The RDE Device in turn may discard any internal records for
 2860 the Operation.

2861 When the RDE Device receives a request with data formatted per the Request Data section below, it shall
 2862 respond with data formatted per the Response Data section.

2863

Table 66 – RDEOperationComplete command format

Type	Request data
uint32	ResourceID The resourceID of a resource in the Redfish Resource PDR to which the Task's operation was targeted
rdeOpID	OperationID Identification number for this Operation; must match the one used for all commands relating to this Operation
Type	Response data
enum8	CompletionCode value: { PLDM_BASE_CODES, ERROR_UNEXPECTED, ERROR_NO_SUCH_RESOURCE }

2864 12.5 RDEOperationStatus command (0x14) format

2865 This command enables the MC to query an RDE Device for the status of an Operation. It is additionally
 2866 used to collect the initial response when an RDE Operation is triggered by a RDEMultipartSend command
 2867 or after a Task finishes asynchronous execution.

2868 When providing result data for an Operation that has finished executing, if there is a response payload
 2869 that fits within the ResponsePayload field while maintaining a message size compatible with the
 2870 negotiated maximum chunk size (see NegotiateMediumParameters, clause 11.2), the RDE Device shall
 2871 include it within this response. Only if including a response payload would cause the message to exceed
 2872 the negotiated chunk size may the RDE Device flag it for transfer via RDEMultipartReceive.

2873 When the RDE Device receives a request with data formatted per the Request Data section below, it shall
 2874 respond with data formatted per the Response Data section. Even with a non-SUCCESS
 2875 CompletionCode, all fields of the Response Data shall be returned.

2876 **Table 67 – RDEOperationStatus command format**

Type	Request data
uint32	ResourceID The resourceID of a resource in the Redfish Resource PDR to which the Task's operation was targeted
rdeOpID	OperationID Identification number for this Operation; must match the one used for all commands relating to this Operation
Type	Response data

Type	Response data
enum8	<p>CompletionCode</p> <p>value: { PLDM_BASE_CODES, ERROR_UNSUPPORTED, ERROR_ETAG_MATCH, ERROR_UNRECOGNIZED_CUSTOM_HEADER }</p> <p>The completion code for RDEOperationStatus shall be one of the following:</p> <p>SUCCESS: An RDE Operation was referenced in the OperationID request field and it is not in the failed state. The actual current status of the RDE Operation is returned in the OperationStatus field. If the OperationID does not correspond to an active Operation, the state shall be reported as OPERATION_INACTIVE.</p> <p>ERROR_UNSUPPORTED, ERROR_ETAG_MATCH, ERROR_UNRECOGNIZED_CUSTOM_HEADER: An RDE Operation in the FAILED state was referenced in the OperationID request field, and the Operation failed with the specified status code. OperationStatus shall be OPERATION_FAILED in this case. These responses indicate a failure in the RDE Operation, not a failure in the RDEOperationStatus command.</p>
enum8	<p>OperationStatus</p> <p>values: { OPERATION_INACTIVE = 0; OPERATION_NEEDS_INPUT = 1; OPERATION_TRIGGERED = 2; OPERATION_RUNNING = 3; OPERATION_HAVE_RESULTS = 4; OPERATION_COMPLETED = 5, OPERATION_FAILED = 6, OPERATION_ABANDONED = 7 }</p>
uint8	<p>CompletionPercentage</p> <p>0..100: percentage complete; 101-253: reserved for future use; 254: not supported or otherwise unable to estimate (but a valid Operation) 255: invalid Operation</p> <p>This value shall be zero if the Operation has not yet been triggered or if the Operation has failed.</p>
uint32	<p>CompletionTimeSeconds</p> <p>An estimate of the number of seconds remaining before the Operation is completed, or 0xFFFF FFFF if such an estimate cannot be provided.</p> <p>This value shall be 0xFFFF FFFF if the Operation has not yet been triggered or if the Operation has failed.</p>
bitfield8	<p>OperationExecutionFlags</p> <ul style="list-style-type: none"> [7:4] - Reserved [3] - CacheAllowed – 1b = yes; shall be 0b for Operations other than read, head. Shall be 0b unless Operation has finished. Referring to RFC 7234, a value of yes shall be considered as equivalent to Cache-Control response header value “public” and a value of no shall be considered as equivalent to Cache-Control response header value “no-store”. Other cache directives are not supported. The decision of whether to allow caching of data is up to the RDE Device. Typically, static data is allowed to be cached unless, for example, it represents sensitive data such as login credentials; data that changes over time is generally not marked as cacheable To process the CacheAllowed flag, the MC shall behave as described in clause 7.2.4.2.7 [2] - HaveResultPayload – 1b = yes. Shall be 0b if Operation has not finished [1] - HaveCustomResponseParameters – 1b = yes. Shall be 0b if Operation has not finished [0] - TaskSpawned – 1b = yes <p>For a failed Operation, this field shall be 0b for all flags other than HaveResultPayload, which may be 1b if a @Message.ExtendedInfo annotation is available to explain the result.</p>

Type	Response data (continued)
uint32	<p>ResultTransferHandle</p> <p>A data transfer handle that the MC may use to retrieve a larger response payload via one or more RDEMultipartReceive commands (see clause 13.2). The RDE Device shall return a transfer handle of 0xFFFFFFFF if Operation execution has not finished or if the Operation has not yet been triggered. In the event of a failed Operation, or if the data fits entirely within the payload of this command response, or if there is no data to retrieve, the RDE Device shall return a null transfer handle, 0x00000000.</p> <p>In the event that data transfer for this Operation is currently in progress (at least one chunk has been transferred but the final chunk has not yet been transferred, and a timeout has not occurred awaiting the request for the next chunk), the RDE Device shall return the transfer handle that was most recently returned in the response message for a RDEMultipartSend or RDEMultipartReceive command.</p>
bitfield8	<p>PermissionFlags</p> <p>Indicates the access level (types of Operations; see Table 33) granted to the resource targeted by the Operation.</p> <ul style="list-style-type: none"> [7:6] - reserved for future use [5] - head access; 1b = access allowed [4] - delete access; 1b = access allowed [3] - create access; 1b = access allowed [2] - replace access; 1b = access allowed [1] - update access; 1b = access allowed [0] - read access; 1b = access allowed <p>This field supports the Allow header. Additional notes on processing the Allow header may be found in clause 7.2.4.2.8</p> <p>.NOTE: The bit mapping for the PermissionFlags field was changed in version 1.0.1 of this specification to match that from the RDEOperationInit command, thereby making the entire response message identical for both of these commands.</p>
uint32	<p>ResponsePayloadLength</p> <p>Length in bytes of the response payload in this message. This value shall be zero under any of the following conditions:</p> <ul style="list-style-type: none"> • The Operation has not yet been triggered • The Operation status is not completed or failed, as indicated by the OperationStatus parameter above. For a failed Operation, a @Message.ExtendedInfo annotation may be supplied in the response payload. • There is no response payload as indicated by Bit 2 of the OperationExecutionFlags parameter above • The entire payload cannot fit within this message, subject to the maximum transfer chunk size as determined at registration time via the NegotiateMediumParameters command

Type	Response data (continued)
varstring	<p>ETag String data for an ETag digest of the target resource; the string text format shall be UTF-8. The ETag may be skipped (an empty string returned in this field) for any of the following actions: Action, Delete, Replace, and Update. The ETag shall also be skipped (an empty string returned in this field) if execution of the Operation has not yet finished.</p> <p>Additional notes on processing ETags may be found in clause 7.2.4.2.4.</p> <p>NOTE ETags provided via this field are not escaped for inclusion in JSON data as they are primarily intended to be used for the ETag HTML header. MCs should be aware that performing a raw comparison of an ETag retrieved from this command with one received as part of BEJ-encoded JSON data will result in a mismatch as the ETag format requires characters that must be escaped in JSON data.</p>
null or bejEncoding	<p>ResponsePayload The response payload. The format of this parameter shall be null (consisting of zero bytes) if the ResponsePayloadLength above is zero; it shall be bejEncoding otherwise.</p>

2878 12.6 RDEOperationKill command (0x15) format

2879 This command enables the MC to request that an RDE Device terminate an Operation. The RDE Device
 2880 shall kill the Operation if the Operation can be killed; however, the MC must be aware that not all
 2881 Operations can be terminated.

2882 When the RDE Device receives a request with data formatted per the Request Data section below, it shall
 2883 respond with data formatted per the Response Data section if it supports the command.

2884

Table 68 – RDEOperationKill command format

Type	Request data
uint32	<p>ResourceID The resourceID of a resource in the Redfish Resource PDR to which the Task's operation was targeted</p>
rdeOpID	<p>OperationID Identification number for this Operation; must match the one used for all commands relating to this Operation</p>

Type	Request data (continued)
bitfield8	<p>KillFlags</p> <p>Flags for killing the Operation:</p> <ul style="list-style-type: none"> [7:3] - reserved for future use [2] - discard_results; if 1b and the RDE Device is in the HAVE_RESULTS state for this Operation, the results of the Operation shall be discarded and the Operation state set to Inactive. The MC shall not set the discard_results bit in conjunction with any other bits in the KillFlags. In the event that the MC violates this restriction, the RDE Device shall respond with completion code ERROR_INVALID_DATA and stop processing the request. [1] - run_to_completion; if 1b, the Operation should be run to completion but no further response should be sent to the MC. The MC shall not set the run_to_completion bit without also setting the discard_record bit. In the event that the MC violates this restriction, the RDE Device shall respond with completion code ERROR_INVALID_DATA and stop processing the request. [0] - discard_record; if 1b and the kill command returns success, the RDE Device shall discard internal records associated with this Operation as soon as it is killed; the RDE Device should not expect the MC to call RedfishOperationComplete for this Operation. If the Operation has spawned a Task, the RDE Device shall not create an Event when execution is finished.
Type	Response data
enum8	<p>CompletionCode</p> <p>value: { PLDM_BASE_CODES, ERROR_OPERATION_ABANDONED, ERROR_OPERATION_FAILED, ERROR_OPERATION_UNKILLABLE, ERROR_NO_SUCH_RESOURCE, ERROR_UNEXPECTED }</p>

2885 12.7 RDEOperationEnumerate command (0x16) format

2886 This command enables the MC to request that an RDE Device enumerate all Operations that are
 2887 currently active (not in state INACTIVE in the Operation lifecycle state machine of clause 9.2.3.2). It is
 2888 expected that the MC will typically use this command during its initialization to discover any Operations
 2889 that spawned Tasks that were active through a shutdown.

2890 NOTE When instantiating Operations, the RDE Device shall not create a new Operation if including the total data
 2891 for all Operations would cause the response message for this command to exceed the negotiated maximum
 2892 transfer chunk size (see clause 11.2) for any of the mediums on which the MC has communicated with the
 2893 RDE Device.

2894 If the RDE Device accepts operations from protocols other than Redfish, it should make them visible as
 2895 RDE Operations while they are active by enumerating them in response to this command.

2896 When the RDE Device receives a request with data formatted per the Request Data section below, it shall
 2897 respond with data formatted per the Response Data section if it supports the command. For a non-
 2898 SUCCESS CompletionCode, only the CompletionCode field of the Response Data shall be returned.

2899

Table 69 – RDEOperationEnumerate command format

Type	Request data
n/a	This request contains no parameters
Type	Response data
enum8	CompletionCode value: { PLDM_BASE_CODES }
uint16	OperationCount The number of active Operations N described in the remainder of this message
uint32	ResourceId [0] The resource ID of the Redfish Resource PDR to which the Operation was targeted. Shall be omitted if OperationCount is zero
rdeOpID	OperationID [0] Operation identifier assigned for the Operation when the MC initialized the Operation via the RDEOperationInit command or when the RDE Device chose to make an external Operation visible via RDE. This field shall be omitted if OperationCount above is zero
enum8	OperationType [0] The type of Operation. Shall be omitted if OperationCount is zero values: { OPERATION_HEAD = 0; OPERATION_READ = 1; OPERATION_CREATE = 2; OPERATION_DELETE = 3; OPERATION_UPDATE = 4; OPERATION_REPLACE = 5; OPERATION_ACTION = 6 } This field shall be omitted if OperationCount above is zero
...	...
uint32	ResourceId [N - 1] The resource ID of the Redfish Resource PDR to which the Operation was targeted
rdeOpID	OperationID [N - 1] Operation identifier assigned for the Operation when the MC initialized the Operation via the RDEOperationInit command or when the RDE Device chose to make an external Operation visible via RDE
enum8	OperationType [N - 1] The type of Operation values: { OPERATION_HEAD = 0; OPERATION_READ = 1; OPERATION_CREATE = 2; OPERATION_DELETE = 3; OPERATION_UPDATE = 4; OPERATION_REPLACE = 5; OPERATION_ACTION = 6 }

2900

13 PLDM for Redfish Device Enablement – Utility commands

2901

13.1 RDEMultipartSend command (0x30) format

2902

This command enables the MC to send a large volume of data to an RDE Device. In the event of a data checksum error, the MC may reissue the first RDEMultipartSend command with the initial data transfer handle; the RDE Device shall recognize this to mean that the transfer failed and respond as if this were the first transfer attempt. If the MC chooses not to restart the transfer, or in any other error occurs, the MC should abandon the transfer. In the latter case, if the transfer is part of an Operation, the MC shall explicitly abort and then finalize the Operation via the RDEOperationKill and RDEOperationComplete commands (see clauses 12.6 and 12.4).

2909 Similarly, in the event of transient transfer errors for individual chunks of the data, the MC may retry those
 2910 chunks by reissuing the RDEMultipartSend command corresponding to those chunks provided it has not
 2911 yet issued a RDEMultipartSend command for a subsequent chunk. When the RDE Device receives a
 2912 request with data formatted per the Request Data section below, it shall respond with data formatted per
 2913 the Response Data section. For a non-SUCCESS CompletionCode with the exception of
 2914 ERROR_BAD_CHECKSUM, only the CompletionCode field of the Response Data shall be returned. In
 2915 the case an ERROR_BAD_CHECKSUM is returned, the RDE Device may set the TransferOperation to
 2916 XFER_FIRST_PART.

2917 NOTE In versions of this specification prior to v1.1.0, this command was named MultipartSend.

2918

Table 70 – RDEMultipartSend command format

Type	Request data
uint32	<p>DataTransferHandle A handle to uniquely identify the chunk of data to be sent. If TransferFlag below is START or START_AND_END, this must match the SendDataTransferHandle that was supplied by the RDE Device in the response to RDEOperationInit. The DataTransferHandle supplied shall be either the initial handle to begin or restart a transfer or the NextDataTransferHandle as specified in the previous chunk.</p>
rdeOpID	<p>OperationID Identification number for this Operation; must match the one previously used for all commands relating to this Operation; 0x0000 if this transfer is not part of an Operation.</p>
enum8	<p>TransferFlag An indication of current progress within the transfer. The value START_AND_END indicates that the entire transfer consists of a single chunk. value: { START = 0, MIDDLE = 1, END = 2, START_AND_END = 3 }</p>
uint32	<p>NextDataTransferHandle The handle for the next chunk of data for this transfer; zero (0x00000000) if no further data.</p>
uint32	<p>DataLengthBytes The length in bytes N of data being sent in this chunk, including both the Data and DataIntegrityChecksum (if present) fields. This value and the data bytes associated with it shall not cause this request message to exceed the negotiated maximum transfer chunk size (clause 11.2).</p>
uint8	<p>Data [0] The first byte of the current chunk of data. Shall be omitted if only the DataIntegrityChecksum is present.</p>
...	...
uint8	<p>Data [N-1] The last byte of the current chunk of data. Shall be omitted if only the DataIntegrityChecksum is present.</p>

Type	Request data (continued)
uint32	<p>DataIntegrityChecksum</p> <p>32-bit CRC for the entirety of data (all parts concatenated together, excluding this checksum). Shall be omitted for non-final chunks (TransferFlag ≠ END or START_AND_END) in the transfer. The DataIntegrityChecksum shall not be split across multiple chunks. If appending the DataIntegrityChecksum would cause this request message to exceed the negotiated maximum transfer chunk size (clause 11.2), the DataIntegrityChecksum shall be sent as the only data in another chunk.</p> <p>For this specification, the CRC-32 algorithm with the polynomial $x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$ (same as the one used by IEEE 802.3) shall be used for the integrity checksum computation. The CRC computation involves processing a byte at a time with the least significant bit first.</p>
Type	Response data
enum8	<p>CompletionCode</p> <p>value: { PLDM_BASE_CODES, ERROR_OPERATION_ABANDONED, ERROR_OPERATION_FAILED, ERROR_UNEXPECTED, ERROR_BAD_CHECKSUM }</p> <p>If the DataTransferHandle does not correspond to a valid chunk, the RDE Device shall return CompletionCode ERROR_INVALID_DATA.</p>
enum8	<p>TransferOperation</p> <p>The follow-up action that the RDE Device is requesting of the MC:</p> <ul style="list-style-type: none"> • XFER_FIRST_PART: resend the initial chunk (restarting the transmission, such as if the checksum of data received did not match the DataIntegrityChecksum in the final chunk) • XFER_NEXT_PART: send the next chunk of data • XFER_ABORT: stop the transmission and do not retry. The MC shall proceed as if the transmission is permanently failed in this case • XFER_COMPLETE: no further follow-up needed, the transmission completed normally <p>value: { XFER_FIRST_PART = 0, XFER_NEXT_PART = 1, XFER_ABORT = 2, XFER_COMPLETE = 3 }</p>

2919 13.2 RDEMultipartReceive command (0x31) format

2920 This command enables the MC to receive a large volume of data from an RDE Device. In the event of a
 2921 data checksum error, the MC may reissue the first RDEMultipartReceive command with the initial data
 2922 transfer handle; the RDE Device shall recognize this to mean that the transfer failed and respond as if this
 2923 were the first transfer attempt. If the MC chooses not to restart the transfer, or in any other error occurs,
 2924 the MC should abandon the transfer. In the latter case, if the transfer is part of an Operation, the MC shall
 2925 explicitly abort and finalize the Operation via the RDEOperationKill and then RDEOperationComplete
 2926 commands (see clauses 12.6 and 12.4).

2927 Similarly, in the event of transient transfer errors for individual chunks of the data, the MC may retry those
 2928 chunks by reissuing the RDEMultipartReceive command corresponding to those chunks provided it has
 2929 not yet issued a RDEMultipartReceive command for a subsequent chunk.

2930 When the RDE Device receives a request with data formatted per the Request Data section below, it shall
 2931 respond with data formatted per the Response Data section if it supports the command. For a non-
 2932 SUCCESS CompletionCode, only the CompletionCode field of the Response Data shall be returned.

2933 NOTE In versions of this specification prior to v1.1.0, this command was named MultipartReceive.

Table 71 – RDEMultipartReceive command format

Type	Request data
uint32	<p>DataTransferHandle A handle to uniquely identify the chunk of data to be retrieved. If TransferOperation below is XFER_FIRST_PART and the OperationID below is zero, this must match the TransferHandle supplied by the RDE Device in the response to the GetSchemaDictionary, GetMessageRegistry, or GetSchemaFile command. If TransferOperation below is XFER_FIRST_PART and the OperationID below is nonzero, this must match the SendDataTransferHandle that was supplied by the RDE Device in the response to RDEOperationInit. If TransferOperation below is XFER_NEXT_PART, this must match the NextDataHandle supplied by the RDE Device with the previous chunk.</p> <p>The DataTransferHandle supplied shall be either the initial handle to begin or restart a transfer or the NextDataTransferHandle supplied with the previous chunk.</p>
rdeOpID	<p>OperationID Identification number for this Operation; must match the one previously used for all commands relating to this Operation; 0x0000 if this transfer is not part of an Operation</p>
enum8	<p>TransferOperation The portion of data requested for the transfer:</p> <ul style="list-style-type: none"> • XFER_FIRST_PART: The MC is asking the transfer to begin or to restart from the beginning • XFER_NEXT_PART: The MC is asking for the next portion of the transfer • XFER_ABORT: The MC is requesting that the transfer be discarded. The RDE Device may discard any internal data structures it is maintaining for the transfer <p>value: { XFER_FIRST_PART = 0, XFER_NEXT_PART = 1, XFER_ABORT = 2 }</p>
Type	Response data
enum8	<p>CompletionCode value: { PLDM_BASE_CODES, ERROR_OPERATION_ABANDONED, ERROR_OPERATION_FAILED, ERROR_UNEXPECTED, ERROR_BAD_CHECKSUM } If the DataTransferHandle does not correspond to a valid chunk, the RDE Device shall return CompletionCode ERROR_INVALID_DATA. If the transfer is aborted, the RDE Device shall acknowledge this status by returning SUCCESS.</p>
enum8	<p>TransferFlag value: { START = 0, MIDDLE = 1, END = 2, START_AND_END = 3 } This field shall be omitted for a non-SUCCESS CompletionCode or if the transfer has been aborted</p>
uint32	<p>NextDataTransferHandle The handle for the next chunk of data for this transfer; zero (0x00000000) if no further data This field shall be omitted for a non-SUCCESS CompletionCode or if the transfer has been aborted</p>
uint32	<p>DataLengthBytes The length in bytes N of data being sent in this chunk, including both the Data and DataIntegrityChecksum (if present) fields. This value and the data bytes associated with it shall not cause this response message to exceed the negotiated maximum transfer chunk size (clause 11.2). This field shall be omitted for a non-SUCCESS CompletionCode or if the transfer has been aborted</p>
uint8	<p>Data [0] The first byte of current chunk of data. Shall be omitted if only the DataIntegrityChecksum is present. This field shall be omitted for a non-SUCCESS CompletionCode or if the transfer has been aborted</p>
...	...

Type	Response data (continued)
uint8	<p>Data [N-1] The last byte of the current chunk of data. Shall be omitted if only the DataIntegrityChecksum is present. This field shall be omitted for a non-SUCCESS CompletionCode or if the transfer has been aborted</p>
uint32	<p>DataIntegrityChecksum 32-bit CRC for the entire block of data (all parts concatenated together, excluding this checksum). Shall be omitted for non-final chunks (TransferFlag ≠ END or START_AND_END) in the transfer or for aborted transfers. The DataIntegrityChecksum shall not be split across multiple chunks. If appending the DataIntegrityChecksum would cause this response message to exceed the negotiated maximum transfer chunk size (clause 11.2), the DataIntegrityChecksum shall be sent as the only data in another chunk. For this specification, the CRC-32 algorithm with the polynomial $x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$ (same as the one used by IEEE 802.3) shall be used for the integrity checksum computation. The CRC computation involves processing a byte at a time with the least significant bit first.</p>

2935 14 Additional Information

2936 14.1 RDE Multipart transfers

2937 The various commands defined in clauses 10 and **Error! Reference source not found.** support bulk
 2938 transfers via the RDEMultipartSend and RDEMultipartReceive commands defined in clause 13. The
 2939 RDEMultipartSend and RDEMultipartReceive commands use flags and data transfer handles to perform
 2940 multipart transfers. A data transfer handle uniquely identifies the next part of the transfer. The data
 2941 transfer handle values are implementation specific. For example, an implementation can use memory
 2942 addresses or sequence numbers as data transfer handles.

2943 NOTE If both the RDE Device and the MC support use of PLDM common multipart transfers, those versions of the
 2944 commands shall be used in lieu of the RDE versions. The following notes apply:
 2945 • All transfers shall consist of a single portion, beginning at offset zero and transferring the entire buffer
 2946 • The TransferContext field, which is defined in [DSP0240](#) to be protocol specific, shall be supplied with the
 2947 OperationID that would have been used with an RDE version of a multipart transfer
 2948 • Handling of aborted transfers, which is defined in [DSP0240](#) to be protocol specific, shall follow the notes
 2949 provided within this specification for multipart transfers.

2950 14.1.1 Flag usage for RDEMultipartSend

2951 The following list shows some requirements for using TransferOperationFlag, TransferFlag, and
 2952 DataTransferHandle in RDEMultipartSend data transfers:

- 2953 • To prepare a large data send for use in an RDE command, a DataTransferHandle shall be sent
 2954 by the MC in the request message of the RDEOperationInit command.
- 2955 • To reflect a data transfer (re)initiated with a RDEMultipartSend command, the
 2956 TransferOperation shall be set to XFER_FIRST_PART in the response message.
- 2957 • For transferring a part after the first part of data, the TransferOperation shall be set to
 2958 XFER_NEXT_PART and the DataTransferHandle shall be set to the NextDataTransferHandle
 2959 that was obtained in the request for the previous RDEMultipartSend command for this data
 2960 transfer.
- 2961 • The TransferFlag specified in the request for a RDEMultipartSend command has the following
 2962 meanings:

- 2963 – START, which is the first part of the data transfer
2964 – MIDDLE, which is neither the first nor the last part of the data transfer
2965 – END, which is the last part of the data transfer
2966 – START_AND_END, which is the first and the last part of the data transfer. In this case, the
2967 transfer consists of a single chunk
- 2968 • For a RDEMultipartSend, the requester shall consider a data transfer complete when it receives
2969 a success CompletionCode in the response to a request in which the TransferFlag was set to
2970 End or StartAndEnd.

2971 **14.1.2 Flag usage for RDEMultipartReceive**

2972 The following list shows some requirements for using TransferOperationFlag, TransferFlag, and
2973 DataTransferHandle in RDEMultipartReceive data transfers:

- 2974 • To prepare a large data transfer receive for use in an RDE command, a DataTransferHandle
2975 shall be sent by the RDE Device in the response message to the RDEOperationInit,
2976 SupplyCustomRequestParameters, or RDEOperationStatus command after an Operation has
2977 finished execution and results are ready for pick-up.
- 2978 • To initiate a data transfer with a RDEMultipartReceive command, the TransferOperation shall
2979 be set to XFER_FIRST_PART in the request message.
- 2980 • For transferring a part after the first part of data, the TransferOperation shall be set to
2981 XFER_NEXT_PART and the DataTransferHandle shall be set to the NextDataTransferHandle
2982 that was obtained in the response to the previous RDEMultipartReceive command for this data
2983 transfer.
- 2984 • The TransferFlag specified in the response of a RDEMultipartReceive command has the
2985 following meanings:
- 2986 – START, which is the first part of the data transfer
2987 – MIDDLE, which is neither the first nor the last part of the data transfer
2988 – END, which is the last part of the data transfer
2989 – START_AND_END, which is the first and the last part of the data transfer
- 2990 • For a RDEMultipartReceive, the requester and responder shall consider a data transfer
2991 complete when the TransferFlag in the response is set to END or START_AND_END. After this
2992 point, the transfer may not be restarted without repeating the invoking commands, such as
2993 GetSchemaDictionary for a multipart transfer of a dictionary.

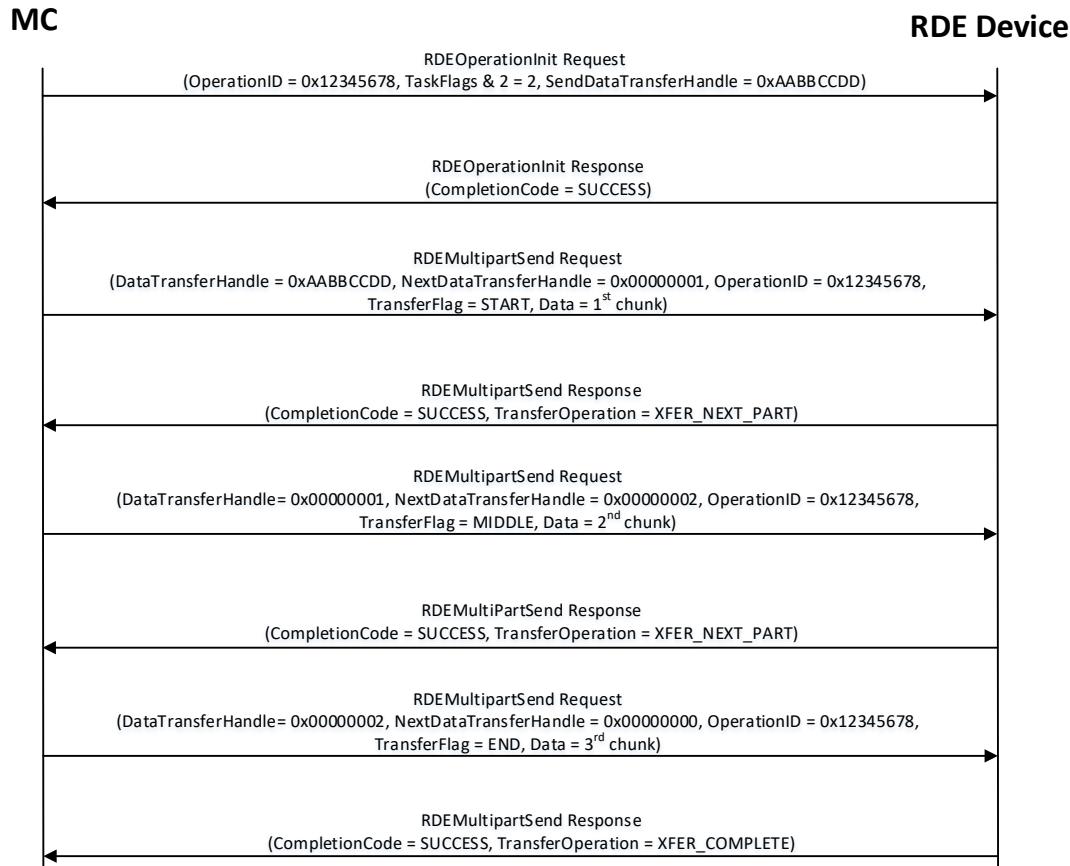
2994 **14.1.3 RDE Multipart transfer examples**

2995 The following examples show how the multipart transfers can be performed using the generic mechanism
2996 defined in the commands.

2997 In the first example, the MC sends data to the RDE Device as part of a Redfish Update operation.
2998 Following the RDEOperationInit command sequence, the MC effects the transfer via a series of
2999 RDEMultipartSend commands. Figure 17 shows the flow of the data transfer.

3000 In the second example, the MC retrieves the dictionary for a schema. The request is initiated via the
3001 GetSchemaDictionary command and then effected via one or more RDEMultipartReceive commands.

3002 Figure 18 shows the flow of the data transfer.



3003

3004

Figure 17 – RDEMultipartSend example

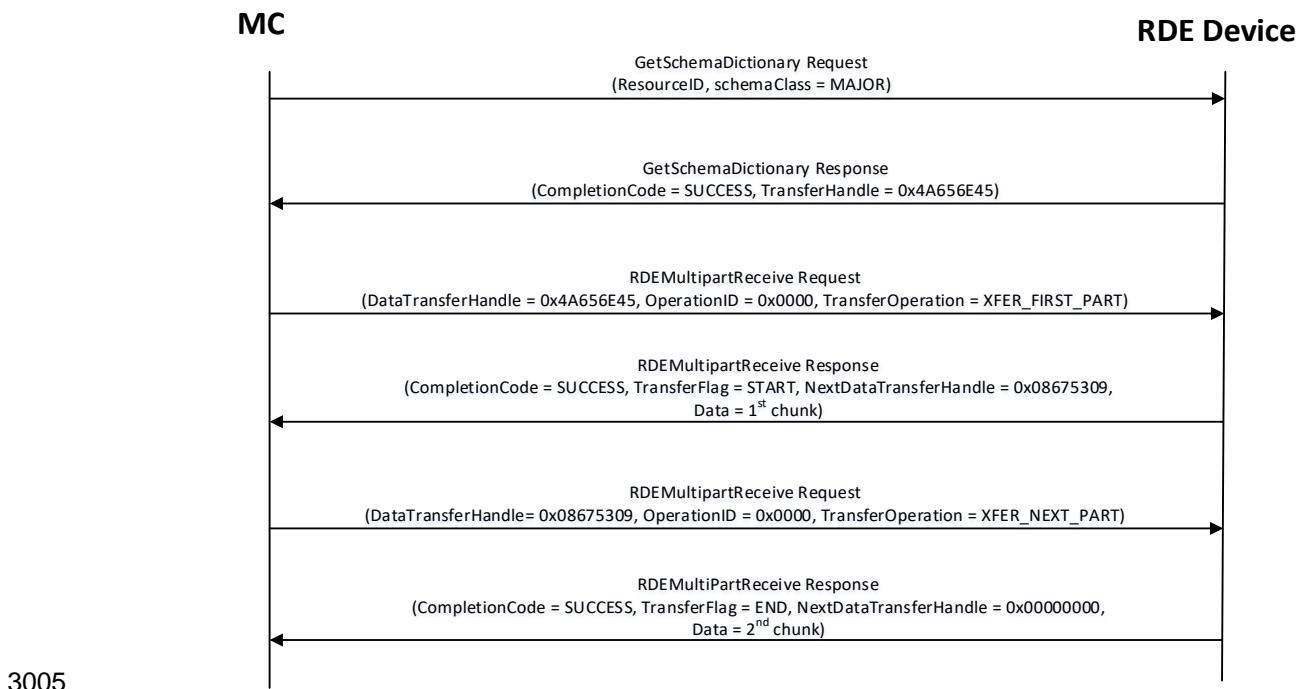


Figure 18 – RDEMultipartReceive example

3007 14.2 Implementation notes

3008 Several implementation notes apply to manufacturers of RDE Devices or of management controllers.

3009 14.2.1 Schema updates

3010 If one or more schemas for an RDE Device are updated, the RDE Device may communicate this to the
 3011 MC by triggering an event for the affected PDRs. When the MC detects a PDR update, it shall reread the
 3012 affected PDRs.

3013 14.2.2 Storage of dictionaries

3014 It is not necessary for the MC to maintain all dictionaries in memory at any given time. It may flush
 3015 dictionaries at will since they can be retrieved on demand from the RDE Devices via the
 3016 GetSchemaDictionary command (clause 11.2). However, if the MC has to retrieve a dictionary “on
 3017 demand” to support a Redfish query, this will likely incur a performance delay in responding to the client.
 3018 For MCs with highly limited memory that cannot retain all the dictionaries they need to support, care must
 3019 thus be exercised in the runtime selection of dictionaries to evict. Such caching considerations are
 3020 outside the scope of this specification.

3021 14.2.3 Dictionaries for related schemas

3022 MCs must not assume that sibling instances of Redfish Resource PDRs in a hierarchy (such as collection
 3023 members) use the same version of a schema. They could, for example, correspond to individual elements
 3024 from an array of hardware (such as a disk array) built by separate manufacturers and supporting different
 3025 versions of a major schema or with different OEM extensions to it. However, at such time as the MC has
 3026 verified that two siblings do in fact use the same schemas, there is no reason to store multiple copies of
 3027 the dictionary corresponding to that schema. Of course, sibling instances of resources stored within the

3028 same PDR share all dictionaries; it is only with instances of resources from separate PDRs that this
3029 applies.

3030 Similarly, it is expected to be fairly commonplace that the system managed by an MC could have multiple
3031 RDE Devices of the same class, such as multiple network adapters or multiple RAID array controllers. In
3032 such cases, however, there is no guarantee that each such RDE Device will support the same version of
3033 any given Redfish schema.

3034 To handle such cases, MCs have two choices. The most straightforward approach is to simply maintain
3035 each dictionary associated with the RDE Device it came from. This of course has space implications. A
3036 more practical approach is to store one copy of the dictionary for each version of the schema and then
3037 keep track of which version of the dictionary to use with which RDE Device. Because RDE Devices may
3038 support only subsets of the properties in resources, care must be taken when employing this approach to
3039 ensure that all supported properties are covered in the dictionaries selected. This may be done by
3040 merging dictionaries at runtime, though details of how to merge dictionaries are out of scope for this
3041 specification. In particular, OEM sections of dictionaries are not generally able to be merged as the
3042 sequence numbers for the names of the different OEM extensions themselves are likely to overlap.

3043 However, an even better approach is available. In Redfish schemas, so long as only the minor and
3044 release version numbers change, schemas are required to be fully backward compatible with earlier
3045 revisions. Individual properties and enumeration values may be added but never removed. The MC can
3046 therefore leverage this to retain only the newest instance of dictionary for each major version supported
3047 by RDE Devices. Again, the fact that RDE Devices may support only subsets of the properties in a
3048 resource means that care must be taken to ensure dictionary support for all the properties used across all
3049 RDE Devices that implement any given schema.

3050 **14.2.4 [MC] HTTP/HTTPS POST Operations**

3051 As specified in [DSP0266](#), a Redfish POST Operation can represent either a Create Operation or an
3052 Action. To distinguish between these cases, the MC may examine the URI target supplied with the
3053 operation. If it points to a collection, the MC may assume that the Operation is a Create; if it points to an
3054 action, the MC may assume the Operation is an Action. Alternatively, the MC may presuppose that the
3055 POST is a Create Operation and if it receives an ERROR_WRONG_LOCATION_TYPE error code from
3056 the RDE Device, retry the Operation as an Action. This second approach reduces the amount of URI
3057 inspection the MC has to perform in order to proxy the Operation at the cost of a small delay in
3058 completion time for the Action case. (The supposition that POSTs correspond to Create Operations could
3059 of course be reversed, but it is expected that Actions will be much rarer than Create Operations.)
3060 Implementers should be aware that such delays could cause a client-side timeout.

3061 Another clue that could be used to differentiate between POSTs intended as create operations vs POSTs
3062 intended as actions would be trial encodings of supplied payload data. If there is no payload data, then
3063 the request is either in error or an action. In this case, the payload should be encoded with the dictionary
3064 for the major schema associated with target resource. On the other hand, if the payload is intended for a
3065 create operation, the correct dictionary to use would be the collection member dictionary, which may be
3066 retrieved via the GetSchemaDictionary command (clause 11.2), specifying
3067 COLLECTION_MEMBER_TYPE as the dictionary to retrieve.

3068 **14.2.4.1 Support for Actions**

3069 When a Redfish client issues a Redfish Operation for an Action, the URI target for the Action will be a
3070 POST of the form /redfish/v1/{path to root of RDE Device component}/{path to RDE Device owned
3071 resource}/Actions/schema_name.action_name. To process this, the MC may translate {path to root of
3072 RDE Device component} and {path to RDE Device owned resource} normally to identify the PDR against
3073 which the Operation should be executed. (If the URI is not in this format, this is another indication that the
3074 POST operation is probably a CREATE.) After it has performed this step, the MC can then check its PDR
3075 hierarchy to find the Redfish Action PDR containing an action named schema_name.action_name. If it

3076 doesn't find one, the MC shall respond with HTTP status code 404, Not Found and stop processing the
3077 Operation.

3078 After the correct Action is located, the MC can translate any request parameters supplied with the Action.
3079 To do so, it should look within the dictionary at the point beginning with the named action, and then
3080 navigate into the Parameters set under the action. From there, standard encoding rules apply. When
3081 supplying a locator for the Action to the RDE Device as part of the RDEOperationInit command, the MC
3082 shall not include the Parameters set as one of the sequence numbers comprising the locator; rather, it
3083 shall stop with the sequence number for the property corresponding to the Action's name.

3084 After the Action is complete, it may contain result parameters. If present, definitions for these will be found
3085 in the dictionary in a ReturnType set parallel to the Parameters set that contained any request
3086 parameters. If an Action does not contain explicit result parameters, the ReturnType set will generally not
3087 be present in the dictionary. The structure of the ReturnType set mirrors exactly that of the Parameters
3088 set.

3089 **14.2.5 Consistency checking of read Operations**

3090 Because the collection of data contained within a schema cannot generally be read atomically by RDE
3091 Devices, issues of consistency arise. In particular, if the RDE Device reads some of the data, performs an
3092 update, and then reads more data, there is no guarantee that data read in the separate "chunks" will be
3093 mutually consistent. While the level of risk that this could pose for a client consumer of the data may vary,
3094 the threat will not. The problem is exacerbated when reads must be performed across multiple resources
3095 in order to satisfy a client request: The window of opportunity for a write to slip in between distinct
3096 resource reads is much larger than the window between reads of individual pieces of data in a single
3097 resource.

3098 To resolve the threat of inconsistency, MCs should utilize a technique known as consistency checking.
3099 Before issuing a read, the MC should retrieve the ETag for the schema to be read, using the
3100 GetResourceETag command (clause 11.5). For a read that spans multiple resources, the global ETag
3101 should be read instead, by supplying 0xFFFFFFFF for the ResourceID in the command. The MC should
3102 then proceed with all of the reads and then check the ETag again. If the ETag matches what was initially
3103 read, the MC may conclude that the read was consistent and return it to the client. Otherwise, the MC
3104 should retry. It is expected that consistency failures will be very rare; however, if after three attempts, the
3105 MC cannot obtain a consistent read, it should report error 500, Internal Server Error to the client.

3106 NOTE For reads that only span a single resource, if the RDE Device asserts the **atomic_resource_read** bit in the
3107 **DeviceCapabilitiesFlags** response message to the NegotiateRedfishParameters command (clause 11.1),
3108 the MC may skip consistency checking.

3109 **14.2.6 [MC] Placement of RDE Device resources in the outward-facing Redfish URI 3110 hierarchy**

3111 In the Redfish Resource PDRs and Redfish Entity Association PDRs that an RDE Device presents, there
3112 will normally be one or a limited number that reflect EXTERNAL (0x0000) as their ContainingResourceID.
3113 These resources need to be integrated into the outward-facing Redfish URI hierarchy. Resources that do
3114 not reflect EXTERNAL as their ContainingResourceID do not need to be placed by the MC; it is the RDE
3115 Device's responsibility to make sure that they are accessible via some chain of Redfish Resource and
3116 Redfish Entity Association PDRs (including PDRs chained via @link properties) that ultimately link to
3117 EXTERNAL.

3118 When retrieving these PDRs for RDE Device components, the MC should read the
3119 ProposedContainingResourceName from the PDR. While following this recommendation is not
3120 mandatory, the MC should use it to inform a placement decision. If the MC does not follow the placement
3121 recommendation, it should read the MajorSchemaName field to identify the type of RDE Device they
3122 correspond to. Within the canon of standard Redfish schemas, there are comparatively few that reside at
3123 the top level, and each has a well-defined place it should appear within the hierarchy. The MC should

3124 thus make a simple map of which top-level schema types map to which places in the hierarchy and use
3125 this to place RDE Devices. In making these placement decisions, the MC should take information about
3126 the hardware platform topology into account so as to best reflect the overall Redfish system.

3127 It may happen that the MC encounters a schema it does not recognize. This can occur, for example, if a
3128 new schema type is standardized after the MC firmware is built. The handling of such cases is up to the
3129 MC. One possibility would be to place the schema in the OEM section under the most appropriate
3130 subobject. For an unknown DMTF standard schema, this should be the OEM/DMTF object. (To tell that a
3131 schema is DMTF standard, the MC may retrieve the published URI via GetSchemaURI command of
3132 clause 11.4, download the schema, and inspect the schema, namespace, or other content.)

3133 Naturally, wherever the MC places the RDE Device component, it shall add a link to the RDE Device
3134 component in the JSON retrieved by a client from the enclosing location.

3135 **14.2.7 LogEntry and LogEntryCollection resources**

3136 RDE Devices that support the LogEntry and LogEntryCollection resources must be aware that large
3137 volumes of LogEntries can overwhelm the 16 bit ResourceID space available for identifying Redfish
3138 Resource PDRs. To handle this case, it is recommended that RDE Devices provide a PDR for the
3139 LogEntryCollection but do NOT provide PDRs for the individual LogEntry instances. Instead, RDE
3140 Devices that support these schemas should also support the link expansion query parameter (see \$levels
3141 in [DSP0266](#) and the LinkExpand parameter from SupplyCustomRequestParameters in clause 12.2). This
3142 means that they should fill out the related resource links in the “Members” section of the response with
3143 bejResourceLinkExpansion data in which the encoded ResourceID is set to zero to ensure that the MC
3144 gets the COLLECTION_MEMBER_TYPE dictionary from the LogEntryCollection.

3145 **14.2.8 On-demand pagination**

3146 In Redfish, certain read operations may produce a very large amount of data. For example, reading a
3147 collection with many members will produce output with size proportional to the number of members.
3148 Rather than overload clients with a huge transfer of data, Redfish Devices may paginate it into chunks
3149 and provide one page at a time with an @odata.nextlink annotation giving a URI from which to retrieve
3150 the next piece.

3151 RDE supports the same pagination approach. It is entirely at the RDE Device’s discretion whether to
3152 paginate and where to draw pagination boundaries. When the RDE Device wishes to paginate, it shall
3153 insert an @odata.nextlink annotation, using a deferred binding pagination reference (see
3154 \$LINK.PDR<resource-ID>.PAGE<pagination-offset>% in clause 8.3), filling in the next page number for
3155 the data being returned. When the MC decodes this deferred binding, it shall create a temporary URI for
3156 the pagination and expose this pagination URI in the decoded JSON response it sends back to the client.
3157 Naturally, the encoded pagination URI must be decodable to extract the page number. Finally, when the
3158 client attempts a read from the pagination URI, the MC shall extract out the page number and send it to
3159 the RDE Device via the PaginationOffset field in the request message for the
3160 SupplyCustomRequestParameters command (clause 12.2).

3161 **14.2.9 Considerations for Redfish clients**

3162 No changes to behavior are required of Redfish clients in order to interact with BEJ-based RDE Devices;
3163 the details of providing them to the client are completely transparent from the client perspective. In fact, a
3164 fundamental design goal of this specification is that it should be impossible for a client to tell whether a
3165 Redfish message was ultimately serviced by an RDE Device that operates in JSON over HTTP/HTTPS or
3166 BEJ over PLDM.

3167

3168 **14.2.10 OriginOfCondition in Redfish events**

3169 The OriginOfCondition field in the Redfish event schema contains a link reference to a Redfish resource
3170 associated with a Redfish event. In typical use cases, resource data is read upon receiving the event to
3171 determine the resource state when the event transpired. This can happen either explicitly, from the client
3172 performing a read on the OriginOfCondition resource, or implicitly, if IncludeOriginOfCondition is set in the
3173 EventDestination when the client registered for Redfish events.

3174 RDE version 1.1 does not provide support for a device to populate the OriginOfCondition field with full
3175 resource data. However, an MC that wishes to minimize the timing window for the race condition may
3176 perform the appropriate read immediately upon receiving the Redfish event.

3177 **14.2.11 [MC] Merging dictionaries with OEM extensions**

3178 When merging dictionaries, MCs should consider that OEM extensions to Redfish schemas are
3179 enumerated alphabetically. In particular, the root objects (sets) of extensions (which come immediately
3180 under “OEM” inside the root object of the host schema) are likely to have conflicting sequence numbers if
3181 different sets of extensions appear in two different dictionaries for a given host schema.

3182 Additionally, no attempt has been made in this specification to make registry dictionaries able to be
3183 merged.

**ANNEX A
(normative)****Change log**

3188

Version	Date	Description
1.0.0	2019-06-25	
1.0.1	2019-12-09	Errata update
1.1.0	2020-11-19	<ul style="list-style-type: none">Added support for nested annotationsEnhanced message registry support, including a new registry dictionary, BEJ encodingImproved ability to identify OEM extensions to schemasAdded support for PLDM common multipart transfers
1.1.1	2021-10-27	Errata update

3189