



1
2
3
4

Document Identifier: DSP0218

Date: 2021-02-11

Version: 1.1.0

5
6

Platform Level Data Model (PLDM) for Redfish Device Enablement

7
8
9
10
11

Supersedes: 1.0.1

Document Class: Normative

Document Status: Published

Document Language: en-US

12

13 Copyright Notice

14 Copyright © 2019, 2021 DMTF. All rights reserved.

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

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29 withdrawn or modified after publication, and shall be indemnified and held harmless by any party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

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

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

CONTENTS

37	Foreword	9
38	Acknowledgments	9
39	Introduction.....	10
40	Document conventions.....	10
41	1 Scope	11
42	2 Normative references	11
43	3 Terms and definitions	13
44	4 Symbols and abbreviated terms.....	15
45	5 Conventions	16
46	5.1 Reserved and unassigned values.....	16
47	5.2 Byte ordering.....	16
48	5.3 PLDM for Redfish Device Enablement data types	16
49	5.3.1 varstring PLDM data type	17
50	5.3.2 schemaClass PLDM data type	17
51	5.3.3 nnint PLDM data type	18
52	5.3.4 bejEncoding PLDM data type	18
53	5.3.5 bejTuple PLDM data type	19
54	5.3.6 bejTupleS PLDM data type	19
55	5.3.7 bejTupleF PLDM data type	19
56	5.3.8 bejTupleL PLDM data type	20
57	5.3.9 bejTupleV PLDM data type	21
58	5.3.10 bejNull PLDM data type	21
59	5.3.11 bejInteger PLDM data type	21
60	5.3.12 bejEnum PLDM data type	22
61	5.3.13 bejString PLDM data type	22
62	5.3.14 bejReal PLDM data type	22
63	5.3.15 bejBoolean PLDM data type	23
64	5.3.16 bejBytestring PLDM data type	23
65	5.3.17 bejSet PLDM data type	24
66	5.3.18 bejArray PLDM data type	24
67	5.3.19 bejChoice data PLDM type	24
68	5.3.20 bejPropertyAnnotation PLDM data type	24
69	5.3.21 bejRegistryItem PLDM data type	25
70	5.3.22 bejResourceLink PLDM data type	26
71	5.3.23 bejResourceLinkExpansion PLDM data type	26
72	5.3.24 bejLocator PLDM data type	26
73	5.3.25 rdeOpID PLDM data type	27
74	6 PLDM for Redfish Device Enablement version	27
75	7 PLDM for Redfish Device Enablement overview	27
76	7.1 Redfish Provider architecture overview	28
77	7.1.1 Roles	28
78	7.2 Redfish Device Enablement concepts	29
79	7.2.1 RDE Device discovery and registration	29
80	7.2.2 Data instances of Redfish schemas: Resources	31
81	7.2.3 Dictionaries	34
82	7.2.4 Redfish Operation support.....	41
83	7.2.5 PLDM RDE Events	51
84	7.2.6 Task support	53
85	7.3 Type code	54
86	7.4 Transport protocol type supported.....	54
87	7.5 Error completion codes.....	54

88	7.6	Timing specification	56
89	8	Binary Encoded JSON (BEJ)	57
90	8.1	BEJ design principles.....	57
91	8.2	SFLV tuples	57
92	8.2.1	Sequence number.....	58
93	8.2.2	Format.....	58
94	8.2.3	Length	58
95	8.2.4	Value.....	58
96	8.3	Deferred binding of data	59
97	8.4	BEJ encoding.....	61
98	8.4.1	Conversion of JSON data types to BEJ.....	61
99	8.4.2	Resource links	62
100	8.4.3	Registry items	62
101	8.4.4	Annotations	62
102	8.4.5	Choice encoding for properties that support multiple data types	64
103	8.4.6	Properties with invalid values	64
104	8.4.7	Properties missing from dictionaries.....	64
105	8.5	BEJ decoding.....	64
106	8.5.1	Conversion of BEJ data types to JSON.....	64
107	8.5.2	Annotations	66
108	8.5.3	Sequence numbers missing from dictionaries.....	66
109	8.5.4	Sequence numbers for read-only properties in modification Operations	66
110	8.6	Example encoding and decoding.....	67
111	8.6.1	Example dictionary.....	67
112	8.6.2	Example encoding	69
113	8.6.3	Example decoding	73
114	8.7	BEJ locators	76
115	9	Operational behaviors	76
116	9.1	Initialization (MC perspective).....	76
117	9.1.1	Sample initialization ladder diagram	77
118	9.1.2	Initialization workflow diagram	78
119	9.2	Operation/Task lifecycle.....	80
120	9.2.1	Example Operation command sequence diagrams.....	80
121	9.2.2	Operation/Task overview workflow diagrams (Operation perspective)	84
122	9.2.3	RDE Operation state machine (RDE Device perspective)	92
123	9.3	Event lifecycle	105
124	10	PLDM commands for Redfish Device Enablement.....	108
125	11	PLDM for Redfish Device Enablement – Discovery and schema commands	109
126	11.1	NegotiateRedfishParameters command (0x01) format	109
127	11.2	NegotiateMediumParameters command (0x02) format.....	111
128	11.3	GetSchemaDictionary command (0x03) format	112
129	11.4	GetSchemaURI command (0x04) format	113
130	11.5	GetResourceETag command (0x05) format.....	114
131	11.6	GetOEMCount command (0x06) format.....	115
132	11.7	GetOEMName command (0x07) format.....	116
133	11.8	GetRegistryCount command (0x08) format.....	116
134	11.9	GetRegistryDetails command (0x09) format.....	117
135	11.10	SelectRegistryVersion command (0x0A) format.....	118
136	11.11	GetMessageRegistry command (0x0B) format	118
137	11.12	GetSchemaFile command (0x0C) format	119
138	12	PLDM for Redfish Device Enablement – RDE Operation and Task commands	120
139	12.1	RDEOperationInit command (0x10) format	120
140	12.2	SupplyCustomRequestParameters command (0x11) format.....	123
141	12.3	RetrieveCustomResponseParameters command (0x12) format.....	127
142	12.4	RDEOperationComplete command (0x13) format.....	128

143 12.5 RDEOperationStatus command (0x14) format 129

144 12.6 RDEOperationKill command (0x15) format 132

145 12.7 RDEOperationEnumerate command (0x16) format 132

146 13 PLDM for Redfish Device Enablement – Utility commands 134

147 13.1 RDEmultipartSend command (0x30) format 134

148 13.2 RDEmultipartReceive command (0x31) format 135

149 14 Additional Information 137

150 14.1 RDE Multipart transfers 137

151 14.1.1 Flag usage for RDEmultipartSend 137

152 14.1.2 Flag usage for RDEmultipartReceive 138

153 14.1.3 RDE Multipart transfer examples 138

154 14.2 Implementation notes 140

155 14.2.1 Schema updates 140

156 14.2.2 Storage of dictionaries 140

157 14.2.3 Dictionaries for related schemas 140

158 14.2.4 [MC] HTTP/HTTPS POST Operations 141

159 14.2.5 Consistency checking of read Operations 142

160 14.2.6 [MC] Placement of RDE Device resources in the outward-facing Redfish URI

161 hierarchy 142

162 14.2.7 LogEntry and LogEntryCollection resources 143

163 14.2.8 On-demand pagination 143

164 14.2.9 Considerations for Redfish clients 143

165 14.2.10 OriginOfCondition in Redfish events 144

166 14.2.11 [MC] Merging dictionaries with OEM extensions 144

167 ANNEX A (normative) Change log 145

168

169 **Figures**

170	Figure 1 – RDE Roles	29
171	Figure 2 – Example linking of Redfish Resource and Redfish Entity Association PDRs	33
172	Figure 3 – Schema linking without Redfish entity association PDRs	34
173	Figure 4 – Dictionary binary format.....	38
174	Figure 5 – DummySimple schema.....	68
175	Figure 6 – DummySimple dictionary – binary form.....	69
176	Figure 7 – Example Initialization ladder diagram	78
177	Figure 8 – Typical RDE Device discovery and registration.....	80
178	Figure 9 – Simple read Operation ladder diagram.....	81
179	Figure 10 – Complex Read Operation ladder diagram	82
180	Figure 11 – Write Operation ladder diagram.....	83
181	Figure 12 – Write Operation with long-running Task ladder diagram	84
182	Figure 13 – RDE Operation lifecycle overview (holistic perspective)	88
183	Figure 14 – RDE Task lifecycle overview (holistic perspective)	91
184	Figure 15 – Operation lifecycle state machine (RDE Device perspective)	105
185	Figure 16 – Redfish event lifecycle overview.....	107
186	Figure 17 – RDEMultipartSend example	139
187	Figure 18 – RDEMultipartReceive example.....	140
188		

189 **Tables**

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

213 Table 24 – bejPropertyAnnotation value encoding for BEJ 25

214 Table 25 – bejPropertyAnnotation value encoding example 25

215 Table 26 – bejRegistryItem value encoding for BEJ 26

216 Table 27 – bejResourceLink value encoding for BEJ 26

217 Table 28 – bejResourceLinkExpansion value encoding for BEJ 26

218 Table 29 – bejLocator value encoding 26

219 Table 30 – rdeOpID data structure 27

220 Table 31 – Redfish dictionary binary format 36

221 Table 32 – Dictionary entry example for a property supporting multiple formats 39

222 Table 33 – Redfish Operations 41

223 Table 34 – Redfish operation headers 43

224 Table 35 – Redfish operation request query options 48

225 Table 36 – Query parameter support requirement 49

226 Table 37 – PLDM for Redfish Device Enablement completion codes 54

227 Table 38 – HTTP codes for standard PLDM completion codes 56

228 Table 39 – Timing specification 56

229 Table 40 – Sequence number dictionary indication 58

230 Table 41 – JSON data types supported in BEJ 58

231 Table 42 – BEJ deferred binding substitution parameters 59

232 Table 43 – Message annotation related property BEJ locator encoding 63

233 Table 44 – DummySimple dictionary (tabular form) 68

234 Table 45 – Initialization Workflow 79

235 Table 46 – Operation lifecycle overview 85

236 Table 47 – Task lifecycle overview 89

237 Table 48 – Task lifecycle state machine 93

238 Table 49 – Event lifecycle overview 105

239 Table 50 – PLDM for Redfish Device Enablement command codes 108

240 Table 51 – NegotiateRedfishParameters command format 110

241 Table 52 – NegotiateMediumParameters command format 112

242 Table 53 – GetSchemaDictionary command format 113

243 Table 54 – GetSchemaURI command format 114

244 Table 55 – GetResourceETag command format 115

245 Table 56 – GetOEMCount command format 115

246 Table 57 – GetOEMName command format 116

247 Table 58 – GetRegistryCount command format 116

248 Table 59 – GetRegistryDetails command format 117

249 Table 60 – SelectRegistryVersion command format 118

250 Table 61 – GetMessageRegistry command format 118

251 Table 62 – GetSchemaFile command format 119

252 Table 63 – RDEOperationInit command format 121

253 Table 64 – SupplyCustomRequestParameters command format 124

254 Table 65 – RetrieveCustomResponseParameters command format 128

255 Table 66 – RDEOperationComplete command format 129

256 Table 67 – RDEOperationStatus command format 129

257 Table 68 – RDEOperationKill command format 132

258 Table 69 – RDEOperationEnumerate command format 133

259 Table 70 – RDEMultipartSend command format 134

260 Table 71 – RDEMultipartReceive command format 136

261

262

263

Foreword

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

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

268 Acknowledgments

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

270 Editor:

- 271 • Bill Scherer – Hewlett Packard Enterprise

272 Contributors:

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

297

Introduction

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

309 Document conventions

310 Clause naming conventions

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

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

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

321 Typographical conventions

322 The following typographical conventions are used in this document:

- 323 • 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 2. 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.

2 Normative references

The following referenced documents are indispensable for the application of this document. For dated or versioned references, only the edition cited (including any corrigenda or DMTF update versions) applies.

- 366 For references without a date or version, the latest published edition of the referenced document
367 (including any corrigenda or DMTF update versions) applies. Earlier versions may not provide sufficient
368 support for this specification.
- 369 DMTF DSP0222, *Network Controller Sideband Interface (NC-SI) Specification 1.1*,
370 https://www.dmtf.org/sites/default/files/standards/documents/DSP0222_1.1.pdf
- 371 DMTF DSP0236, *MCTP Base Specification 1.2*,
372 http://dmtof.org/sites/default/files/standards/documents/DSP0236_1.2.pdf
- 373 DMTF DSP0240, *Platform Level Data Model (PLDM) Base Specification 1.1*,
374 http://dmtof.org/sites/default/files/standards/documents/DSP0240_1.1.pdf
- 375 DMTF DSP0241, *Platform Level Data Model (PLDM) Over MCTP Binding Specification 1.0*,
376 http://dmtof.org/sites/default/files/standards/documents/DSP0241_1.0.pdf
- 377 DMTF DSP0245, *Platform Level Data Model (PLDM) IDs and Codes Specification 1.3*,
378 http://dmtof.org/sites/default/files/standards/documents/DSP0245_1.3.pdf
- 379 DMTF DSP0248, *Platform Level Data Model (PLDM) for Platform Monitoring and Control Specification*
380 *1.1*, http://dmtof.org/sites/default/files/standards/documents/DSP0248_1.1.pdf
- 381 DMTF DSP0266, *Redfish Scalable Platforms Management API Specification 1.6*,
382 http://www.dmtf.org/sites/default/files/standards/documents/DSP0266_1.6.pdf
- 383 DMTF DSP0267, *PLDM for Firmware Update Specification 1.0*,
384 https://www.dmtf.org/sites/default/files/standards/documents/DSP0267_1.0.pdf
- 385 DMTF DSP4004, *DMTF Release Process 2.4*,
386 http://dmtof.org/sites/default/files/standards/documents/DSP4004_2.4.pdf
- 387 ECMA International Standard ECMA-404, *The JSON Data Interchange Syntax*, [http://www.ecma-](http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf)
388 [international.org/publications/files/ECMA-ST/ECMA-404.pdf](http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf)
- 389 IETF RFC2781, *UTF-16, an encoding of ISO 10646*, February 2000,
390 <http://www.ietf.org/rfc/rfc2781.txt>
- 391 IETF STD63, *UTF-8, a transformation format of ISO 10646* <http://www.ietf.org/rfc/std/std63.txt>
- 392 IETF RFC4122, *A Universally Unique Identifier (UUID) URN Namespace*, July 2005,
393 <http://www.ietf.org/rfc/rfc4122.txt>
- 394 IETF RFC4646, *Tags for Identifying Languages*, September 2006,
395 <http://www.ietf.org/rfc/rfc4646.txt>
- 396 IETF RFC7231, R. Fielding et al., *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*,
397 <https://tools.ietf.org/html/rfc7231>
- 398 IETF RFC 7232, R. Fielding et al., *Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests*,
399 <http://www.ietf.org/rfc/rfc7232.txt>
- 400 IETF RFC 7234, R. Fielding et al., *Hypertext Transfer Protocol (HTTP/1.1): Caching*,
401 <https://tools.ietf.org/rfc/rfc7234.txt>
- 402 ISO 8859-1, *Final Text of DIS 8859-1, 8-bit single-byte coded graphic character sets — Part 1: Latin*
403 *alphabet No.1*, February 1998
- 404 ISO/IEC Directives, Part 2, *Rules for the structure and drafting of International Standards*,
405 <http://isotc.iso.org/livelink/livelink.exe?func=ll&objId=4230456&objAction=browse&sort=subtype>

406 ITU-T X.690 (08/2015), *Information technology – ASN.1 encoding rules: Specification of Basic Encoding*
407 *Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)*,
408 <http://handle.itu.int/11.1002/1000/12483>

409 Open Data Protocol, <https://www.oasis-open.org/standards#odatav4.0>

410 3 Terms and definitions

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

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

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

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

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

428 3.1

429 Action

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

432 3.2

433 Annotation

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

436 3.3

437 Client

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

440 3.4

441 Collection

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

444 3.5

445 Device Component

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

- 447 **3.6**
448 **Dictionary**
449 A binary lookup table containing translation information that allows conversion between BEJ and JSON
450 formats of data for a given resource
- 451 **3.7**
452 **Discovery**
453 The process by which an MC determines that an RDE Device supports PLDM for Redfish Device
454 Enablement
- 455 **3.8**
456 **Major Schema**
457 The primary schema defining the format of a collection of data, usually a published standard Redfish
458 schema.
- 459 **3.9**
460 **Member**
461 Any of the independent resources contained within a collection
- 462 **3.10**
463 **Metadata**
464 Information that describes data of interest, such as its type format, length in bytes, or encoding method
- 465 **3.11**
466 **OData**
467 The [Open Data protocol](#), a source of annotations in Redfish, as defined by OASIS.
- 468 **3.12**
469 **OEM Extension**
470 Any manufacturer-specific addition to major schema
- 471 **3.13**
472 **Property**
473 An individual datum contained within a Resource
- 474 **3.14**
475 **RDE Device**
476 Any PLDM terminus containing an RDE Provider that requires the intervention of an MC to receive
477 Redfish communications
- 478 **3.15**
479 **RDE Provider**
480 Any RDE Device that responds to RDE Operations. See also **Redfish Provider**.
- 481 **3.16**
482 **RDE Operation**
483 The sequence of PLDM messages and operations that represent a Redfish Operation being executed by
484 an MC and/or an RDE Device on behalf of a client. See also **Redfish Operation**.

- 485 **3.17**
486 **Redfish Operation**
487 Any Redfish operation transmitted via HTTP or HTTPS from a client to an MC for execution. See also
488 **RDE Operation**.
- 489 **3.18**
490 **Redfish Provider**
491 Any entity that responds to Redfish Operations. See also **RDE Provider**.
- 492 **3.19**
493 **Registration**
494 The process of enabling a compliant RDE Device with an MC to be an RDE Provider
- 495 **3.20**
496 **Resource**
497 A hierarchical set of data organized in the format specified in a Redfish Schema.
- 498 **3.21**
499 **Schema**
500 Any regular structure for organizing one or more fields of data in a hierarchical format
- 501 **3.22**
502 **Task**
503 Any Operation for which an RDE Device cannot complete execution in the time allotted to respond to the
504 PLDM triggering command message sent from the MC and for which the MC creates standard Redfish
505 Task and TaskMonitor objects
- 506 **3.23**
507 **Triggering Command**
508 The PLDM command that supplies the last bit of data needed for an RDE Device to begin execution of an
509 RDE Operation
- 510 **3.24**
511 **Truncated**
512 When applied to a dictionary, one that is limited to containing conversion information for properties
513 supported by an RDE Device
- 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
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 5.3.14
bejBoolean	Boolean data encoded for BEJ per clause 5.3.15

Data Type	Interpretation
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
550 vast majority of schemas for Redfish resources.
- 551 • EVENT: the standard DMTF-published event schema, for occurrences that clients may wish to
552 be notified about.
- 553 • ANNOTATION: the standard DMTF-published annotation schema that captures metadata about
554 a major schema or payload. This schemaClass shall not be used as the primary schema for
555 BEJ encodings as annotations are specially encoded alongside the primary schema.
- 556 • COLLECTION_MEMBER_TYPE: for resources that correspond to Redfish collections, this
557 class enables access to the major schema for members of that collection from the context of the
558 collection resource. (Unlike regular resources, collections in Redfish are unversioned and
559 contain multiple members.) This schemaClass shall not be used for BEJ encodings.

- 560
- 561
- 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).
- 563
- 564
- 565

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

Type	Description
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> <p>[7:4] - principal data type; see Table 9 below for values</p> <p>[3] - reserved flag. 1b indicates the flag is set</p> <p>[2] - nullable_property flag ***. 1b indicates the flag is set</p> <p>[1] - read_only_property_and_top_level_annotation flag **. 1b indicates the flag is set</p> <p>[0] - deferred_binding flag *. 1b indicates the flag is set</p>

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 twos 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

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)

612 **5.3.12 bejEnum PLDM data type**613 **Table 14 – bejEnum value encoding for BEJ**

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

614 **5.3.13 bejString PLDM data type**

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

616 **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

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

619 **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

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

622 **5.3.14 bejReal PLDM data type**623 BEJ encoding for *whole*, *fract*, and *exp* that represent the base 10 encoding $whole.fract \times 10^{exp}$.

624 NOTE There is no need to express special values (positive infinity, negative infinity, NaN, negative zero) because
 625 these cannot be expressed in JSON.

626

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)

627 In order to distinguish between the cases where the exponent is zero and the exponent is omitted
 628 entirely, an omitted exponent shall be encoded with a length of zero bytes; the exponent of zero shall be
 629 encoded with a single byte (of value zero). (These cases are numerically identical but visually distinct in
 630 standard text-based JSON encoding.)

631 As an example, Table 18 shows the encoding of the JSON number “1.0005e+10”:

632

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	0x0A	<i>Exp</i> (10)

633 **5.3.15 bejBoolean PLDM data type**

634 The bejBoolean PLDM data type captures boolean data.

635

Table 19 – bejBoolean value encoding for BEJ

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

636 **5.3.16 bejBytestring PLDM data type**

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

639

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)

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

640 5.3.17 bejSet PLDM data type

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

643 **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)

644 5.3.18 bejArray PLDM data type

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

647 **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)

648 5.3.19 bejChoice data PLDM type

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

652 **Table 23 – bejChoice value encoding for BEJ**

Type	Description
bejTuple	Selected option

653 5.3.20 bejPropertyAnnotation PLDM data type

654 The bejPropertyAnnotation PLDM data type captures the encoding of a property annotation in the form
655 property@annotationtype.annotationname. When the bejTupleF format code is set to

656 bejPropertyAnnotation, the sequence number bejTupleS in the outer bejTuple shall be for the annotated
 657 property. The value bejTupleV of the outer bejTuple shall be as follows:

658 **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

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

660 "Status@Redfish.RequiredOnCreate" : false

661 **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

662 **5.3.21 bejRegistryItem PLDM data type**

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

666

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

667 5.3.22 bejResourceLink PLDM data type

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

671

Table 27 – bejResourceLink value encoding for BEJ

Type	Description
nnint	ResourceID of Redfish Resource PDR for linked schema

672 5.3.23 bejResourceLinkExpansion PLDM data type

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

677

Table 28 – bejResourceLinkExpansion value encoding for BEJ

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

678 5.3.24 bejLocator PLDM data type

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

685

Table 29 – bejLocator value encoding

Type	Description
nnint	LengthBytes Total length in bytes of the N sequence numbers comprising this locator
bejTupleS	Sequence number [0]
bejTupleS	Sequence number [1]
bejTupleS	Sequence number [2]

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

686 **5.3.25 rdeOpID PLDM data type**

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

689 **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.

690 **6 PLDM for Redfish Device Enablement version**

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

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

696 **7 PLDM for Redfish Device Enablement overview**

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

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

707 Implementing the Redfish protocol as defined by [DSP0266](#) is a big challenge when passing requests to
 708 and from devices such as network adapters that have highly limited processing capabilities and memory
 709 space. Redfish’s messages are prohibitively large because they are encoded for human readability in
 710 HTTP/HTTPS using JavaScript Object Notation (JSON). This specification details a compressed
 711 encoding of Redfish payloads that is suitable for such devices. It further identifies a common method to
 712 use PLDM to communicate these messages between a management controller and the devices that host
 713 the data the operations target. The functionality of providing a complete Redfish service is distributed
 714 across components that function in different roles; this is discussed in more detail in clause 7.1.1.

715 The basic format for PLDM messages is defined in [DSP0240](#). The specific format for carrying PLDM
 716 messages over a particular transport or medium is given in companion documents to the base

717 specification. For example, [DSP0241](#) defines how PLDM messages are formatted and sent using MCTP
718 as the transport. Similarly, [DSP0222](#) defines how PLDM messages are formatted and sent using NC-SI
719 and RBT as the transport. The payloads for PLDM messages are application specific. The Platform Level
720 Data Model (PLDM) for Redfish Device Enablement specification defines PLDM message payloads that
721 support the following items and capabilities:

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

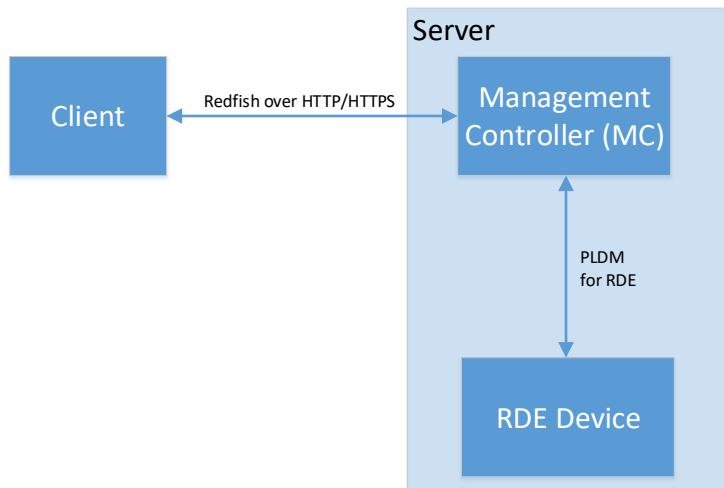
741 7.1 Redfish Provider architecture overview

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

749 7.1.1 Roles

750 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.



751

752

Figure 1 – RDE Roles

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

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

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

770 7.2 Redfish Device Enablement concepts

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

772 7.2.1 RDE Device discovery and registration

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

779 7.2.1.1 RDE Device discovery

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

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

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

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

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

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

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

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

811 7.2.1.2 RDE Device registration

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

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

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

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

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

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

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

854 7.2.2 Data instances of Redfish schemas: Resources

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

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

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

869 Links between resources can be modeled in three different ways. Direct subordinate linkage, such as
870 physical enclosure or being a component in a ComputerSystem, may be represented by setting the
871 ContainingResourceID field of the Redfish Resource PDR to the Resource ID for the parent resource. In
872 Redfish terminology, this relation is used to show subordinate resources. The parent field for the logical
873 root of a device component is set to EXTERNAL, 0x0000.

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

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

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

889 7.2.2.1 Alignment of resources

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

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

899 7.2.2.2 Example linking of PDRs within RDE Devices

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

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

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

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

913 The connections discussed so far are all direct parent linkages in the Redfish Resource PDRs because
914 the links they represent are the direct subordinate resource links from the standard Redfish storage
915 model. However, the Redfish storage model also includes notations that drives are related to (contained
916 within) a volume and that drives are related to (present inside) a chassis. These resource relations can be
917 modeled using Redfish Entity Association PDRs if the MC is managing the links. Alternatively, they can
918 be implicitly managed by the RDE Device. In this case, the RDE Device will expose the links itself by
919 filling in a Links section of the relevant resource data with references to the linked resources. While the

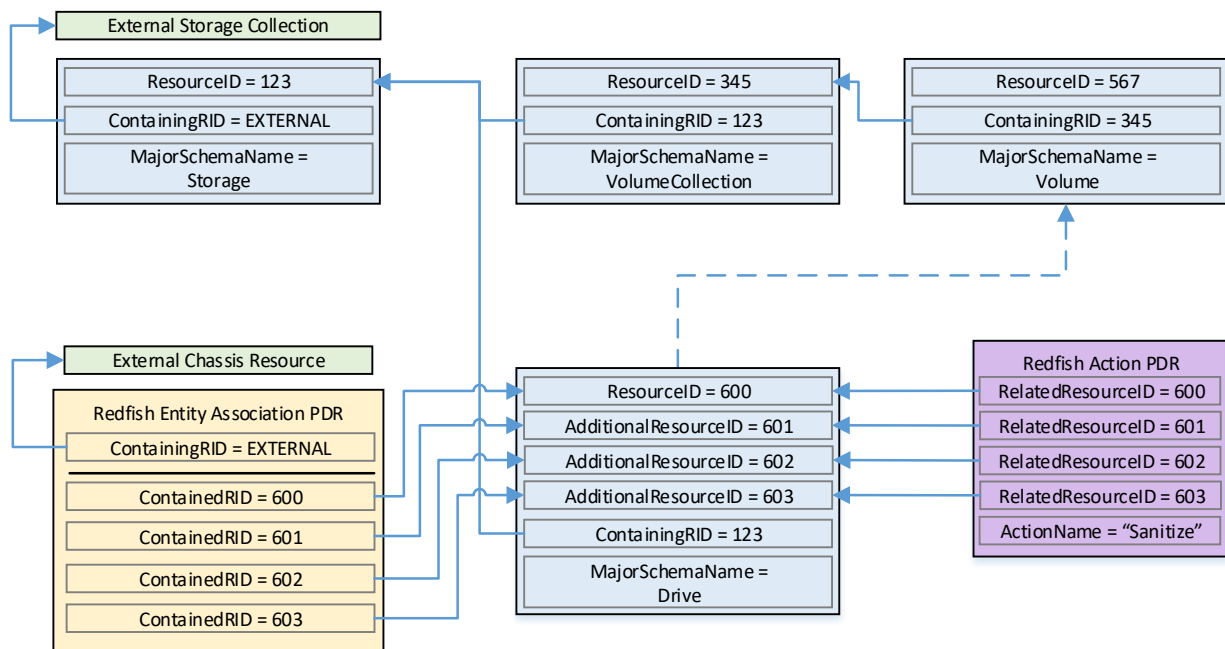
920 RDE Device could in theory provide a Redfish Entity Association PDR for this case, it serves no purpose
 921 for the MC.

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

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

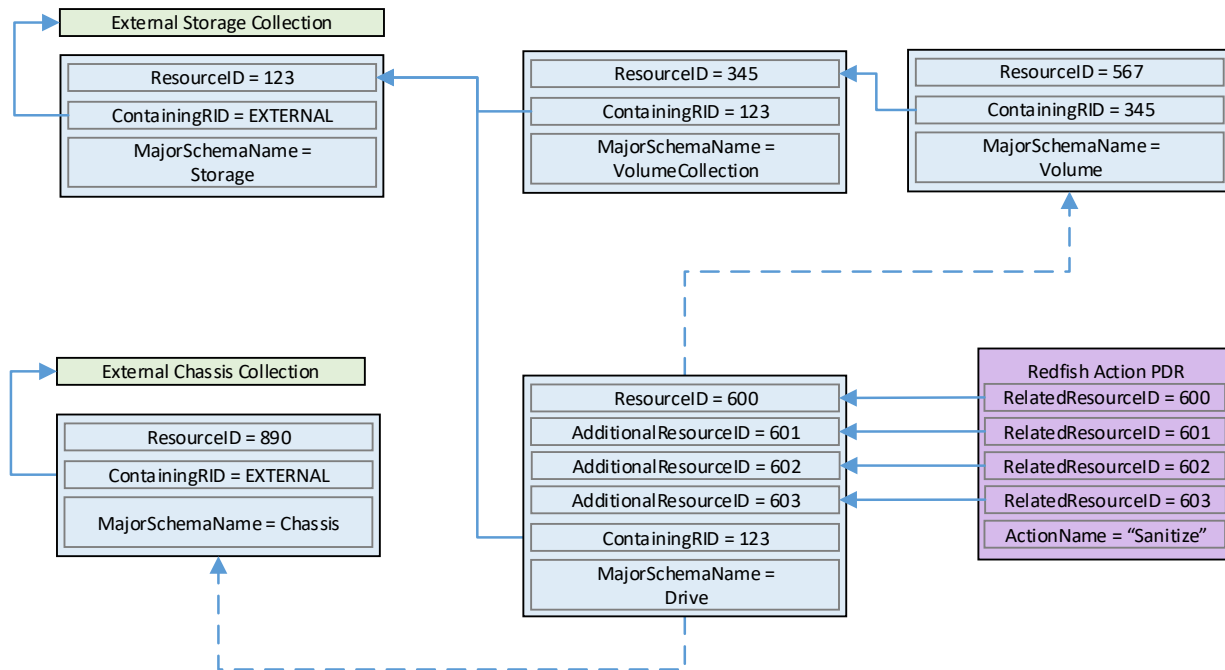
932 Finally, each of the drives supports a Sanitize operation. This is shown by instantiating a Redfish Action
 933 PDR naming the Sanitize action and linking it to each of the drives.

934 As an alternative to the PDR layout of Figure 2, the RDE Device exposes its own chassis resource
 935 (labeled as Resource ID 890) rather than having the drives be part of an external chassis. The PDR for
 936 this chassis resource shows ContainingResourceID EXTERNAL to demonstrate that it belongs in the
 937 system chassis collection resource. With this modification, the links between the chassis resource and the
 938 drives can be managed internally by the RDE Device and hence no Redfish Entity Association PDR is
 939 necessary.



940

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



942

943

Figure 3 – Schema linking without Redfish entity association PDRs

944 **7.2.3 Dictionaries**

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

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

- 953 • Standard Redfish data schema (aka the major schema)
- 954 • Standard Redfish Event schema
- 955 • Standard Redfish Annotation schema
- 956 • Standard Redfish Error schema

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

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

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

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.

963 may include OEM extensions, and that may also be truncated to contain only entries for properties
 964 supported by the RDE Device.

965 7.2.3.1 Canonizing a schema into a dictionary

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

- 972 1) The children properties (properties immediately contained within other properties such as sets
 973 or arrays) shall collectively receive an independent set of sequence numbers ranging from zero
 974 to $N - 1$, where N is the number of children. Sequence numbers for properties that do not share
 975 a common parent are not related in any way.
- 976 2) For the initial revision of a Redfish schema (usually v1.0), sequence numbers shall be assigned
 977 according to a strict alphabetical ordering of the property names from the schema.
- 978 3) In order to preserve backward compatibility with earlier versions of schemas, for subsequent
 979 revisions of Redfish schemas, the sequence numbers for child properties added in that revision
 980 shall be assigned sequence numbers N to $N + A - 1$, where N is the number of sequence
 981 numbers assigned in the previous revision and A is the number of properties added in the
 982 present revision. (In other words, we append to the existing set and use sequence numbers
 983 beginning with the next one available.) The new sequence numbers shall be assigned
 984 according to a strict alphabetical ordering of their names from the schema.
- 985 4) In the event that a property is deleted from a schema, its sequence number shall not be reused;
 986 the sequence number for the deleted property shall forever remain allocated to that property.
- 987 5) As with properties, the values of an enumeration shall collectively receive an independent set of
 988 sequence numbers ranging from zero to $N - 1$, where N is the number of enumeration values.
 989 Sequence numbers for enumeration values not belonging to the same enumeration are not
 990 related in any way.
- 991 6) For the initial version of a Redfish schema, sequence numbers for enumeration values shall be
 992 assigned according to a strict alphabetical ordering of the enumeration values from the schema.
- 993 7) In order to preserve backward compatibility with earlier versions of schemas, for subsequent
 994 revisions of Redfish schemas, the sequence numbers for enumeration values added in that
 995 revision shall be assigned sequence numbers N to $N + A - 1$, where N is the number of
 996 sequence numbers assigned in the previous revision and A is the number of enumeration
 997 values added in the present revision. The new sequence numbers shall be assigned according
 998 to a strict alphabetical ordering of their value strings from the schema.
- 999 8) In the event that an enumeration value is deleted from a schema, its sequence number shall not
 1000 be reused; the sequence number for the deleted enumeration value shall forever remain
 1001 allocated to that enumeration value.

1002 After the sequence numbers for properties and enumeration values are assigned, they shall be
 1003 collected together with other information from the Redfish and OEMs schema to build a dictionary in
 1004 the format detailed in clause 7.2.3.2. For every Redfish Resource PDR the RDE Device offers, it shall
 1005 maintain a dictionary that it can send to the MC on demand in response to a GetSchemaDictionary
 1006 command (clause 11.2).

1007 NOTE Rules 2 and 3 above imply that schema child properties may not be in strict alphabetical order. For example,
 1008 suppose a property node in a schema started with child fields "red", "orange", and "yellow" in version 1.0. Because
 1009 this is the initial version, the fields would be alphabetized: "orange" would get sequence number 0; "red", 1; and
 1010 "yellow" would get 2. If version 1.1 of the schema were to add "blue" and "green", they would be assigned sequence

1011 numbers 3 and 4 respectively (because that is the alphabetical ordering of the new properties). The initial three
 1012 properties retain their original sequence numbers.

1013 For all custom dictionaries, including all truncated dictionaries, the sequence numbers listed for
 1014 standard Redfish schema properties supported by the RDE Device shall match the sequence
 1015 numbers for those same properties from the standard dictionary. This allows MCs to potentially
 1016 merge related dictionaries from RDE Devices that share a common class.

1017 Sequence numbers for array elements shall be assigned to match the zero-based index of the array
 1018 element.

1019 NOTE The ordering rules provided in this clause apply to dictionaries only. In particular, data encoded in either JSON
 1020 or BEJ format is by definition unordered.

1021 **7.2.3.2 Dictionary binary format**

1022 The binary format of dictionaries shall be as follows. All integer fields are stored little endian:

1023 **Table 31 – Redfish dictionary binary format**

Type	Dictionary Data
uint8	VersionTag Dictionary format version tag: 0x00 for DSP0218 v1.0.0
bitfield8	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
uint16	EntryCount Number N of entries contained in this dictionary
uint32	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.
uint32	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.
bejTupleF	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.
uint16	SequenceNumber [0] Entry 0 property sequence number
uint16	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.
uint16	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.

Type	Dictionary Data
uint8	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.
uint16	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.
...	...
bejTupleF	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.
uint16	SequenceNumber [N – 1] Entry (N – 1) property sequence number
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.

1024 Intuitively, the dictionary binary format may be thought of as a header (orange) followed by an array of
 1025 entry data (blue) followed by a table of the strings (green) naming the properties and enumeration values
 1026 for the entries. Figure 4 displays this data in graphical format:

1027

Byte offset				
DWORD	+0	+1	+2	+3
00	VersionTag 0x00	DictionaryFlags	EntryCount ₁	EntryCount ₂
01	SchemaVersion ₁	SchemaVersion ₂	SchemaVersion ₃	SchemaVersion ₄
02	DictionarySize ₁	DictionarySize ₂	DictionarySize ₃	DictionarySize ₄
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
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	...
...	Copyright _{terminator}			

1028 **Figure 4 – Dictionary binary format**

1029 * Name strings will not be present in the dictionary for anonymous format options of BEJ Choice-
 1030 formatted properties or for anonymous array entries.

1031 **7.2.3.2.1 Hierarchical organization of entries**

1032 Within this binary format, the entries shall be sorted into clusters representing a breadth-first traversal of
 1033 the hierarchy presented by a schema. Each cluster shall in turn consist of all the sibling nodes contained
 1034 within a common parent, sorted by sequence number per the rules defined in clause 7.2.3 above. An
 1035 example of this organization may be found in clause 8.6.1.

1036 NOTE While not mandatory, it is acceptable that multiple dictionary entries may point to a common complex subtype
 1037 to allow reuse of that information and reduce the overall size of the dictionary. For example, Resource.status is
 1038 commonly used multiple times within the same schema, so having a single offset for it can trim some length from the
 1039 dictionary.

1040 **7.2.3.3 Properties that support multiple formats**

1041 For properties that support multiple formats, the dictionary shall contain an entry linking the property
 1042 name string to the BEJ Choice format. This choice entry shall in turn link to a series of anonymous child
 1043 entries (name offset = 0x0000) that are of the various data formats supported by the property. For
 1044 example, if a TCP/IP hostname property supports both string (“www.dmtf.org”) and numeric (the 32-bit
 1045 equivalent of 72.47.235.184) values, the dictionary might contain rows such as the following:

1046 **Table 32 – Dictionary entry example for a property supporting multiple formats**

Row	Sequence Number	Format	Name	Child Pointer
...
15	0	choice	“hostname”	18
...
18	0	string	null	null
19	1	integer	null	null
...

1047 NOTE Following the rules for sequence number assignment (see clause 7.2.3.1), each cluster of properties
 1048 contained within a given set and each cluster of enumeration values are numbered separately. Hence sequence
 1049 numbers may be repeated within a dictionary.

1050 An exception to this rule is that properties that support null and exactly one other data format shall be
 1051 collapsed into a single entry in the dictionary listing only the non-null data format. The nullable_property
 1052 bit in the bejTupleF value of the format entry in the dictionary shall be set to 1b in this case. This case is
 1053 common in the standard Redfish schemas, where most properties are nullable. This is flagged with the
 1054 “nullable” keyword in the CSDL schemas, but in the JSON schemas, it manifests as the supported type
 1055 list for the property consisting of NULL and either a solitary second type or a collection of strings that form
 1056 an enumeration.

1057 **7.2.3.4 Annotation dictionary format**

1058 Standard Redfish annotations are derived from three sources: the Redfish, odata, and message
 1059 schemas. The annotations that can be part of a JSON payload are collected together into the redfish-
 1060 payload-annotations.vX.Y.Z.json schema file. This clause details special notes that apply to building the
 1061 annotation dictionary:

- 1062 • The dictionary entries for properties in the annotation dictionary shall include the entire name of
 1063 the annotation, beginning with the ‘@’ sign and including both the annotation source (one of
 1064 redfish, message, or odata) and the annotation’s name itself. For example, the dictionary Name
 1065 field for the @odata.id property shall be an offset to the string “@odata.id”.
- 1066 • The dictionary entries for patternProperties in the annotation dictionary shall be stripped of the
 1067 wildcard patterns before the ‘@’ sign and of the trailing ‘\$’ sign but shall otherwise be treated
 1068 identically to standard properties. For example, the dictionary Name field for the “^[a-zA-Z][a-
 1069 zA-Z0-9_]*”?@Message.ExtendedInfo\$” patternProperty shall be an offset to the string
 1070 “@Message.ExtendedInfo”.
- 1071 • In accordance with the rules presented in clause 7.2.3, the top-level entries for annotations
 1072 (those containing the names of the annotations themselves) shall be sorted alphabetically
 1073 together for the initial version of the schema’s dictionary, and shall be appended to the list with

1074 each schema revision. Stated explicitly, the annotations from the properties and
1075 patternProperties shall be comingled together within the entries for each revision of the
1076 dictionary.

- 1077 • Dictionary entries for children properties of annotations, such as the anonymous string value
1078 array entries for @Redfish.AllowableValues shall be structured and formatted per the rules
1079 presented in clause 7.2.3.

1080 7.2.3.5 Registry dictionary format

1081 Redfish messages are used in multiple places, including annotations, events, and errors. The actual
1082 message data may be retrieved from any of the various message registries including standard Redfish
1083 and OEM registries. These messages are referred to by name as the value of a string field in hosting
1084 schemas, so names such as “NetworkDevice.1.0.LinkFlapDetected” appear in BEJ-encoded JSON data
1085 for previous versions of this specification. To reduce the size of such encodings, RDE version 1.1
1086 introduces the notion of a Registry dictionary that can be referenced via the bejRegistryItem encoding
1087 format. Replacing the message name with a sequence number in the Registry dictionary achieves a
1088 reduction in encoded data for messages. This clause details special notes that apply to building the
1089 registry dictionary:

- 1090 • The registry dictionary shall consist of a top-layer set named “registry”
 - 1091 – Entries within the set shall be named for each of the registry items supported by the RDE
1092 Device. The full odata name for these entries shall be incorporated in the dictionary, and
1093 they shall be sorted lexicographically.
 - 1094 – The type of the registry items shall be bejString, and they shall be flagged as read-only.
- 1095 • Both full and truncated registry dictionaries are permitted.
- 1096 • MCs shall not attempt to merge registry dictionaries from different devices or dictionaries
1097 retrieved from the same device at different times.
- 1098 • If using the DMTF dictionary builder tool (see clause 7.2.3.7), see the tool documentation for
1099 information on how to build the registry dictionary for a device.
- 1100 • Schema entries that correspond to registry items shall be encoded in dictionaries as being of
1101 type bejString, not bejRegistryItem. This ensures backward compatibility with earlier versions of
1102 the RDE specification

1103 7.2.3.6 Links between schemas

1104 Links in Redfish schemas, identifiable as entries with Odata type odata.id, shall be represented in
1105 dictionaries as entries with format = bejString. As described in clause 8.4.2, runtime encoding of Odata
1106 links may be performed via any of bejString (with deferred bindings), bejResourceLink, or (for expansion)
1107 bejResourceLinkExpansion. This is a special case wherein a valid encoding may differ from the type
1108 specified in the dictionary.

1109 7.2.3.7 Building dictionaries

1110 Available online at <https://github.com/DMTF/RDE-Dictionary>, the RDE dictionary builder automates the
1111 process of building an RDE dictionary from CSDL formatted schemas.

1112 It supports standard Redfish schemas, standalone OEM schemas, and OEM extensions to standard
1113 Redfish schemas and can build full or truncated dictionaries. For more information about installation,
1114 usage and examples of using the dictionary builder, refer to the README.md file at the above URL.

1115 **7.2.4 Redfish Operation support**

1116 Redfish Operations are sent from a client to a Redfish Provider that is able to process them and respond
 1117 appropriately. These operations are encoded in JSON and transported via either the HTTP or the HTTPS
 1118 protocol.

1119 In this specification, the MC is the Redfish Provider to which the client sends operations. However, rather
 1120 than responding directly, the MC is a proxy that conveys these operations to the RDE Devices that
 1121 maintain the data and can provide responses to client requests. The proxied operations (that are
 1122 transmitted to the RDE Device as RDE Operations) are encoded in BEJ (clause 8) and transported via
 1123 PLDM. The MC, in its role as proxy Redfish Provider for the RDE Devices, translates the JSON/HTTP(S)
 1124 requests from the client into BEJ/PLDM for the RDE Device, and then translates the BEJ/PLDM response
 1125 from the RDE Device into a JSON/HTTP(S) response for the client.

1126 **7.2.4.1 Primary Operations**

1127 There are seven primary Redfish Operations. These are summarized in Table 33.

1128 **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.

1129 The only Redfish Operation that is required to be supported in RDE is Read; however, it is expected that
 1130 implementations will support Update as well. Create and Delete are conditionally required for RDE
 1131 Devices that contain collections; Action is conditionally required for RDE Devices that support Redfish
 1132 schema-defined actions. The Head and Replace Redfish Operations are strictly optional.

1133 **7.2.4.1.1 HTTP/HTTPS and Redfish**

1134 A full discussion of the HTTP/HTTPS protocol is beyond the scope of this specification; however, a
 1135 minimalist overview of key concepts relevant to Redfish Device Enablement follows. Readers are directed
 1136 to [DSP0266](#) for more detailed information on the usage of HTTP and HTTPS with Redfish and to
 1137 standard documentation for more general information on the HTTP/HTTPS protocols themselves.

1138 **7.2.4.1.1.1 Redfish Operation requests**

1139 Every Redfish request has a target URI to which it should be applied; this URI is the target of the
 1140 HTTP/HTTPS verb listed in Table 33. The URI may consist of several parts of interest for purposes of this
 1141 specification: a prefix that points to the RDE Device being managed, a subpath within the RDE Device
 1142 management topology, a specific resource selection preceded by an octothorp character (#), and one or
 1143 more query options preceded by a question mark (?) character.

1144 Many, but not all, Redfish requests have a JSON payload associated with them. For example, a POST
1145 operation to create a new child element in a collection would normally contain a JSON payload for the
1146 data being supplied for that new child element.

1147 Finally, every Redfish HTTP/HTTPS request will contain a series of headers, each of which modifies it in
1148 some fashion.

1149 7.2.4.1.1.2 Redfish Operation responses

1150 The response to a Redfish HTTP/HTTPS request will also contain several elements. First, the response
1151 will contain a status code that represents the result of the operation. Like for requests, [DSP0266](#) defines
1152 several response headers that may need to be supplied in conjunction with a Redfish response. Finally, a
1153 JSON payload may be present such as in the case of a read operation.

1154 7.2.4.1.1.3 Generic handling of Redfish Operations

1155 Generically, to handle processing of a Redfish HTTP/HTTPS request, the MC will typically implement the
1156 following steps. This overview ignores error conditions, timeouts, and long-lived Tasks. A much more
1157 detailed treatment may be found in clause 9.

- 1158 1) Parse the prefix of the supplied URI to pinpoint the RDE Device that the operation targets.
- 1159 2) Parse the RDE Device portion of the URI to identify the specific place in the RDE Device's
1160 management topology targeted by the operation.
- 1161 3) Identify the Redfish Resource PDR that represents that portion of the data.
- 1162 4) Using the HTTP/HTTPS verb and other request information, determine the type of Redfish
1163 operation that the client is trying to perform.
- 1164 5) Translate any request headers (clause 7.2.4.2) and query options (clause 7.2.4.3) into
1165 parameters to the corresponding PLDM request message(s).
- 1166 6) Translate the JSON payload, if present, into a corresponding BEJ (clause 8) payload for the
1167 request, using a dictionary appropriate for the target Redfish Resource PDR.
- 1168 7) Send the PLDM for Redfish Device Enablement RDEOperationInit command (clause 12.1) to
1169 begin the Operation.
- 1170 8) Send any BEJ payload to the RDE Device via one or more PLDM for Redfish Device
1171 Enablement RDEMultipartSend commands (clause 13.1) unless it was small enough to be
1172 inlined in the RDEOperationInit command.
- 1173 9) Send any request parameters to the RDE Device via the PLDM for Redfish Device Enablement
1174 SupplyCustomRequestParameters command (clause 12.2).
- 1175 10) If there was a payload but no request parameters, send the RDEOperationStatus command
1176 (clause 12.5).
- 1177 11) Retrieve and decode any BEJ-encoded JSON data for any Operation response payloads via
1178 one or more PLDM for Redfish Device Enablement RDEMultipartReceive commands (clause
1179 13.2).
- 1180 12) Retrieve any response parameters via the PLDM for Redfish Device Enablement
1181 RetrieveCustomResponseHeaders command (clause 12.3).
- 1182 13) Send the PLDM for Redfish Device Enablement RDEOperationComplete command (clause
1183 12.4) to inform the RDE Device that it may discard any data structures associated with the
1184 Task.
- 1185 14) Translate the BEJ response payload, if present, into JSON format for return to the client, using
1186 an appropriate dictionary.

1187 15) Prepare and send the final response to the client, adding the various HTTP/HTTPS response
 1188 headers (clause 7.2.4.2) appropriate to the type of Redfish operation that was just performed.

1189 **7.2.4.2 Redfish operation headers**

1190 Several HTTP/HTTPS transport layer headers modify Redfish operations when translated in the context
 1191 of RDE Operations. These are summarized in Table 34. Implementation notes for how the MC and RDE
 1192 Device shall support some of these modifiers – when attached to Redfish operations – may be found in
 1193 the indicated subsections. For headers not listed here, the implementation is outside the scope of this
 1194 specification; implementers shall refer to [DSP0266](#) and standard HTTP/HTTPS documentation for more
 1195 information on processing these headers.

1196 **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
Response Headers (continued)			
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.

1197 **7.2.4.2.1 If-Match request header**

1198 The MC shall support the If-Match header when applied to Redfish HTTP/HTTPS PUT and PATCH
 1199 operations; support for other Redfish operations is optional.

1200 The parameter for this header is an ETag.

1201 In order to support this header, the MC shall convey the supplied ETag to the RDE Device via the
1202 ETag[0] field of the PLDM SupplyCustomRequestParameters command (clause 12.2) request message
1203 and supply the value ETAG_IF_MATCH for the ETagOperation field of the same message. For this
1204 header, the MC shall supply the value 1 for the ETagCount field of the request message.

1205 When the RDE Device receives an ETAG_IF_MATCH within the ETagOperation field in the
1206 SupplyCustomRequestParameters command, it shall verify that the ETag matches the current state of the
1207 targeted schema data instance before proceeding with the RDE Operation. In the event of a mismatch, it
1208 shall respond to the SupplyCustomRequestParameters command with completion code
1209 ERROR_ETAG_MATCH.

1210 In the event that both an If-Match and If-None-Match request header are supplied by the client, the MC
1211 shall respond with HTTP status code 400 – Bad Request – to the client and stop processing the request.
1212 The MC shall not send such a malformed request to the RDE Device.

1213 **7.2.4.2.2 If-None-Match request header**

1214 The MC may optionally support the If-None-Match header when applied to Redfish HTTP/HTTPS GET
1215 and HEAD operations.

1216 The parameter for this header is a comma-separated list of ETags.

1217 In order to support this header, the MC shall convey the supplied ETag(s) to the RDE Device via the
1218 ETag[i] fields of the PLDM SupplyCustomRequestParameters command (clause 12.2) request message
1219 and supply the value ETAG_IF_NONE_MATCH for the ETagOperation field of the same message. For
1220 this header, the MC shall supply the value N for the ETagCount field of the request message where N is
1221 the number of entries in the comma-separated list.

1222 When the RDE Device receives an ETAG_IF_NONE_MATCH within the ETagOperation field in the
1223 SupplyCustomRequestParameters command, it shall verify that none of the supplied ETags matches the
1224 current state of the targeted schema data instance before proceeding with the RDE Operation. In the
1225 event of a match, it shall respond to the SupplyCustomRequestParameters command with completion
1226 code ERROR_ETAG_MATCH.

1227 In the event that both an If-Match and If-None-Match request header are supplied by the client, the MC
1228 shall respond with HTTP status code 400 – Bad Request – to the client and stop processing the request.
1229 The MC shall not send such a malformed request to the RDE Device.

1230 **7.2.4.2.3 Custom HTTP headers**

1231 The MC shall support custom headers when applied to any Redfish HTTP/HTTPS operation. For
1232 purposes of this specification, an RDE custom header shall be considered as one with a prefix “PLDM-
1233 RDE-“ and for which no standard handling is described either in this specification or in [DSP0266](#). All
1234 discussion of custom headers in this specification shall be restricted to HTTP/HTTPS custom headers of
1235 this form.

1236 The parameters for custom headers will vary by actual header type.

1237 In order to support RDE custom headers, the MC shall bundle them (including the PLDM-RDE prefix) into
1238 the request message for an invocation of the SupplyCustomRequestParameters command (clause 12.2).
1239 To do so, the MC shall set the HeaderCount request parameter to the number of custom request
1240 parameters. For each RDE custom request parameter n , the MC shall set HeaderName[n] and
1241 HeaderParameter[n] to the name and value of the request parameter, respectively. Custom headers other
1242 than those prefixed “PLDM-RDE-“ shall not be supplied to RDE Devices in this manner.

1243 When the RDE Device receives RDE custom request parameters, it may perform any custom handling for
1244 the parameter. If it does not support a specific RDE custom request parameter received, the RDE Device
1245 shall respond with the ERROR_UNRECOGNIZED_CUSTOM_HEADER completion code.

1246 Similarly, when the RDE Device has custom response parameters to send back to a client, it shall set the
1247 HaveCustomResponseParameters flag in the OperationExecutionFlags response field of the
1248 RDEOperationInit, SupplyCustomRequestParameters, or RDEOperationStatus command to ask the MC
1249 to retrieve these parameters. Then, in response to the RetrieveCustomResponseParameters command
1250 (clause 12.3), the RDE Device shall set the ResponseHeaderCount field to the number of custom
1251 response headers it wants to send back to the client. For each custom response parameter *n*, the RDE
1252 Device shall set HeaderName[*n*] and HeaderParameter[*n*] to the name and value of the response
1253 parameter, respectively.

1254 Following completion of the main Operation, the MC shall check the HaveCustomResponseParameters
1255 flag in the OperationExecutionFlags response field to see if the RDE Device is supplying custom
1256 response headers (which should have an X-PLDM-RDE prefix). If the flag is set (with value 1b), the MC
1257 shall use the RetrieveCustomResponseParameters command (clause 12.3) to recover them from the
1258 RDE Device. The MC shall then append the recovered headers to the Redfish Operation response.

1259 **7.2.4.2.3.1 PLDM-RDE-Expand-Type**

1260 The MC may optionally support use of the PLDM-RDE-Expand-Type header when it receives a Redfish
1261 HTTP/HTTPS GET operation with the \$expand query option (see clause 7.2.4.3.3) to convey the
1262 expansion type parameter to the RDE Device.

1263 The parameter for this header is the type of expansion to be used in the expansion, one of
1264 EXPAND_DOT (“.”), EXPAND_TILDE (“~”), or EXPAND_STAR (“*”) and shall match the parameter given
1265 as the value of the \$expand query option. If this header is not supplied to the RDE Device, expansion
1266 shall default to type EXPAND_DOT. If no expansion type is supplied to the \$expand query option, the MC
1267 may either send this header with the default type (EXPAND_DOT) or omit it.

1268 **7.2.4.2.4 ETag response header**

1269 The MC shall provide an ETag header in response to every Redfish HTTP/HTTPS GET or HEAD
1270 operation.

1271 The parameter for this header is an ETag.

1272 In order to support this header, the RDE Device shall generate a digest of the schema data instance after
1273 each modification to the data in accordance with [RFC 7232](#). When the MC begins a GET or HEAD
1274 operation to the RDE Device via a PLDM RDEOperationInit command (clause 12.1), the RDE Device
1275 shall populate the ETag field in the response message to the command where the RDE Operation has
1276 completed (one of RDEOperationInit, SupplyCustomRequestParameters, or RDEOperationStatus) with
1277 this digest.

1278 When it receives an ETag field in the response message for a completed RDE Operation, the MC shall
1279 then populate this header with the digest it receives.

1280 **7.2.4.2.5 Link response header**

1281 The MC shall provide one or more Link headers in response to every Redfish HTTP/HTTPS GET and
1282 HEAD operation as described in [DSP0266](#).

1283 The parameter for this header is a URI.

1284 This header has three forms as described in [DSP0266](#); all three shall be supported by MCs. The handling
1285 for these three forms is detailed in the next three clauses.

1286 No special action is needed on the part of an RDE Device to support any form of the link response
1287 header.

1288 7.2.4.2.5.1 Schema form

1289 The MC shall provide a link header with “rel=describedby” to provide a schema link for the data that is or
1290 would be returned in response to a Redfish HTTP/HTTPS GET or HEAD operation. The MC may obtain
1291 this link in any of several manners:

- 1292 • An @odata.context annotation in read data may contain the schema reference.
- 1293 • The MC may have the schema reference cached.
- 1294 • The MC may retrieve the schema reference directly from the PDR encapsulating the instance of
1295 the schema data by invoking the PLDM GetSchemaURI command (clause 11.4).

1296 An example of a schema form link header is as follows; readers are referred to [DSP0266](#) for more detail:

1297

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

1298 7.2.4.2.5.2 Annotation form

1299 The MC should provide a link header to provide an annotation link for the data that is or would be
1300 returned in response to a Redfish HTTP/HTTPS GET or HEAD operation. The MC may obtain this link in
1301 any of several manners:

- 1302 • The MC may inspect annotations to determine whether @odata or @Redfish annotations are
1303 used.
- 1304 • The MC may retrieve the schema reference directly from the PDR encapsulating the instance of
1305 the schema data by invoking the PLDM GetSchemaURI command (clause 11.4)

1306 An example of an annotation form link header is as follows; readers are referred to [DSP0266](#) for more
1307 detail:

1308

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

1309 7.2.4.2.5.3 Passthrough form

1310 The MC shall translate link annotations returned from the RDE Device in response to a Redfish
1311 HTTP/HTTPS GET operation into link headers. In this form, the MC shall also include the schema path to
1312 the link.

1313 An example of a passthrough form link header is as follows; readers are referred to [DSP0266](#) for more
1314 detail:

1315

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

1316 7.2.4.2.6 Location response header

1317 The MC shall provide a Location header in response to every Redfish HTTP/HTTPS POST that effects a
1318 successful create operation. The MC shall also provide a Location header in response to every Redfish
1319 Operation that spawns a long-running Task when executed as an RDE Operation.

1320 The parameter for this header is a URI.

1321 In order to support this header for completed create operations, the RDE Device shall populate the
1322 NewResourceID response parameter in the response message for the
1323 RetrieveCustomResponseParameters command (clause 12.3) with the Resource ID of the newly created
1324 collection element. Upon receipt, the MC shall combine this resource ID with the topology information
1325 contained in the Redfish Resource PDRs for the targeted PDR up through the device component root to
1326 create a local URI portion that it shall then combine with its external management URI for the RDE Device
1327 to build a complete URI for the newly added collection element. The MC shall then populate this header
1328 with the resulting URI.

1329 In order to support this header for Redfish Operations that spawn long-running Tasks when executed as
1330 RDE Operations, the MC shall generate a TaskMonitor URL for the Operation and populate the Location
1331 header with the generated URL. See clause 7.2.6 for more details.

1332 **7.2.4.2.7 Cache-Control response header**

1333 The MC shall provide a Cache-Control header in response to every Redfish HTTP/HTTPS GET or HEAD
1334 operation.

1335 In order to support this header for HTTP/HTTPS GET operations, the RDE Device shall mark the
1336 CacheAllowed flag in the OperationExecutionFlags field of the response message for the triggering
1337 command for the read or head Operation with an indication of the caching status of data read.

1338 When the MC reads the CacheAllowed flag in the OperationExecutionFlags field of the response
1339 message for a completed RDE Operation, it shall populate the Cache-Control response header with an
1340 appropriate value. Specifically, if the RDE Device indicates that the data is cacheable, the MC shall
1341 interpret this as equivalent to the value “public” as defined in [RFC 7234](#); otherwise, the MC shall interpret
1342 this as equivalent to the value “no-store” as defined in [RFC 7234](#).

1343 **7.2.4.2.8 Allow response header**

1344 The MC shall provide an Allow header in response to every Redfish HTTP/HTTPS operation that is
1345 rejected by the RDE Device specifically for the reason of being a disallowed operation, giving the
1346 ERROR_NOT_ALLOWED completion code (clause 7.5). The MC shall additionally provide an Allow
1347 response header in response to every GET (or HEAD, if supported) Redfish operation.

1348 In order to support this header, when the RDE Device responds to an RDE command with
1349 ERROR_NOT_ALLOWED, or in response to a GET or HEAD Redfish operation, it shall populate the
1350 PermissionFlags field of its response message with an indication of the operations that are permitted.

1351 When the MC reads the PermissionFlags field of the response message for a completed RDE Operation,
1352 the MC shall populate this header with the supplied information.

1353 **7.2.4.2.9 Retry-After response header**

1354 The MC shall provide a Retry-After header in response to every non-HEAD Redfish Operation that when
1355 conveyed to the RDE Device results in any transient failure (ERROR_NOT_READY; see clause 7.5).

1356 The parameter for this header is the length of time in seconds the client should wait before retrying the
1357 request.

1358 When the RDE Device needs to defer an RDE Operation, it shall return ERROR_NOT_READY in
1359 response to the RDEOperationInit command that begins the Operation. The RDE Device must now
1360 choose whether to supply a specific deferral timeframe or to use the default deferral timeframe. To specify
1361 a specific deferral timeframe, the RDE Device shall also set the HaveCustomResponseParameters flag in
1362 the OperationExecutionFlags response field of the RDEOperationInit command to inform the MC that it
1363 should retrieve deferral information. Then, if it did set the HaveCustomResponseParameters flag, in
1364 response to the RetrieveCustomResponseParameters command (clause 12.3), the RDE Device shall set
1365 the DeferralTimeframe and DeferralUnits parameters appropriately to indicate how long it is requesting
1366 the client to wait before resubmitting the request.

1367 As an alternative to specifying a deferral timeframe via the response message for
1368 RetrieveCustomResponseParameters, the RDE Device may skip setting the
1369 HaveCustomResponseParameters flag in the OperationExecutionFlags response field of the
1370 RDEOperationInit command to request that the MC supply a default deferral timeframe on its behalf.

1371 When it receives the response to the RDEOperationInit command, the MC shall check the
1372 HaveCustomResponseParameters flag in the OperationExecutionFlags response field to see if the RDE

1373 Device has an extended response. If the flag is set (with value 1b), the MC shall use the
 1374 RetrieveCustomResponseParameters command (clause 12.3) to recover the deferral timeframe from the
 1375 DeferralTimeframe and DeferralUnits fields of the response message. If the flag was not set, or if the RDE
 1376 Device supplied an unknown deferral timeframe (0xFF), the MC shall use a default value of 5 seconds. It
 1377 shall then populate this header with the deferral value.

1378 Both the MC and RDE Device shall be prepared for possibility that the client may retry the operation
 1379 before this deferral timeframe elapses: Operations can be re-initiated by impatient end users.

1380 7.2.4.3 Redfish Operation request query options

1381 In addition to HTTP/HTTPS headers, the standard Redfish management protocol defines several query
 1382 options that a client may specify in a URI to narrow the request in Redfish GET Operations. For any query
 1383 option not listed here, the MC may support it in a fashion as described in [DSP0266](#).

1384 **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

1385 Support requirements for query parameters are described in Table 36.

1386

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

1387 **7.2.4.3.1 \$skip query option**

1388 The MC should support \$skip query options when provided as part of a target URI for a Redfish
 1389 HTTP/HTTPS GET operation.

1390 The parameter for this query option is an integer representing the number of members of a resource
 1391 collection to skip over. See [DSP0266](#) for more details on the usage of \$skip.

1392 To support this query option, the MC shall supply the \$skip parameter in the CollectionSkip field of the
 1393 SupplyCustomRequestParameters (clause 12.2) request message. In the event that this query option is
 1394 not supplied as part of the target URI for an HTTP/HTTPS GET operation, the MC shall supply a value of
 1395 zero in this field if it otherwise needs to supply extended request parameters; it shall not send the
 1396 SupplyCustomRequestParameters just to supply a value of zero for the CollectionSkip field.

1397 When processing an RDE read Operation for a resource collection, the RDE Device shall check the
 1398 CollectionSkip parameter from the SupplyCustomRequestParameters request message to determine the
 1399 number of members to skip over in its response, per [DSP0266](#). In the event that the MC did not indicate
 1400 the presence of extended request parameters, the RDE Device shall interpret this as a CollectionSkip
 1401 value of zero. If the parameter for \$skip equals the number of elements in the collection, the RDE Device
 1402 shall return an empty list. If the parameter for \$skip exceeds the number of elements in the collection, the
 1403 RDE Device shall return ERROR_OPERATION_FAILED and, in accordance with the Redfish standard
 1404 [DSP0266](#) respond with an annotation specifying that the value is invalid (see
 1405 QueryParameterOutOfRange in the Redfish base message registry).

1406 **7.2.4.3.2 \$top query option**

1407 The MC should support \$top query options when provided as part of the target URI for a Redfish
 1408 HTTP/HTTPS GET operation.

1409 The parameter for this query option is an integer representing the number of members of a resource
 1410 collection to return. See [DSP0266](#) for more details on the usage of \$top. If the parameter for \$top
 1411 exceeds the remaining number of members in a resource collection, the number returned shall be
 1412 truncated to those remaining. For a \$top value of zero, the response shall consist of an empty list.

1413 To support this query option, the MC shall supply the \$top parameter in the CollectionTop field of the
 1414 SupplyCustomRequestParameters (clause 12.2) request message. In the event that this query option is
 1415 not supplied as part of the target URI for an HTTP/HTTPS GET operation, the MC shall supply a value of
 1416 0xFFFF in this field; it shall not send the SupplyCustomRequestParameters just to supply a value of
 1417 unlimited for the CollectionTop field.

1418 When processing an RDE read Operation for a resource collection, the RDE Device shall check the
 1419 CollectionTop parameter from the SupplyCustomRequestParameters request message to determine the
 1420 number of members to respond with, per [DSP0266](#). The RDE Device shall interpret a value of 0xFFFF as
 1421 indicating that there is no limit to the number of members it should return for the referenced resource

1422 collection. In the event that the MC did not indicate the presence of extended request parameters, the
1423 RDE Device shall interpret this as a CollectionTop value of unlimited.

1424 **7.2.4.3.3 \$expand query option**

1425 The MC should support \$expand query options when provided as part of the target URI for a Redfish
1426 HTTP/HTTPS GET operation.

1427 The parameter for this query option is a string representing the links (Navigation properties) to expand in
1428 place, “gluing together” the results of multiple reads into a single JSON response payload. This parameter
1429 may be an absolute string specifying the exact link to be expanded, or it may be any of three wildcards.
1430 The first wildcard, an asterisk (*), means that all links should be expanded. The second wildcard, a dot (.),
1431 means that subordinate links (those that are directly referenced i.e., not in the Links Property section of
1432 the resource) should be expanded. The third wildcard, a tilde (~), means that dependent links (those that
1433 are not directly referenced i.e., in the Links Property section of the resource) should be expanded. See
1434 [DSP0266](#) for more details on the usage of \$expand.

1435 To support an expansion type wildcard received from the Redfish Client, the MC should send the PLDM-
1436 RDE-Expand-Type custom header described in clause 7.2.4.2.3.1 to the RDE Device via the
1437 SupplyCustomRequestParameters command of clause 12.2.

1438 If the \$levels query option qualifier is not present in conjunction with the \$expand query option, the MC
1439 shall treat this as equivalent to \$levels=1.

1440 To support the \$expand query option, the RDE Device should concatenate linked resource data into the
1441 BEJ data it returns for an RDE read Operation, using the bejResourceLinkExpansion PLDM data type
1442 described in Clause 5.3.23.

1443 **7.2.4.3.4 \$levels query option qualifier**

1444 The MC should support the \$levels qualifier to the \$expand query option when provided as part of the
1445 target URI for a Redfish HTTP/HTTPS GET operation or when provided implicitly by having \$expand
1446 provided as part of a Redfish HTTP/HTTPS GET operation without having the \$levels query option
1447 qualifier supplied.

1448 The parameter for this query option is an integer representing the number of schema links to expand into.
1449 If no \$level qualifier is present, the MC shall interpret this as equivalent to \$levels=1.

1450 To support this parameter, the MC can select between two choices: passing it on to the RDE Device or
1451 supporting it itself. The method by which this choice is made is implementation-specific and out of scope
1452 for this specification. If the RDE Device indicates that it cannot support \$levels expansion by setting the
1453 expand_support bit to zero in the DeviceCapabilitiesFlags in the response message to the
1454 NegotiateRedfishParameters command (clause 11.1), or if the expansion type is not “All Links” (see
1455 clause 7.2.4.3.3), the MC shall not select passing it to the RDE Device.

1456 If the MC chooses to pass this query option to the RDE Device, it shall transmit the supplied value to the
1457 RDE Device via the SupplyCustomRequestParameters command in the LinkExpand parameter.

1458 If the MC chooses to handle this query option itself, it shall recursively issue reads to “expand out” data
1459 for links embedded in data it reads. Such links may be identified during the BEJ decode process as tuples
1460 with a format of bejResourceLink (clause 5.3.21). The corresponding value of the node represents the
1461 Resource ID for the Redfish Resource PDR representing the data to embed within the structure of data
1462 already read. The \$levels qualifier dictates the depth of recursion for this process.

1463 When the RDE Device receives a LinkExpand value of greater than zero in extended request parameters
1464 as part of an RDE read operation, it shall “expand out” all resource links (as defined in [DSP0266](#)) to the
1465 indicated depth by encoding them as bejResourceLinkExpansions in the response BEJ data for the
1466 command. If the RDE Device previously did not set the expand_support flag in the

1467 DeviceCapabilitiesFlags field of the NegotiateRedfishParameters command, it may instead ignore the
1468 value (treating it as zero).

1469 Implementers should refer to [DSP0266](#) for more details and caveats to be applied when expanding links
1470 with \$levels > 1.

1471 **7.2.4.3.5 \$select query option qualifier**

1472 The MC may support \$select as a qualifier to the \$expand query option or as a standalone query option,
1473 provided in either case as part of the target URI for a Redfish HTTP/HTTPS GET operation.

1474 The parameter for this query option is a string containing a comma-separated list of properties to be
1475 retrieved from the GET operation; the caller is asking that all other properties be suppressed. See
1476 [DSP0266](#) for more details on the usage of \$select.

1477 If it supports this parameter, the MC should perform the GET operation normally up to the point of
1478 retrieving BEJ-formatted data from the RDE Device. When decoding the BEJ data, however, the MC
1479 should silently discard any property not part of the \$select list.

1480 No action is needed on the part of an RDE Device to support this query option.

1481 **7.2.4.3.6 Excerpt query option**

1482 The MC may support the excerpt query option when provided as part of a target URI for a Redfish
1483 HTTP/HTTPS GET operation. There is no parameter for this command.

1484 To support this parameter, the MC shall set the excerpt_flag in the OperationFlags field of the
1485 RDEOperationInit request command. Thereafter, no special treatment is required on the part of the MC.

1486 When the RDE Device is flagged that the client requested an excerpt, it may support the request by
1487 restricting properties returned in the read to those flagged with the excerpt schema annotation. If the
1488 schema does not contain any such flagged properties, or if the RDE Device does not support the excerpt
1489 query option, it shall return the complete resource.

1490 Further details of the excerpt query option may be found in [DSP0266](#).

1491 **7.2.4.4 HTTP/HTTPS status codes**

1492 The MC shall comply with [DSP0266](#) in all matters pertaining to the HTTP/HTTPS status codes returned
1493 for Redfish GET, PATCH, PUT, POST, DELETE, and HEAD operations. Typical status codes for
1494 operational errors may be found in clause 7.5.

1495 **7.2.4.5 Multihosting and Operations**

1496 A single RDE Device may find that it is attached to multiple MCs. This can introduce complications from
1497 concurrency if conflicting Operations are issued and requires an RDE Device to decide whether an
1498 Operation should be visible to an MC other than the one that issued it. Support for multiple MCs is out of
1499 scope for this specification. In particular, the behavior of the RDE Device in the face of concurrent
1500 commands from multiple MCs is undefined.

1501 **7.2.5 PLDM RDE Events**

1502 An Event is an abstract representation of any happening that transpires in the context of the RDE Device,
1503 particularly one that is outside of the normal command request/response sequence. A Redfish Message
1504 Event consists of JSON data that includes elements such as the index of a standardized text string and a
1505 collection of parameters that provide clarification of the specifics of the Event that has transpired. The full
1506 schema for Events may be found in the standard Redfish Message schema; additionally, OEM extensions
1507 to this schema are possible.

1508 In this specification, a second class of events, Task Executed Events, allow RDE Devices to report that a
1509 Task has finished executing and that the MC should retrieve Operation results. The data for these events
1510 includes elements such as the Operation identifier and the resource with which the Operation is
1511 associated.

1512 As with any other PLDM eventing, the RDE Device advertises that it supports Events by listing support for
1513 the PLDM for Platform Monitoring and Control SetEventReceiver command (see [DSP0248](#)). The MC, for
1514 its part, may then select between two methods by which it will know that Events are available. If the MC
1515 configured the RDE Device to use asynchronous events through the SetEventReceiver command, the
1516 RDE Device shall use the PLDM for Platform Monitoring and Control PlatformEventMessage command
1517 (see [DSP0248](#)) to inform the MC by sending the Event directly. Otherwise, the RDE Device can be
1518 configured to polling mode using the same SetEventReceiver command. The MC uses the PLDM for
1519 Platform Monitoring and Control PollForPlatformEventMessage command (see [DSP0248](#)) for this
1520 purpose. The selection of any polling interval is determined by the MC and is outside the scope of this
1521 specification.

1522 Whether retrieved synchronously or asynchronously, once the MC gets the Event, it may process it.
1523 Redfish Message Events are packaged using the redfishMessageEvent eventClass; Task Executed
1524 Events are packaged using the redfishTaskExecutedEvent eventClass (see [DSP0248](#) for both
1525 eventClasses).

1526 Handling of Task Executed Events is described with Tasks in clause 7.2.6. For Redfish Message Events,
1527 the MC may decode the BEJ-formatted payload of Event data using the appropriate Event schema
1528 dictionary specific to the PDR from which the message was sent.

1529 For a more detailed view of the Event lifecycle, see clause 9.3.

1530 NOTE Events are optional in standard Redfish; however, support for Task Executed Events is mandatory in this
1531 specification if the RDE Device supports asynchronous execution for long-running Operations.

1532 7.2.5.1 [MC] Event subscriptions

1533 In Redfish, a client may request to be notified whenever a Redfish Event occurs. Per [DSP0266](#), to do so,
1534 the client uses a Redfish CREATE operation to add a record to the EventSubscription collection. This
1535 record in turn contains information on the various Event types that the client wishes to receive Events for.
1536 To unsubscribe, the client uses a Redfish DELETE operation to remove its record. Among other
1537 properties, the EventSubscription record contains a URI to which the Event should be forwarded. MCs
1538 that support Events shall support at least one Redfish event subscription.

1539 Event types are global across all schemas; there is no provision at this time ([DSP0266](#) v1.6) in Redfish
1540 for a client to subscribe to just one schema at a time. Further, there is generally no capacity for an RDE
1541 Device to send an HTTP/HTTPS record directly to an external recipient. Events are optional in Redfish;
1542 however, if the MC chooses to provide Event subscription support, it must comply with the following
1543 requirements:

- 1544 • The MC shall provide full support for the EventSubscription collection as a Redfish Provider per
1545 [DSP0266](#).
- 1546 • When it receives an Event subscription request (in the form of a Redfish CREATE operation on
1547 the EventSubscription collection), the MC shall parse the EventTypes array property of the
1548 request to identify the type or types of Events the client is interested in receiving
- 1549 • When the MC receives a Redfish Message Event from an RDE Device, it shall check the
1550 EventType of the Event received against the desired EventTypes for each active client. For
1551 each match, the MC shall forward the Event (translating any @Message.ExtendedInfo
1552 annotations, of course, from BEJ to JSON) to the client as a standard Redfish Provider for the
1553 Event service.

1554 7.2.6 Task support

1555 In PLDM for Redfish Device Enablement, every Redfish HTTP/HTTPS operation is effected as an RDE
1556 Operation. Most Operations, once sent to the RDE Device for execution, may be executed quickly and
1557 the results sent directly in the response message to the request message that triggered them.

1558 It may however transpire that in order for an RDE Device to complete an Operation, it requires more time
1559 than the available window within which the RDE Device is required to send a response. In this case, the
1560 RDE Device has two possible paths to follow. If the current number of extant Tasks is less than the RDE
1561 Device/MC capability intersection (as determined from the call to NegotiateRedfishParameters; see
1562 clause 11.1), the RDE Device shall mark the Operation as a long-running Task and execute it
1563 asynchronously. Otherwise, the RDE Device shall return `ERROR_CANNOT_CREATE_OPERATION` in
1564 its response message to indicate that no new Task slots are available (see clause 7.5).

1565 While the internal data structures used by an RDE Device to manage an Operation are outside the scope
1566 of this specification, they should include at a minimum the `rdeOpID` assigned (usually by the MC) when
1567 the Operation was first created. This allows the MC to reference the Task in subsequent commands to kill
1568 it (`RDEOperationKill`, clause 12.6) or query its status (`RDEOperationStatus`, clause 12.5).

1569 For its part, the MC shall provide full support for the Task collection as a Redfish Provider per [DSP0266](#).
1570 When the MC finds that an Operation has spawned a Task, it shall perform the following steps in order to
1571 comply with the requirements of [DSP0266](#):

- 1572 1) The MC shall instantiate a new TaskMonitor URL and a new member of the Task collection. The
1573 TaskMonitor URL should incorporate or reference (such as via a lookup table) the following data so
1574 that it can map from the TaskMonitor URL back to the correct Redfish resource – and thus the
1575 correct dictionary – for providing status query updates:
 - 1576 a) The `ResourceID` for the resource to which the RDE Operation was targeted
 - 1577 b) The `rdeOpID` for the Operation itself
- 1578 2) The MC shall return response code 202, Accepted, to the client and include the Location response
1579 header populated with the TaskMonitor URL.
- 1580 3) In response to a subsequent Redfish GET Operation applied to the TaskMonitor URL or to the Task
1581 collection member, the MC shall invoke the `RDEOperationStatus` (see clause 12.5) command to
1582 obtain the latest status for the Operation and communicate it to the client in accordance with
1583 [DSP0266](#). If the GET was applied to a TaskMonitor URL and the Operation has been completed, the
1584 MC shall supply the completed results to the client.
 - 1585 a) If the result of the `RDEOperationStatus` command was that the Operation has finished
1586 execution, the MC shall delete both the TaskMonitor URL and the Task collection member
1587 associated with the Operation.
- 1588 4) In response to a Redfish DELETE Operation applied to the TaskMonitor URL or to the Task
1589 collection member, the MC shall attempt to abort the associated Operation via the `RDEOperationKill`
1590 (see clause 12.6) command. It shall then remove both the TaskMonitor URL and the Task collection
1591 member.
- 1592 5) If the RDE Operation finishes before the client polls the TaskMonitor URL, the MC may collect and
1593 store the results of the Operation.
 - 1594 a) In accordance with [DSP0266](#), the MC should retain Operation results until the client retrieves
1595 them. It may refuse to accept further Operations until previous results have been claimed.
 - 1596 b) If the client attempts to collect Operation results after the MC has discarded them, the MC shall
1597 respond with an error HTTP status code as defined in [DSP0266](#).

1598 When the RDE Device finishes execution of a Task, it generates a Task Executed Event to inform the MC
1599 of this status change. The MC can then retrieve the results (via `RDEOperationStatus`) and eventually

1600 forward them to the client. To mark the Task as complete and allow the RDE Device to discard any
 1601 internal data structures used to manage the Task, the MC shall call RDEOperationComplete (clause
 1602 12.4).

1603 For a more detailed overview of the Operation/Task lifecycle from the MC's perspective, see clause
 1604 7.2.4.1.1.3. A detailed flowchart of the Operation/Task lifecycle may be found in clause 9.2.1.4, and a
 1605 finite state machine for the Task lifecycle (from the RDE Device's perspective) may be found in clause
 1606 9.2.3.

1607 7.3 Type code

1608 Refer to [DSP0245](#) for a list of PLDM Type Codes in use. This specification uses the PLDM Type Code
 1609 000110b as defined in [DSP0245](#).

1610 7.4 Transport protocol type supported

1611 PLDM can support bindings over multiple interfaces; refer to [DSP0245](#) for the complete list. All transport
 1612 protocol types can be supported for the commands defined in Table 50.

1613 7.5 Error completion codes

1614 Table 37 lists PLDM completion codes for Redfish Device Enablement. The usage of individual error
 1615 completion codes is defined within each of the PLDM command clauses. When communicating results
 1616 back to the client, implementations should provide HTTP error codes as described below.

1617 **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.	500 Internal Server Error
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

Value	Name	Description	HTTP Error Code
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
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 X-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

1618 HTTP Error codes returned when Operations complete with standard PLDM completion codes should be
 1619 as follows:

1620

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
ERROR_INVALID_PLDM_TYPE	Not a supported PLDM type	501 Not Implemented

1621 7.6 Timing specification

1622 Table 39 below defines timing values that are specific to this document. The table below defines the
 1623 timing parameters defined for the PLDM Redfish Specification. In addition, all timing parameters listed in
 1624 [DSP0240](#) for command timeouts, command response times, and number of retries shall also be followed.

1625

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	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. 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.

1626 8 Binary Encoded JSON (BEJ)

1627 This clause defines a binary encoding of Redfish JSON data that will be used for communicating with
1628 RDE Devices. At its core, BEJ is a self-describing binary format for hierarchical data that is designed to
1629 be straightforward for both encoding and decoding. Unlike in ASN.1, BEJ uses no contextual encodings;
1630 everything is explicit and direct. While this requires the insertion of a bit more metadata into BEJ encoded
1631 data, the tradeoff benefit is that no lookahead is required in the decoding process. The result is a
1632 significantly streamlined representation that fits in a very small memory footprint suitable for modern
1633 embedded processors.

1634 8.1 BEJ design principles

1635 The core design principles for BEJ are focused around it being a compact binary representation of JSON
1636 that is easy for low-power embedded processors to encode, decode, and manipulate. This is important
1637 because these ASICs typically have highly limited memory and power budgets; they must be able to
1638 process data quickly and efficiently. Naturally, it must be possible to fully reconstruct a textual JSON
1639 message from its BEJ encoding.

1640 The following design principles guided the development of BEJ:

- 1641 1) It must be possible to support full expressive range of JSON.
- 1642 2) The encoding should be binary and compact, with as much of the encoding as possible
1643 dedicated to the JSON data elements. The amount of space afforded to metadata that conveys
1644 elements such as type format and hierarchy information should be carefully limited.
- 1645 3) There is no need to support multiple encoding techniques for one type of data; there is therefore
1646 no need to distinguish which encoding technique is in use.
- 1647 4) Schema information – such as the names of data items – does not need to be encoded into BEJ
1648 because the recipient can use a prior knowledge of the data organization to determine semantic
1649 information about the encoded data. In contrast to JSON, which is unordered, BEJ must adopt
1650 an explicit ordering for its data to support this goal.
- 1651 5) The need for contextual awareness should be minimized in the encoding and decoding process.
1652 Supporting context requires extra lookup tables (read: more memory) and delays processing
1653 time. Everything should be immediately present and directly decodable. Giving up a few bytes
1654 of compactness in support of this goal is a worthwhile tradeoff.

1655 8.2 SFLV tuples

1656 Each piece of JSON data is encoded as a tuple of PLDM type bejTuple and consists of the following:

- 1657 1) Sequence number: the index within the canonical schema at the current hierarchy level for the
1658 datum. For collections and arrays, the sequence number is the 0-based array index of the
1659 current element.
- 1660 2) Format: the type of data that is encoded.
- 1661 3) Length: the length in bytes of the data.
- 1662 4) Value: the actual data, encoded in a format-specific manner.

1663 These tuple elements collectively describe a single piece of JSON data; each piece of JSON data is
1664 described by a separate tuple. Requirements for each tuple element are detailed in the following clauses.

1665 SFLV tuples are represented by elements of the bejTuple PLDM type defined in clause 5.3.5.

1666 **8.2.1 Sequence number**

1667 The Sequence Number tuple field serves as a stand-in for the JSON property name assigned to the data
 1668 element the tuple encodes. Sequence numbers align to name strings contained within the dictionary for a
 1669 given schema. Sequence numbers are represented by elements of the bejTupleS PLDM type defined in
 1670 clause 5.3.6.

1671 The low-order bit of a sequence number shall indicate the dictionary to which it belongs according to the
 1672 following table:

1673 **Table 40 – Sequence number dictionary indication**

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

1674 **8.2.2 Format**

1675 The Format tuple field specifies the kind of data element that the tuple is representing.

1676 Formats are represented by elements of the bejTupleF PLDM type defined in clause 5.3.7.

1677 **8.2.3 Length**

1678 The Length tuple field details the length in bytes of the contents of the Value tuple field.

1679 Lengths are represented by elements of the bejTupleL PLDM type defined in clause 5.3.8.

1680 **8.2.4 Value**

1681 The Value tuple field contains an encoding of the actual data value for the JSON element described by
 1682 this tuple. The format of the value tuple field is variable but follows directly from the format code in the
 1683 Format tuple field.

1684 The following JSON data types are supported in BEJ:

1685 **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

BEJ Type	JSON Type	Description
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 property@annotation
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

1686 If the deferred_binding flag (see the bejTupleF PLDM type definition in clause 5.3.7) is set, the string
 1687 encoded in the value tuple element contains substitution macros that the MC is to supply on behalf of the
 1688 RDE Device when populating a message to send back to the client. See clause 8.3 for more details.

1689 Values are represented by elements of the bejTupleV PLDM type defined in clause 5.3.9.

1690 8.3 Deferred binding of data

1691 The data returned to a client from a Redfish operation typically contains annotation metadata that specify
 1692 URIs and other bits of information that are assigned by the MC when it performs RDE Device discovery
 1693 and registration. In practice, the only way for an RDE Device to know the values for these annotations
 1694 would be for it to somehow query the MC about them. Instead, we define substitution macros that the
 1695 RDE Device may use to ask the MC to supply these bits of information on its behalf. RDE Devices shall
 1696 not invoke substitution macros for information that they know and can provide themselves.

1697 All substitution macros are bracketed with the percent sign (%) character. While it would in theory be
 1698 possible for the MC to check every string it decodes for the presence of this escape character, in practice
 1699 that would be an inefficient waste of MC processing time. Instead, the RDE Device shall flag any string
 1700 containing substitution macros with the deferred binding bit set to inform the MC of their presence; the
 1701 MC shall only perform macro substitution if the deferred binding bit is set. The MC shall support the
 1702 deferred bindings listed in Table 42.

1703 **Table 42 – BEJ deferred binding substitution parameters**

Macro	Data to be substituted	Example substitutions
%%	A single % character	%
%L<resource-ID>	The MC-assigned URI of an RDE Provider defined resource (specified by a resource ID within the target PDR), or /invalid.PDR<resource-ID> if unrecognized resource ID	/invalid.PDR123

Macro	Data to be substituted	Example substitutions
%P<resource-ID>.PAGE<pagination-offset>	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 /invalid.PDR<resource-ID>.PAGE<pagination-offset> if unrecognized resource ID or pagination offset < 1	/invalid.PDR101.PAGE-1
%PD	The MC-assigned URI for an MC-managed PCIeDevice.PCIeDevice correlating to this RDE Device, or /invalid.PCIeDevice if one cannot be identified. If the RDE Device manages its own PCIeDevice.PCIeDevice resource, it shall use the %L binding when referring to it,	/invalid.PCIeDevice
%PF<function-info>	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. Function-info shall be a string of lower-case hexadecimal digits corresponding to the PCIe function number for the function. If the RDE Device manages its own PCIeFunction.PCIeFunction resource, it shall use the %L binding when referring to it,	/invalid.PCIeFunction.nonhexdigits /redfish/v1/chassis/1/PCIeDevices/ NIC/PCIeFunctions/1
%PI	The MC-assigned URI for an MC-managed PCIeDevice.PCIeInterface correlating to this RDE Device, or /invalid.PCIeInterface if one cannot be identified. If the RDE Device manages its own PCIeDevice.PCIeInterface resource, it shall use the %L binding when referring to it	/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

Macro	Data to be substituted	Example substitutions
%l<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

1704 **8.4 BEJ encoding**

1705 This clause presents implementation considerations for the BEJ encoding process. For standard resource
 1706 encoding (as opposed to annotations), the BEJ conversion dictionary is built to encode the same
 1707 hierarchical data format as the schema itself. Implementations should therefore track their context inside
 1708 the dictionary in parallel with tracking their location in the data to be encoded. While not mandatory, a
 1709 recursive implementation will prove in most cases to be the easiest approach to realize this tracking.

1710 Like with JSON encodings of data, there is no defined ordering for properties in BEJ data; encoders are
 1711 therefore free to encode properties in any order.

1712 **8.4.1 Conversion of JSON data types to BEJ**

1713 Recognition of [JSON](#) data types enables them to be encoded properly. In Redfish, every property is
 1714 encoded in the format "property_name" : property_value. Whitespace between syntactic elements is
 1715 ignored in JSON encodings.

1716 **8.4.1.1 JSON objects**

1717 A JSON object consists of an opening curly brace ('{'), zero or more comma-separated properties, and
 1718 then a closing curly brace ('}'). JSON objects shall be encoded as BEJ sets with the properties inside the
 1719 curly braces encoded recursively as the value tuple contents of the BEJ set. Following the precedent
 1720 established in JSON, the properties contained within a JSON object may be encoded in BEJ in any order.
 1721 In particular, the encoding order for a collection of properties is not required to match their respective
 1722 sequence numbers.

1723 **8.4.1.2 JSON arrays**

1724 A JSON array consists of an opening square brace ('['), zero or more comma-separated JSON values all
 1725 of a common data type (typically objects in Redfish), and then a closing square brace. JSON arrays shall
 1726 be encoded as BEJ arrays with the data inside the square braces encoded recursively as instances of the
 1727 value tuple contents of the BEJ array. The immediate contents of a JSON array shall be encoded in order
 1728 corresponding to their array indices.

1729 The sequence numbers for BEJ array immediate child elements shall match the zero-based array index
 1730 of the children. These sequence numbers are not represented in the dictionary; it is the responsibility of a
 1731 BEJ encoder/decoder to understand that this is how array data instances are handled.

1732 **8.4.1.3 JSON numbers**

1733 In JSON, there is no distinction between integer and real data; both are collected together as the number
 1734 type. For BEJ, numeric data shall be encoded as a BEJ integer if it contains neither a decimal point nor
 1735 an exponentiation marker ('e' or 'E') and as a BEJ real otherwise.

1736 **8.4.1.4 JSON strings**

1737 When converting JSON strings to BEJ format, a null terminator shall be appended to the string.

1738 **8.4.1.5 JSON Boolean**

1739 In JSON, Boolean data consists of one of the two sentinels "true" or "false". These sentinels shall be
 1740 encoded as BEJ Boolean data with an appropriate value field.

1741 **8.4.1.6 JSON null**

1742 In JSON, null data consists of the sentinel "null". This sentinel shall be encoded as BEJ Null data only if
 1743 the datatype for the property in the schema is null. For a nullable property (identified via the third tag bit
 1744 from the dictionary entry or by the schema), null data shall be encoded as its standard type (from the
 1745 dictionary) with length zero and no value tuple element.

1746 **8.4.2 Resource links**

1747 Most Redfish schemas contain links to other schemas within their properties, formatted as @odata.id
 1748 annotations. When encoding these links in BEJ, the URI may be encoded as any of bejString,
 1749 bejResourceLink, or bejResourceLinkExpansion. If encoded as a bejString, deferred binding substitutions
 1750 may be employed as needed to complete the reference.

1751 **8.4.3 Registry items**

1752 Redfish messages contain items from collated collections called registries. When encoding Redfish
 1753 message registries in BEJ, the string may be encoded as either bejString or bejRegistryItem.

1754 **8.4.4 Annotations**

1755 Redfish annotations may be recognized as properties with a name string containing the "at" sign ('@').
 1756 Several annotations are defined in Redfish, including some that are mandatory for inclusion with any
 1757 Redfish GET Operation. The RDE Device is responsible for ensuring that these mandatory annotations
 1758 are included in the results of an RDE read Operation.

1759 Annotations in Redfish have two forms:

- 1760 • Standalone form annotations have the form "@annotation_class.annotation_name" :
 1761 annotation_value.
 - 1762 – Example: "@odata.id": "/redfish/v1/Systems/1/"
 - 1763 – Standalone annotations shall be encoded with the BEJ data type listed in the annotation
 1764 dictionary in the row matching the annotation name string
- 1765 • Property annotation form annotations have the form
 1766 "property@annotation_class.annotation_name" : annotation_value.
 - 1767 – Example: "ResetType@Redfish.AllowableValues" : ["On", "PushPowerButton"]
 - 1768 – Property annotation form annotations shall be encoded with the BEJ Property Annotation
 1769 data type; the annotation value shall be encoded as a dependent child of the annotation
 1770 entry. See clause 5.3.20.

1771 NOTE Unlike major schema resource properties, annotations have a flat namespace from which sequence numbers
 1772 are drawn. To identify the sequence number for an annotation, an encoder should start at the root of the annotation
 1773 dictionary and then find the string matching the annotation name (including the '@' sign and the annotation source)
 1774 within this set. In particular, the sequence number for an annotation is independent of the current encoding context.

1775 Special handling is required when the RDE Device sends a message annotation to the MC. The related
 1776 properties property inside the annotation's data structure is formatted as an array of strings, but the RDE
 1777 Device has only sequence numbers to work with: the RDE Device may not be able to supply the property
 1778 name for the sequence number. If the RDE Device knows the name of the related property that is
 1779 relevant for the message annotation, it may supply the name directly as an array element. Otherwise, it
 1780 shall encode into the array element a BEJ locator by concatenating the following string components:

1781

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

1782 **8.4.4.1 Nested annotations**

1783 The data format for an annotation may be a simple property such as a string or an integer, but it may also
 1784 be a compound property such as a set. In this latter case it is further possible that the set can itself
 1785 contain an annotation. To distinguish the case where the sequence number for annotation data refers
 1786 anew to a top-level annotation instead of to a property within the set for an annotation, the format byte of
 1787 the BEJ tuple for that annotation shall have the read_only_property_and_top_level_annotation bit set to
 1788 1b.

1789 **8.4.5 Choice encoding for properties that support multiple data types**

1790 If the encoder finds a property that is listed in the dictionary as being of type BEJ choice, it shall encode
1791 the property with type bejChoice in the BEJ format tuple element. The actual value and selected data type
1792 shall be encoded as a dependent child of the tuple containing the bejChoice element. See clauses 5.3.19
1793 and 7.2.3.3.

1794 **8.4.6 Properties with invalid values**

1795 If the MC is encoding an update request from a client that includes a property value that does not match a
1796 required data type according to the dictionary it is translating from, the MC shall in accordance with the
1797 Redfish standard [DSP0266](#) respond to the client with HTTP status code 400 and a
1798 @Message.ExtendedInfo annotation specifying the property with the value format error (see
1799 PropertyValueFormatError, PropertyValueTypeError in the Redfish base message registry). Similarly, if
1800 the value supplied for a property such as an enumeration does not match any required values, the MC
1801 shall in accordance with the Redfish standard [DSP0266](#) respond to the client with HTTP status code 400
1802 and a @Message.ExtendedInfo annotation specifying the property with a value not in the accepted list
1803 (see PropertyValueNotInList in the Redfish base message registry).

1804 **8.4.7 Properties missing from dictionaries**

1805 When encoding JSON data, an encoder may find that the name of a property does not correspond to a
1806 string found in the dictionary. If the encoder is the RDE Device, this should never happen as the RDE
1807 Device is responsible for the dictionary. This situation therefore represents a non-compliant RDE
1808 implementation.

1809 If the MC finds that a property does not correspond to a string found in the dictionary from an RDE
1810 Device, it should in accordance with the Redfish standard [DSP0266](#) respond to the client with HTTP
1811 status code 200 or 400 and an annotation specifying the property as unsupported (see PropertyUnknown
1812 in the Redfish base message registry). The MC may continue to process the client request.

1813 **8.5 BEJ decoding**

1814 This clause presents implementation considerations for the BEJ decoding process.

1815 Properties in BEJ data may be encoded in any order. Decoders must therefore be prepared to accept
1816 data in whatever order it was encoded.

1817 **8.5.1 Conversion of BEJ data types to JSON**

1818 When decoding from BEJ to JSON, the following rules shall be followed. In each of the following,
1819 “property_name” shall be taken to mean the name of the property or annotation as decoded from the
1820 relevant dictionary. For all data types, if the length tuple field is zero, the data shall be decoded as
1821 follows:

1822 “property_name” : null

1823 When multiple properties appear sequentially within a set, they shall be delimited with commas.

1824 **8.5.1.1 BEJ Set**

1825 A BEJ Set shall be decoded to the following format, with the text inside angle brackets (‘<’, ‘>’) replaced as
1826 indicated:

1827 “property_name” : { <set dependent children decoded individually as a comma-separated list > }

1828 8.5.1.2 BEJ Array

1829 A BEJ Array shall be decoded to the following format, with the text inside angle brackets ('<', '>') replaced
1830 as indicated:

1831 "property_name" : [<array dependent children decoded individually as a comma-separated list>]

1832 8.5.1.3 BEJ Integer and BEJ Real

1833 BEJ Integers and BEJ Reals shall be decoded to the following format, with the text inside angle brackets
1834 ('<', '>') replaced as indicated:

1835 "property_name" : "<decoded numeric value>"

1836 8.5.1.4 BEJ String

1837 BEJ Strings shall be decoded to the following format, with the text inside angle brackets ('<', '>') replaced
1838 as indicated. When converting BEJ strings to JSON format, the null terminator shall be dropped as JSON
1839 string encodings do not include null terminators.

1840 "property_name" : "<decoded string value>"

1841 8.5.1.5 BEJ Boolean

1842 BEJ Booleans shall be decoded to the following format, with the text inside angle brackets ('<', '>')
1843 replaced as indicated (note that the "true" and "false" sentinels are not encased in quote marks):

1844 "property_name" : <true or false, depending on the decoded value>

1845 8.5.1.6 BEJ Null

1846 BEJ Null shall be decoded to the following format:

1847 "property_name" : null

1848 8.5.1.7 BEJ Resource Link

1849 A BEJ Resource Link shall be decoded to the following format, with the text inside angle brackets ('<', '>')
1850 replaced as indicated.

1851 "property_name" : "<URI for the resource corresponding the Redfish Resource PDR with the
1852 supplied ResourceID>"

1853 MCs shall be aware that either a BEJ Resource Link or a BEJ Resource Link Expansion may be encoded
1854 for a dictionary entry that lists its type as BEJ Resource Link.

1855 8.5.1.8 BEJ Resource Link expansion

1856 A BEJ Resource Link Expansion shall be decoded to the following format, with the text inside angle
1857 brackets ('<', '>') replaced as indicated.

1858 <full resource data for the Redfish Resource PDR corresponding to the supplied ResourceID>

1859 NOTE property_name is not included in the decoded JSON output in this case.

1860 If the supplied ResourceID is zero and the parent resource is a collection, the MC shall use the
1861 COLLECTION_MEMBER_TYPE schema dictionary obtained from the collection resource (rather than
1862 trying to use a dictionary from the members) to decode resource data.

1863 MCs shall be aware that either a BEJ Resource Link or a BEJ Resource Link Expansion may be encoded
1864 for a dictionary entry that lists its type as BEJ Resource Link.

1865 8.5.2 Annotations

1866 This clause documents the approach for decoding the two types of Redfish annotations to JSON text.

1867 8.5.2.1 Standalone annotations

1868 Standalone annotations (data from decoded from the annotation dictionary) shall be decoded to the
1869 following format, with the bit inside angle brackets ('<', '>') replaced as indicated:

1870 "@annotation_class.annotation_name" : "<decoded annotation value>"

1871 8.5.2.2 BEJ property annotations

1872 BEJ Property Annotations shall be decoded to the following format, with the bit inside angle brackets ('<',
1873 '>') replaced as indicated:

1874 "property_name@annotation_class.annotation_name" : "<decoded annotation value from the
1875 annotation's dependent child node>"

1876 8.5.2.3 [MC] Related Properties in message annotations

1877 When a message annotation is sent from the RDE Device to the MC, the related properties field of
1878 message annotations requires special handling in RDE. Specifically, the array element string values are
1879 BEJ locators to individual properties, may be encoded as a colon-delimited string (see clause 8.4.3).
1880 When decoding, the MC shall check the first character of the supplied string. If it is a colon (:), the MC
1881 shall extract the individual sequence numbers for the BEJ locator, and then use them to identify the
1882 property name to send back to the client for the annotation. If the first character of the supplied string is
1883 not a colon, the MC shall return the supplied string unmodified.

1884 8.5.3 Sequence numbers missing from dictionaries

1885 It may transpire that when decoding BEJ data, a decoder finds a sequence number not in its dictionary.
1886 The handling of this case differs between the RDE Device and the MC.

1887 If the RDE Device finds an unrecognized sequence number as part of the payload for a put, patch, or
1888 create operation, the RDE Device shall in accordance with the Redfish standard [DSP0266](#) respond with
1889 an annotation specifying the sequence number as an unsupported property (see PropertyUnknown in the
1890 Redfish base message registry). The RDE Device may continue to decode the remainder of the payload
1891 and perform the requested Operation upon the portion it understands.

1892 If the MC finds an unrecognized sequence number as part of the response payload for a get or action
1893 Operation, or as part of a @Message.ExtendedInfo annotation response for any other Operation, it shall
1894 treat this as a failure on the part of the RDE Device and respond to the client with HTTP status code 500,
1895 Internal Server Error.

1896 8.5.4 Sequence numbers for read-only properties in modification Operations

1897 If the RDE Device is performing a modification operation (create, put, patch, or some actions), and it finds
1898 a sequence number corresponding to a property that is read-only, the RDE Device should in accordance
1899 with the Redfish standard [DSP0266](#) respond with an annotation specifying the sequence number as a
1900 non-updateable property (see PropertyNotWritable in the Redfish base message registry). The RDE
1901 Device may continue to decode and update with the remainder of the payload.

1902 8.6 Example encoding and decoding

1903 The following examples demonstrate the BEJ encoding and decoding processes. For illustrative
 1904 purposes, we show the data collected in an XML form that happens to align with the schema; however,
 1905 there is no requirement that data be stored in this form. Indeed, it is very unlikely that any RDE Device
 1906 would do so.

1907 The examples in this clause use the example dictionary from clause 8.6.1.

1908 8.6.1 Example dictionary

1909 The example dictionary is based on the DummySimple JSON schema presented in Figure 5:

```

1910 {
1911     "$ref": "#/definitions/DummySimple",
1912     "$schema": "http://json-schema.org/draft-04/schema#",
1913     "copyright": "Copyright 2018 DMTF. For
1914         the full DMTF copyright policy, see http://www.dmtf.org/about/policies/copyright",
1915     "definitions": {
1916         "LinkStatus": {
1917             "enum": [
1918                 "NoLink",
1919                 "LinkDown",
1920                 "LinkUp"
1921             ],
1922             "type": "string"
1923         },
1924         "DummySimple" : {
1925             "additionalProperties": false,
1926             "description": "The DummySimple schema represents a very simple schema used to
1927                 demonstrate the BEJ dictionary format.",
1928             "longDescription": "This resource shall not be used except for illustrative
1929                 purposes. It does not correspond to any real hardware or software.",
1930             "patternProperties": {
1931                 "^[a-zA-Z][a-zA-Z0-9_]*?@(odata|Redfish|Message|Privileges)\\. [a-zA-Z][a-zA-
1932 Z0-9_\\.]+$": {
1933                     "description": "This property shall specify a valid odata or Redfish
1934                         property.",
1935                     "type": [
1936                         "array",
1937                         "boolean",
1938                         "number",
1939                         "null",
1940                         "object",
1941                         "string"
1942                     ]
1943                 }
1944             },
1945             "properties": {
1946                 "@odata.context": {
1947                     "$ref":
1948                         "http://redfish.dmtf.org/schemas/v1/odata.v4_0_1.json#/definitions/context"
1949                 },
1950                 "@odata.id": {
1951                     "$ref":
1952                         "http://redfish.dmtf.org/schemas/v1/odata.v4_0_1.json#/definitions/id"
1953                 },
1954                 "@odata.type": {
1955                     "$ref":
1956                         "http://redfish.dmtf.org/schemas/v1/odata.v4_0_1.json#/definitions/type"
1957                 },
1958                 "ChildArrayProperty": {
1959                     "items": {
1960                         "additionalProperties": false,
1961                         "type": "object",
1962                         "properties": {
1963                             "LinkStatus": {
1964                                 "anyOf": [

```

```

1965         {
1966             "$ref": "#/definitions/LinkStatus"
1967         },
1968         {
1969             "type": "null"
1970         }
1971     ],
1972     "readOnly": true
1973 },
1974 "AnotherBoolean": {
1975     "type": "boolean"
1976 }
1977 },
1978 },
1979 "type": "array"
1980 }
1981 },
1982 "SampleIntegerProperty": {
1983     "type": "integer"
1984 },
1985 "Id": {
1986     "type": "string",
1987     "readOnly": true
1988 },
1989 "SampleEnabledProperty": {
1990     "type": "boolean"
1991 }
1992 }
1993 },
1994 "title": "#DummySimple.v1_0_0.DummySimple"
1995 }
    
```

Figure 5 – DummySimple schema

NOTE This is not a published DMTF Redfish schema.

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

Row	Sequence Number	Format	Name	Child Pointer	Child Count
9	1	string	LinkUp	null	0
10	2	string	NoLink	null	0

2000 Finally, in binary form, the dictionary appears as shown in Figure 6. (Colors in this example match those used in
 2001 Figure 4.)

2002

2003

2004

2005

2006

2007

2008

2009

2010

2011

2012

2013

2014

2015

2016

2017

2018

2019

2020

2021

2022

2023

2024

2025

2026

2027

2028

2029

2030

2031

2032

2033

2034

2035

2036

2037

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
0x00	0x16	0x9C	0x00	0x34	0x03	0x00	0x00
0x00	0x00	0x00	0x16	0xB2	0x00	0x00	0x00
0x00	0x48	0x00	0x02	0x00	0x00	0x00	0x00
0x74	0x00	0x00	0x00	0x00	0x00	0x00	0x0F
0xC8	0x00	0x46	0x01	0x00	0x5C	0x00	0x03
0x00	0x0B	0xD7	0x00	0x50	0x00	0x00	0x00
0x00	0x00	0x00	0x09	0xE2	0x00	0x50	0x01
0x00	0x00	0x00	0x00	0x00	0x07	0xEB	0x00
0x50	0x02	0x00	0x00	0x00	0x00	0x00	0x07
0xF2	0x00	0x44	0x75	0x6D	0x6D	0x79	0x53
0x69	0x6D	0x70	0x6C	0x65	0x00	0x43	0x68
0x69	0x6C	0x64	0x41	0x72	0x72	0x61	0x79
0x50	0x72	0x6F	0x70	0x65	0x72	0x74	0x79
0x00	0x49	0x64	0x00	0x53	0x61	0x6D	0x70
0x6C	0x65	0x45	0x6E	0x61	0x62	0x6C	0x65
0x64	0x50	0x72	0x6F	0x70	0x65	0x72	0x74
0x79	0x00	0x53	0x61	0x6D	0x70	0x6C	0x65
0x49	0x6E	0x74	0x65	0x67	0x65	0x72	0x50
0x72	0x6F	0x70	0x65	0x72	0x74	0x79	0x00
0x41	0x6E	0x6F	0x74	0x68	0x65	0x72	0x42
0x6F	0x6F	0x6C	0x65	0x61	0x6E	0x00	0x4C
0x69	0x6E	0x6B	0x53	0x74	0x61	0x74	0x75
0x73	0x00	0x4C	0x69	0x6E	0x6B	0x44	0x6F
0x77	0x6E	0x00	0x4C	0x69	0x6E	0x6B	0x55
0x70	0x00	0x4E	0x6F	0x4C	0x69	0x6E	0x6B
0x00	0x18	0x43	0x6F	0x70	0x79	0x72	0x69
0x67	0x68	0x74	0x20	0x28	0x63	0x29	0x20
0x32	0x30	0x31	0x38	0x20	0x44	0x4D	0x54
0x46	0x00						

2038 **Figure 6 – DummySimple dictionary – binary form**

2039 **8.6.2 Example encoding**

2040 For this example, we start with the following data (shown here in an XML representation).

2041 NOTE The names assigned to array elements are fictitious and inserted for illustrative purposes only. Also, the
 2042 encoding sequence presented here is only one possible approach; any sequence that generates the same result is
 2043 acceptable. The value of the @odata.id annotation shown here is a deferred binding (see clause 8.3) that assumes
 2044 the DummySimple resource corresponds to a Redfish Resource PDR with resource ID 10. Finally, for illustrative
 2045 purposes we omit here the header bytes contained within the bejEncoding type that are not part of the bejTuple
 2046 PLDM type.
 2047

2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066

```
<Item name="DummySimple" type="set">
  <Item name="@odata.id" type="string" value="%L10">
  <Item name="ChildArrayProperty" type="array">
    <Item name="array element 0">
      <Item name="AnotherBoolean" type="boolean" value="true"/>
      <Item name="LinkStatus" type="enum" enumtype="String">
        <Enumeration value="NoLink"/>
      </Item>
    </Item>
    <Item name="array element 1">
      <Item name="LinkStatus" type="enum" enumtype="String">
        <Enumeration value="LinkDown"/>
      </Item>
    </Item>
  </Item>
  <Item name="Id" type="string" value="Dummy ID"/>
  <Item name="SampleIntegerProperty" type="number" value="12"/>
</Item>
```

2067 The first step of the encoding process is to insert sequence numbers, which can be retrieved from the
2068 relevant dictionary. (For purposes of this example, we are assuming that the @odata.id annotation is
2069 sequence number 16 in the annotation dictionary.) Sequence numbers for array elements correspond to
2070 their zero-based index within the array.

2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088

```
<Item name="DummySimple" type="set" seqno="major/0">
  <Item name="@odata.id" type="string" value="%L10" seqno="annotation/16">
  <Item name="ChildArrayProperty" type="array" seqno="major/0">
    <Item name="array element 0" seqno="major/0">
      <Item name="AnotherBoolean" type="boolean" value="true" seqno="major/0"/>
      <Item name="LinkStatus" type="enum" enumtype="String" seqno="major/1">
        <Enumeration value="NoLink" seqno="major/2"/>
      </Item>
    </Item>
    <Item name="array element 1" seqno="major/1">
      <Item name="LinkStatus" type="enum" enumtype="String" seqno="major/1">
        <Enumeration value="LinkDown" seqno="major/0"/>
      </Item>
    </Item>
  </Item>
  <Item name="Id" type="string" value="Dummy ID" seqno="major/1"/>
  <Item name="SampleIntegerProperty" type="integer" value="12" seqno="major/3"/>
</Item>
```

2089 After the sequence numbers are fully characterized, they can be encoded. We encode which dictionary
2090 these sequence numbers came by shifting them left one bit to insert 0b (major dictionary) or 1b
2091 (annotation dictionary) as the low order bit per clause 8.2.1. As the sequence numbers are now assigned,
2092 names of properties and enumeration values are no longer needed:
2093
2094

```

2095 <Item type="set" seqno="0">
2096   <Item seqno="33" type="string" value="%L10" seqno="annotation/16">
2097     <Item type="array" seqno="0">
2098       <Item seqno="0">
2099         <Item type="boolean" value="true" seqno="0"/>
2100         <Item type="enum" enumtype="String" seqno="2">
2101           <Enumeration seqno="4"/>
2102         </Item>
2103       </Item>
2104       <Item seqno="2">
2105         <Item type="enum" enumtype="String" seqno="2">
2106           <Enumeration seqno="0"/>
2107         </Item>
2108       </Item>
2109     </Item>
2110   <Item type="string" value="Dummy ID" seqno="2"/>
2111   <Item type="integer" value="12" seqno="6"/>
2112 </Item>

```

2113 The next step is to convert everything into BEJ SFLV Tuples. Per clause 5.3.12, the value of an
 2114 enumeration is the sequence number for the selected option.

```

2116 {0x01 0x00, set, [length placeholder], value={count=3,
2117   {0x01 0x21, string, [length placeholder], value="%L10"}
2118   {0x01 0x00, array, [length placeholder], value={count=2,
2119     {0x01 0x00, set, [length placeholder], value={count=2,
2120       {0x01 0x00, boolean, [length placeholder], value=true}
2121       {0x01 0x02, enum, [length placeholder], value=2}
2122     }}
2123     {0x01 0x02, set, [length placeholder], value={count=1,
2124       {0x01 0x02, enum, [length placeholder], value=0}
2125     }}
2126   }}
2127 {0x01 0x02, string, [length placeholder], value="Dummy ID"}
2128 {0x01 0x06, integer, [length placeholder], value=12}
2129 }}

```

2130 We now encode the formats and the leaf nodes, following Table 9. For sets and arrays, the value
 2131 encoding count prefix is a nonnegative Integer; we can encode that now as well per Table 4. Note the null
 2132 terminator for the string. The encoded sequence numbers for enumeration values do not need a
 2133 dictionary selector inserted as the LSB as the dictionary was already indicated with the sequence number
 2134 for the enumeration itself in the format tuple field. The @odata.id annotation string value contains a
 2135 deferred binding, so we set that bit in the format tuple field.

```

2137 {0x01 0x00, 0x00, [length placeholder], {0x01 0x04,
2138   {0x01 0x21, 0x51, [length placeholder], 0x25 0x4C 0x31 0x30}
2139   {0x01 0x00, 0x10, [length placeholder], {0x01 0x02,
2140     {0x01 0x00, 0x00, [length placeholder], {0x01 0x02,
2141       {0x01 0x00, 0x70, [length placeholder], 0xFF}
2142       {0x01 0x02, 0x40, [length placeholder], 0x01 0x02}
2143     }}
2144     {0x01 0x02, 0x00, [length placeholder], {0x01 0x01,
2145       {0x01 0x02, 0x40, [length placeholder], 0x01 0x00}
2146     }}
2147   }}
2148 {0x01 0x02, 0x50, [length placeholder],
2149   0x44 0x75 0x6D 0x6D 0x79 0x20 0x49 0x44 0x00}
2150 {0x01 0x06, 0x30, [length placeholder], 0x0C}
2151 }}

```

2152 All that remains is to fill in the length values. We begin at the leaves:

```
2153
2154 {0x01 0x00, 0x00, [length placeholder], {0x01 0x04,
2155 {0x01 0x21, 0x51, 0x01 0x04, 0x25 0x4C 0x31 0x30}
2156 {0x01 0x00, 0x10, [length placeholder], {0x01 0x02,
2157 {0x01 0x00, 0x00, [length placeholder], {0x01 0x02,
2158 {0x01 0x00, 0x70, 0x01 0x01, 0xFF}
2159 {0x01 0x02, 0x40, 0x01 0x02, 0x01 0x02}
2160 }}
2161 {0x01 0x02, 0x00, [length placeholder], {0x01 0x01,
2162 {0x01 0x02, 0x40, 0x01 0x02, 0x01 0x00}
2163 }}
2164 }}
2165 {0x01 0x02, 0x50, 0x01 0x09,
2166 0x44 0x75 0x6D 0x6D 0x79 0x20 0x49 0x44 0x00}
2167 {0x01 0x06, 0x30, 0x01 0x01, 0x0C}
2168 }}
```

2169 We then work our way from the leaves towards the outermost enclosing tuples. First, the array element
2170 sets:

```
2171
2172 {0x01 0x00, 0x00, [length placeholder], {0x01 0x04,
2173 {0x01 0x21, 0x51, 0x01 0x04, 0x25 0x4C 0x31 0x30}
2174 {0x01 0x00, 0x10, [length placeholder], {0x01 0x02,
2175 {0x00, 0x00, 0x01 0x0F, {0x01 0x02,
2176 {0x01 0x00, 0x07, 0x01 0x01, 0xFF}
2177 {0x01 0x20, 0x04, 0x01 0x02, 0x01 0x02}
2178 }}
2179 {0x01 0x02, 0x00, 0x01 0x09, {0x01 0x01,
2180 {0x01 0x02, 0x40, 0x01 0x02, 0x01 0x00}
2181 }}
2182 }}
2183 {0x01 0x02, 0x50, 0x01 0x09,
2184 0x44 0x75 0x6D 0x6D 0x79 0x20 0x49 0x44 0x00}
2185 {0x01 0x06, 0x30, 0x01 0x01, 0x0C}
2186 }}
```

2187 Next, the array itself:

```
2188
2189 {0x01 0x00, 0x00, [length placeholder], {0x01 0x04,
2190 {0x01 0x21, 0x51, 0x01 0x04, 0x25 0x4C 0x31 0x30}
2191 {0x01 0x00, 0x10, 0x01 0x24, {0x01 0x02,
2192 {0x01 0x00, 0x00, 0x01 0x0F, {0x01 0x02,
2193 {0x01 0x00, 0x70, 0x01 0x01, 0xFF}
2194 {0x01 0x02, 0x40, 0x01 0x02, 0x01 0x02}
2195 }}
2196 {0x01 0x02, 0x00, 0x01 0x09, {0x01 0x01,
2197 {0x01 0x02, 0x40, 0x01 0x02, 0x01 0x00}
2198 }}
2199 }}
2200 {0x01 0x02, 0x50, 0x01 0x09,
2201 0x44 0x75 0x6D 0x6D 0x79 0x20 0x49 0x44 0x00}
2202 {0x01 0x06, 0x30, 0x01 0x01, 0x0C}
2203 }}
```


2204 Finally, the outermost set:

```

2205
2206 {0x01 0x00, 0x00, 0x01 0x48, {0x01 0x04,
2207 {0x01 0x21, 0x51, 0x01 0x04, 0x25 0x4C 0x31 0x30}
2208 {0x01 0x00, 0x10, 0x01 0x24, {0x01 0x02,
2209 {0x01 0x00, 0x00, 0x01 0x0F, {0x01 0x02,
2210 {0x01 0x00, 0x70, 0x01 0x01, 0xFF}
2211 {0x01 0x02, 0x40, 0x01 0x02, 0x01 0x02}
2212 }}
2213 {0x01 0x02, 0x00, 0x01 0x09, {0x01 0x01,
2214 {0x01 0x02, 0x40, 0x01 0x02, 0x01 0x00}
2215 }}
2216 }}
2217 {0x01 0x02, 0x50, 0x01 0x09,
2218 0x44 0x75 0x6D 0x6D 0x79 0x20 0x49 0x44 0x00}
2219 {0x01 0x06, 0x30, 0x01 0x01, 0x0C}
2220 }}
    
```

2221 The encoded bytes may now be read off, and the inner encoding is complete:

```

2222
2223 0x01 0x00 0x00 0x01 : 0x48 0x01 0x04 0x01
2224 0x21 0x51 0x01 0x04 : 0x25 0x4C 0x31 0x30
2225 0x01 0x00 0x10 0x01 : 0x24 0x01 0x02 0x01
2226 0x00 0x00 0x01 0x0F : 0x01 0x02 0x01 0x00
2227 0x70 0x01 0x01 0xFF : 0x01 0x02 0x40 0x01
2228 0x02 0x01 0x02 0x01 : 0x02 0x00 0x01 0x09
2229 0x01 0x01 0x01 0x02 : 0x40 0x01 0x02 0x01
2230 0x00 0x01 0x02 0x50 : 0x01 0x09 0x44 0x75
2231 0x6D 0x6D 0x79 0x20 : 0x49 0x44 0x00 0x01
2232 0x06 0x30 0x01 0x01 : 0x0C
    
```

2233 **8.6.3 Example decoding**

2234 The decoding process is largely the inverse of the encoding process. For this example, we start with the
 2235 final encoded data from clause 8.6.1:

```

2236
2237 0x01 0x00 0x00 0x01 : 0x48 0x01 0x04 0x01
2238 0x21 0x51 0x01 0x04 : 0x25 0x4C 0x31 0x30
2239 0x01 0x00 0x10 0x01 : 0x24 0x01 0x02 0x01
2240 0x00 0x00 0x01 0x0F : 0x01 0x02 0x01 0x00
2241 0x70 0x01 0x01 0xFF : 0x01 0x02 0x40 0x01
2242 0x02 0x01 0x02 0x01 : 0x02 0x00 0x01 0x09
2243 0x01 0x01 0x01 0x02 : 0x40 0x01 0x02 0x01
2244 0x00 0x01 0x02 0x50 : 0x01 0x09 0x44 0x75
2245 0x6D 0x6D 0x79 0x20 : 0x49 0x44 0x00 0x01
2246 0x06 0x30 0x01 0x01 : 0x0C
    
```

2247 The first step of the decoding process is to map the byte data to {SFLV} tuples, using the length bytes and
 2248 set/array counts to identify tuple boundaries:

```

2249
2250
    
```

2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266

```
{S=0x01 0x00, F=0x00, L=0x01 0x3F, V={0x01 0x04,
  {S=0x01 0x21, F=0x51, L=0x01 0x04, V=0x25 0x4C 0x31 0x30}
  {S=0x01 0x00, F=0x10, L=0x01 0x24, V={0x01 0x02,
    {S=0x01 0x00, F=0x00, L=0x01 0x0F, V={0x01 0x02,
      {S=0x01 0x00, F=0x70, L=0x01 0x01, V=0xFF}
      {S=0x01 0x02, F=0x40, L=0x01 0x02, V=0x01 0x02}
    }}
    {S=0x01 0x02, F=0x00, L=0x01 0x09, V={0x01 0x01,
      {S=0x01 0x02, F=0x40, L=0x01 0x02, V=0x01 0x00}
    }}
  }}
  {S=0x01 0x02, F=0x50, L=0x01 0x09,
    V=0x44 0x75 0x6D 0x6D 0x79 0x20 0x49 0x44 0x00}
  {0x01 S=0x06, F=0x30, L=0x01 0x01, V=0x0C}
}}
```

2267
2268

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

2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282

```
{S=0x01 0x00, F=0x00, V={
  {S=0x01 0x21, F=0x51, V=0x25 0x4C 0x31 0x30}
  {S=0x01 0x00, F=0x10, V={
    {S=0x01 0x00, F=0x00, V={
      {S=0x01 0x00, F=0x70, V=0xFF}
      {S=0x01 0x02, F=0x40, V=0x01 0x02}
    }}
    {S=0x01 0x02, F=0x00, V={
      {S=0x01 0x02, F=0x40, V=0x01 0x00}
    }}
  }}
  {S=0x01 0x02, F=0x50, V=0x44 0x75 0x6D 0x6D 0x79 0x20 0x49 0x44 0x00}
  {S=0x01 0x06, F=0x30, V=0x0C}
}}
```

2283
2284

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

2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299

```
{S=0x01 0x00, set, V={
  {S=0x01 0x21, string with deferred binding, V=0x25 0x4C 0x31 0x30}
  {S=0x01 0x00, array, V={
    {S=0x01 0x00, set, V={
      {S=0x01 0x00, boolean, V=0xFF}
      {S=0x01 0x02, enum, V=0x01 0x02}
    }}
    {S=0x01 0x02, set, V={
      {S=0x01 0x02, enum, V=0x01 0x00}
    }}
  }}
  {S=0x01 0x02, string, V=0x44 0x75 0x6D 0x6D 0x79 0x20 0x49 0x44 0x00}
  {S=0x01 0x06, integer, V=0x0C}
}}
```

2300
2301
2302
2303
2304

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

2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319

```
{S=0x01 0x00, set, {
  {S=0x01 0x21, string, "/redfish/v1/systems/1/DummySimples/1"}
  {S=0x01 0x00, array, {
    {S=0x01 0x00, set, {
      {S=0x01 0x00, boolean, true}
      {S=0x01 0x02, enum, <value 2>}
    }}
    {S=0x01 0x02, set, {
      {S=0x01 0x02, enum, <value 0>}
    }}
  }}
  {S=0x01 0x02, string, "Dummy ID"}
  {S=0x01 0x06, integer, 12}
}}
```

2320 Next, we decode the sequence numbers to identify which dictionary they select:

2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335

```
{S=major/0, set, {
  {S=annotation/16, string, "/redfish/v1/systems/1/DummySimples/1"}
  {S=major/0, array, {
    {S=major/0, set, {
      {S=major/0, boolean, true}
      {S=major/1, enum, <value 2>}
    }}
    {S=major/1, set, {
      {S=major/1, enum, <value 0>}
    }}
  }}
  {S=major/1, string, "Dummy ID"}
  {S=major/3, integer, 12}
}}
```

2336 Next, we use the selected dictionary to replace decoded sequence numbers with the strings they
2337 represent:

2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352

```
{"DummySimple", set, {
  {"@odata.id", string, "/redfish/v1/systems/1/DummySimples/1"}
  {"ChildArrayProperty", array, {
    {<Array element 0>, set, {
      {"AnotherBoolean", boolean, true}
      {"LinkStatus", enum, "NoLink"}
    }}
    {<Array element 1>, set, {
      {"LinkStatus", enum, "LinkDown"}
    }}
  }}
  {"Id", string, "Dummy ID"}
  {"SampleIntegerProperty", integer, 12}
}}
```

2353 We can now write out the decoded BEJ data in JSON format if desired (an MC will need to do this to
2354 forward an RDE Device's response to a client, but an RDE Device may not need this step):
2355
2356

2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373

```

{
  "DummySimple" : {
    "@odata.id" : "/redfish/v1/systems/1/DummySimples/1",
    "ChildArrayProperty" : [
      {
        "AnotherBoolean" : true,
        "LinkStatus" : "NoLink"
      },
      {
        "LinkStatus" : "LinkDown"
      }
    ],
    "Id" : "Dummy ID",
    "SampleIntegerProperty" : 12
  }
}

```

2374 8.7 BEJ locators

2375 A BEJ locator represents a particular location within a resource at which some operation is to take place.
2376 The locator itself consists of a list of sequence numbers for the series of nodes representing the traversal
2377 from the root of the schema tree down to the point of interest. The list of schema nodes is concatenated
2378 together to form the locator. A locator with no sequence numbers targets the root of the schema.

2379 NOTE The sequence numbers are absolute as they are relative to the schema, not to the subset of the schema for
2380 which the RDE Device supports data. This enables a locator to be unambiguous.

2381 As an example, consider a locator, encoded for the example dictionary of clause 8.6.1:

2382 0x01 0x08 0x01 0x00 0x01 0x00 0x01 0x06 0x01 0x02

2383 Decoding this locator, begins with decoding the length in bytes of the locator. In this case, the first two
2384 bytes specify that the remainder of the locator is 8 bytes long. The next step is to decode the bejTupleS-
2385 formatted sequence numbers. The low-order bit of each sequence number references the schema to
2386 which it refers; in this case, the pattern 0b indicates the major schema. Decoding produces the following
2387 list:

2388 0, 0, 3, 1

2389 Now, referring to the dictionary enables identification of the target location. Remember that all indices are
2390 zero-based:

- 2391 • The first zero points to DummySimple
- 2392 • The second zero points to the first child of DummySimple, or ChildArrayProperty
- 2393 • The three points to the fourth element in the ChildArrayProperty array, an anonymous instance
2394 of the array type (array instances are not reflected in the dictionary, but are implicitly the
2395 immediate children of any array)
- 2396 • The one points to the second child inside the ChildArray element type, or LinkStatus

2397 9 Operational behaviors

2398 This clause describes the operational behavior for initialization, Operations/Tasks, and Events.

2399 9.1 Initialization (MC perspective)

2400 The following clauses present initialization of RDE Devices with MCs.

2401 9.1.1 Sample initialization ladder diagram

2402 Figure 7 presents the ladder diagram for an example initialization sequence.

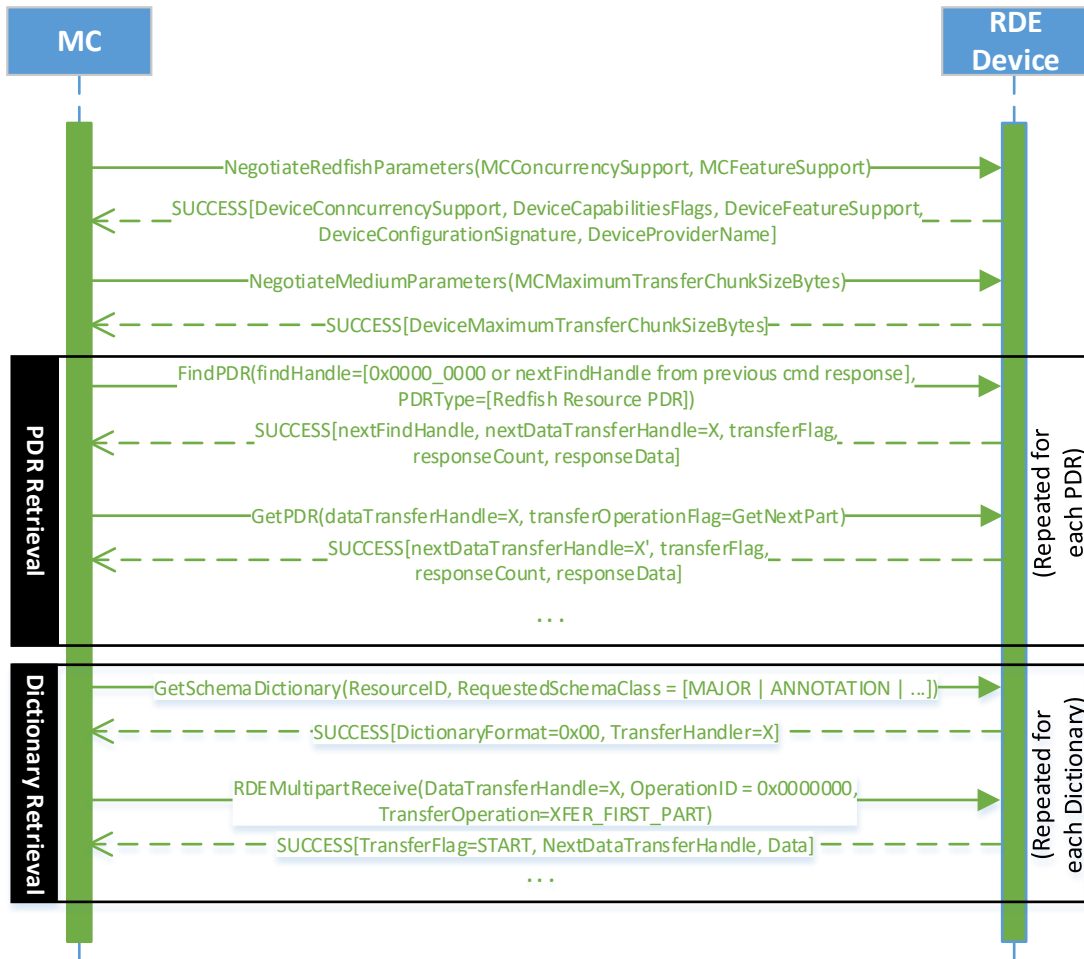
2403 Once the MC detects the RDE Device, it begins the discovery process by invoking the
2404 NegotiateRedfishParameters command to determine the concurrency and feature support for the RDE
2405 Device. It then uses the NegotiateMediumParameters command to determine the maximum message
2406 size that the MC and the RDE Device can both support. This finishes the RDE discovery process.

2407 After discovery comes the RDE registration process. It consists of two parts, PDR retrieval and dictionary
2408 retrieval. To retrieve the RDE PDRs, the MC utilizes the PLDM for Platform Monitoring and Control
2409 FindPDR command to locate PDRs that are specific to RDE⁴. For each such PDR located, the MC then
2410 retrieves it via one or more message sequences in the PLDM for Platform Monitoring and Control
2411 GetPDR command.

2412 After all the PDRs are retrieved, the next step is to retrieve dictionaries. For each Redfish Resource PDR
2413 that the MC retrieved, it retrieves the relevant dictionaries via a standardized process in which it first
2414 executes the GetSchemaDictionary command to obtain a transfer handle for the dictionary. It then uses
2415 the transfer handle with the RDE MultipartReceive command to retrieve the corresponding dictionary.

2416 Multiple initialization variants are possible; for example, it is conceivable that retrieval of some or all
2417 dictionaries could be postponed until such time as the MC needs to translate BEJ and/or JSON code for
2418 the relevant schema. Further, the MC may be able to determine that one or more of the dictionaries it has
2419 already retrieved is adequate to support a PDR and thus skip retrieving that dictionary anew. Finally, if the
2420 DeviceConfigurationSignature from the NegotiateRedfishParameters command matches the one for data
2421 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.



2422

2423

Figure 7 – Example Initialization ladder diagram

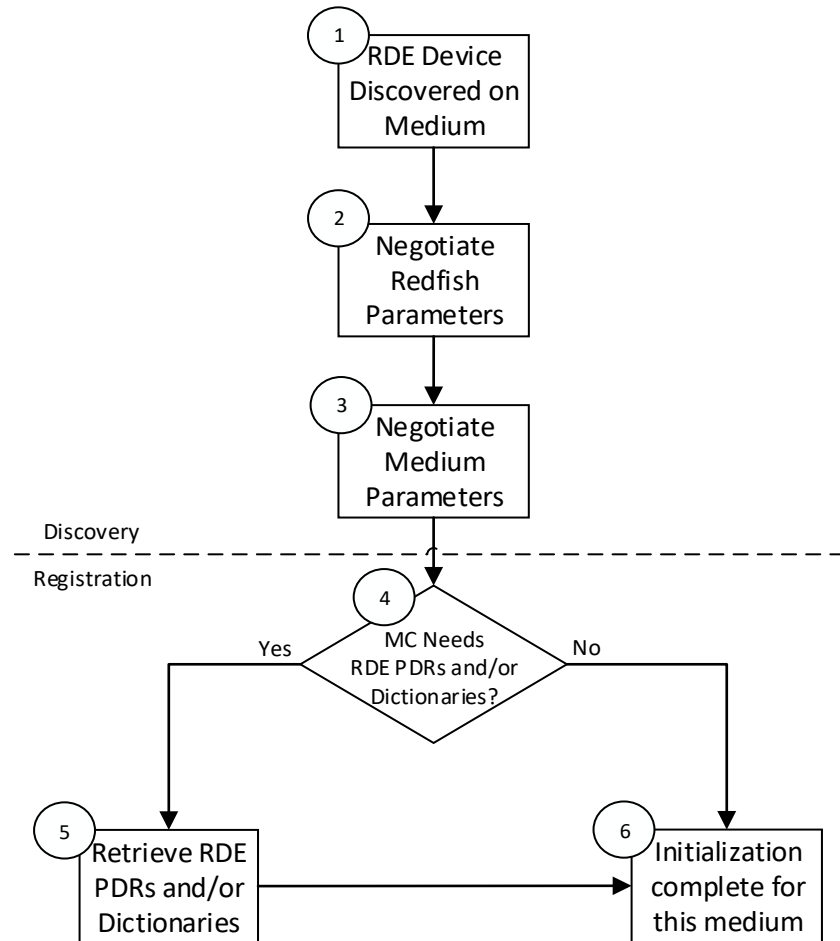
2424 **9.1.2 Initialization workflow diagram**

2425 Table 45 details the information presented visually in Figure 8.

2426

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_REDFISH	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



2427

2428

Figure 8 – Typical RDE Device discovery and registration

2429 **9.2 Operation/Task lifecycle**

2430 The following clauses present the Task lifecycle from two perspectives, first from an Operation-centric
 2431 viewpoint and then from the RDE Device perspective. MC and RDE Device implementations of RDE shall
 2432 comply with the sequences presented here.

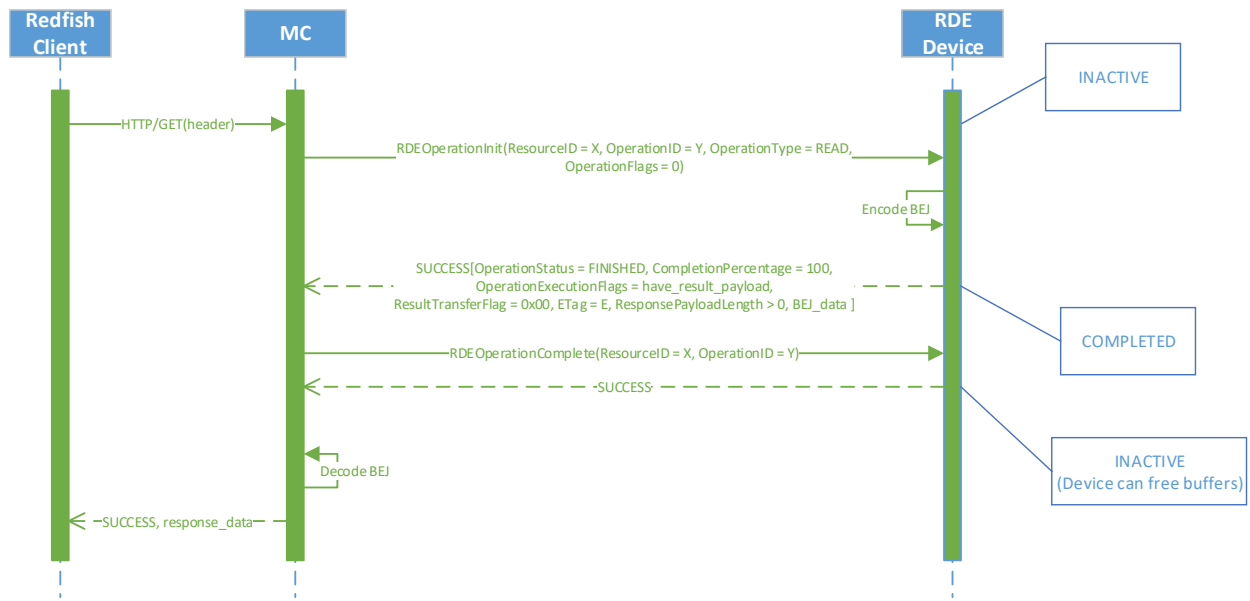
2433 **9.2.1 Example Operation command sequence diagrams**

2434 This clause presents request/response messaging sequences for common Operations.

2435 **9.2.1.1 Simple read Operation ladder diagram**

2436 Figure 9 presents the ladder diagram for a simple read Operation. The Operation begins when the
 2437 Redfish client sends a GET request over an HTTP connection to the MC. The MC decodes the URI
 2438 targeted by the GET operation to pin it down to a specific resource and PDR and sends the
 2439 RDEOperationInit command to the RDE Device that owns the PDR, with OperationType set to READ.
 2440 The RDE Device now has everything it needs for the Operation, so it performs a BEJ encoding of the
 2441 schema data for the requested resource and sends it as an inline payload back to the MC. Sending inline
 2442 is possible in this case because the read data is small enough to not cause the response message to
 2443 exceed the maximum transfer size that was previously negotiated in the NegotiateMediumParameters
 2444 command. The MC in turn has all of the results for the Operation, so it sends RDEOperationComplete to
 2445 finalize the Operation. The RDE Device can now throw away the BEJ encoded read result, so it does so

2446 and responds to the MC with success. Finally, the MC uses the dictionary it previously retrieved from the
 2447 RDE Device to decode the BEJ payload for the read command into JSON data and the MC sends the
 2448 JSON data back to the client.



2449

2450

Figure 9 – Simple read Operation ladder diagram

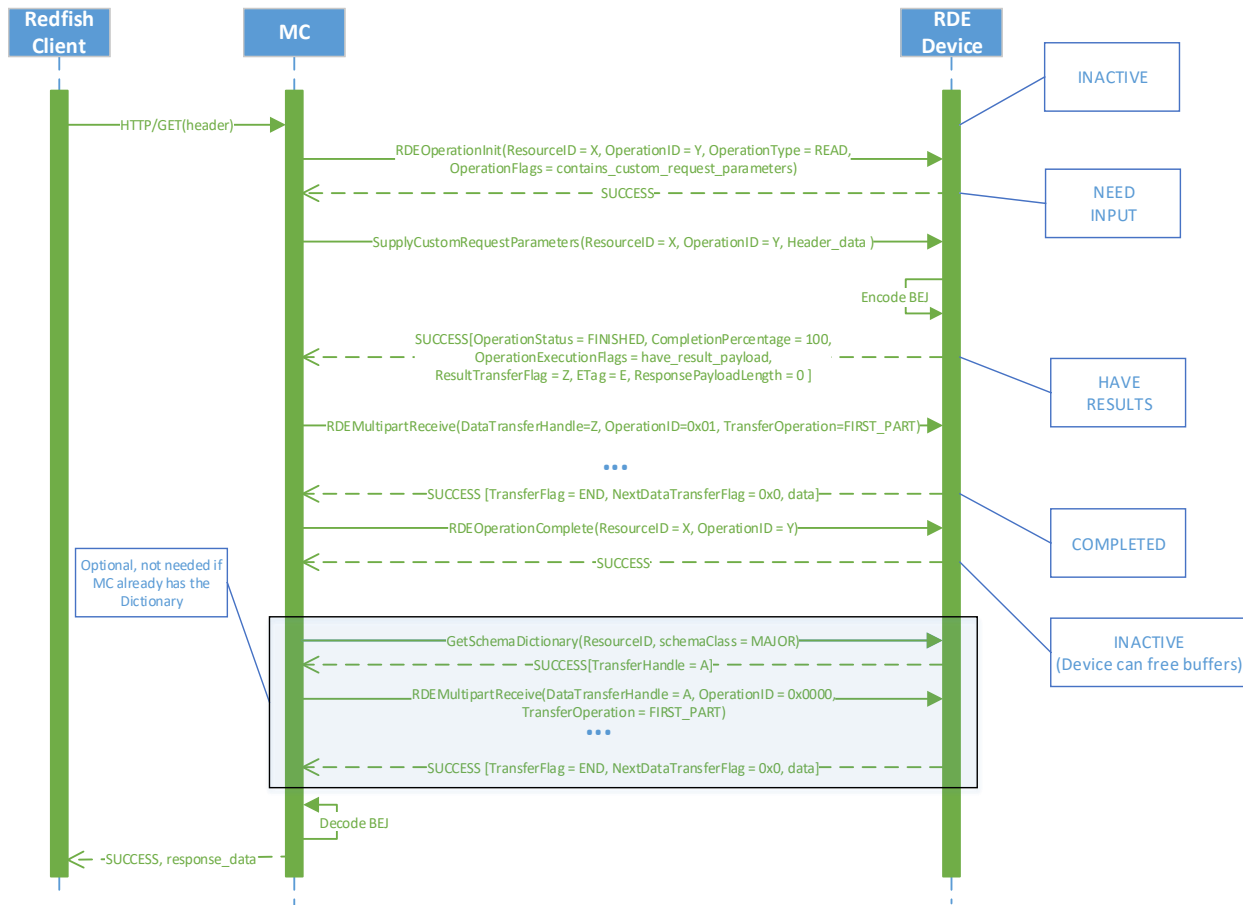
2451

9.2.1.2 Complex read Operation diagram

2452 Figure 10 presents the ladder diagram for a more complex read Operation. As with the simple read case,
 2453 the Operation begins when the Redfish client sends a GET request over an HTTP connection to the MC.
 2454 The MC again decodes the URI targeted by the GET operation to pin it down to a specific resource and
 2455 PDR and sends the RDEOperationInit command to the RDE Device that owns the PDR, with
 2456 OperationType set to READ. In this case, however, the OperationFlags that the MC sent with the
 2457 RDEOperationInit command indicate that there are supplemental parameters to be sent to the RDE
 2458 Device, so the RDE Device must wait for these before beginning work on the Operation. The MC sends
 2459 these supplemental parameters to the RDE Device via the SupplyCustomRequestParameters command.

2460 At this point, the RDE Device has everything it needs for the Operation, so just as before, the RDE
 2461 Device performs a BEJ encoding of the schema data for the requested resource. As opposed to the
 2462 previous example, in this case the BEJ-encoded payload is too large to fit within the response message,
 2463 so the RDE Device instead supplied a transfer handle that the MC can use to retrieve the BEJ payload
 2464 separately. The MC, seeing this, performs a series of RDEMultipartReceive commands to retrieve the
 2465 payload. Once it is all transferred, the MC has everything it needs. Whether it needed to retrieve a
 2466 dictionary or it already had one, the MC now sends the RDEOperationComplete command to finalize the
 2467 Operation and allow the RDE Device to throw away the BEJ encoded read result. If the MC needs a
 2468 dictionary to decode the BEJ payload, it may retrieve one via the GetSchemaDictionary command
 2469 followed by one or more RDEMultipartReceive commands to retrieve the binary dictionary data.
 2470 (Normally, the MC would have retrieved the dictionary during initialization; however, if the MC has limited
 2471 storage space to cache dictionaries, it may have been forced to evict it.) Finally, the MC uses the
 2472 dictionary to decode the BEJ payload for the read command into JSON data and then the MC sends the
 2473 JSON data back to the client.
 2474

2475



2476

2477

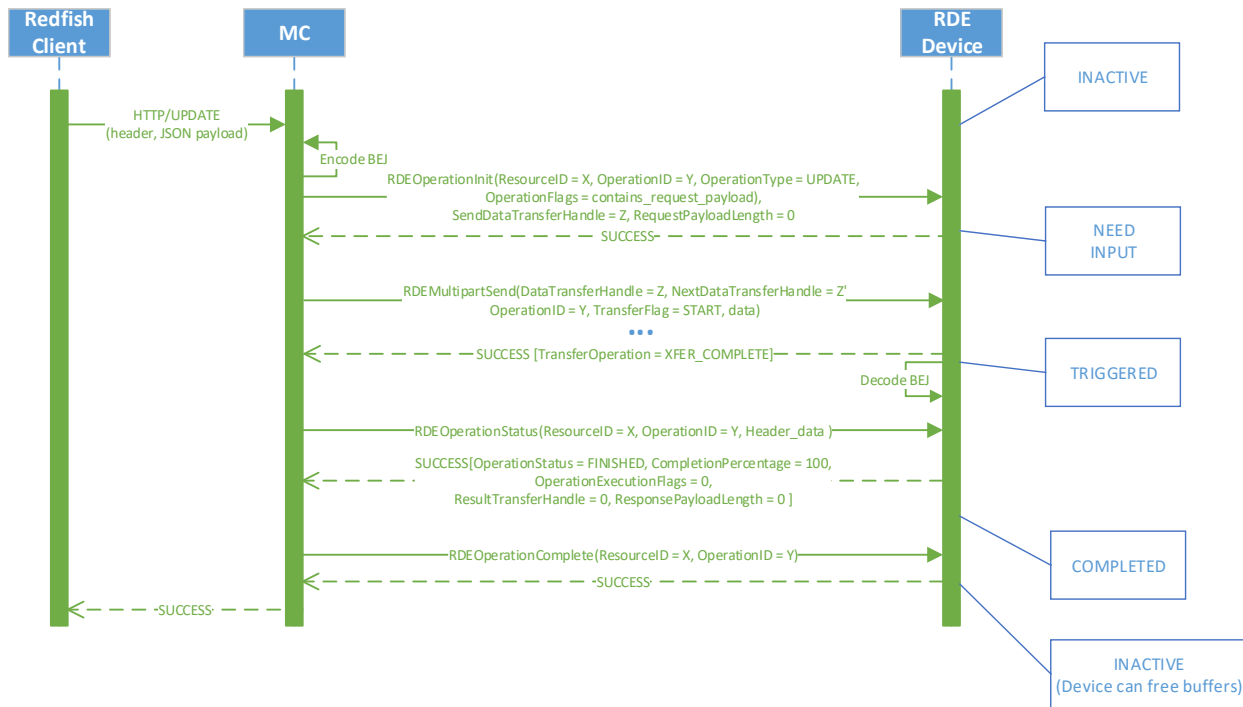
Figure 10 – Complex Read Operation ladder diagram

2478 **9.2.1.3 Write (update) Operation ladder diagram**

2479 Figure 11 presents the ladder diagram for a write Operation. As with the read cases, the Operation begins
 2480 when the Redfish client sends a request over an HTTP connection to the MC, in this case, an UPDATE.
 2481 Once again, the MC decodes the URI targeted by the UPDATE Operation to pin it down to a specific
 2482 resource and PDR. Before it can send the RDEOperationInit command to the RDE Device that owns the
 2483 PDR, the MC must perform a BEJ encoding of the JSON payload it received from the Redfish client. If the
 2484 BEJ encoded payload were small enough to fit within the maximum transfer chunk, the MC could inline it
 2485 with the RDEOperationInit command; however, in this example, that is not the case. The MC therefore
 2486 sends RDEOperationInit with the OperationType set to UPDATE and a nonzero transfer handle. Seeing
 2487 this, the RDE Device knows to expect a larger payload via RDEMulticastSend.

2488 The MC uses the RDEMulticastSend command to transfer the encoded payload to the RDE Device in one
 2489 or more chunks. The contains_request_parameters Operation flag is not set, so the RDE Device will not
 2490 expect supplemental parameters as part of this Operation. Having everything it needs to execute, the
 2491 RDE Device moves to the TRIGGERED state. The MC now sends the RDEOperationStatus command to
 2492 the RDE Device to have it execute the Operation. (In practice, the RDE Device is allowed to begin
 2493 executing the Operation as soon as it has received the request payload, so it may choose not to wait for
 2494 the RDEOperationStatus command to do so.) The RDE Device executes the Operation and sends the

2495 results to the MC as the response to the RDEOperationStatus command. As before, the MC finalizes the
 2496 Operation via RDEOperationComplete and then sends the results back to the client.



2497

2498

Figure 11 – Write Operation ladder diagram

2499 **9.2.1.4 Write (update) with Long-running Task Operation Ladder Diagram**

2500

2501 Figure 12 presents the ladder diagram for a write Operation that spawns a long-running Task. As with the
 2502 previous case, the Operation begins when the Redfish client sends an UPDATE request over an HTTP
 2503 connection to the MC, and the MC decodes the URI targeted by the UPDATE Operation to pin it down to
 2504 a specific resource and PDR. Before it can send the RDEOperationInit command to the RDE Device that
 2505 owns the PDR, the MC must perform a BEJ encoding of the JSON payload it received from the Redfish
 2506 client. Unlike the previous example, the BEJ encoded payload here is small enough to fit in the maximum
 2507 transfer chunk, so the MC inlines it into the RDEOperationInit request command. Again, the
 2508 contains_request_parameters Operation flag is not set, so the RDE Device will not expect supplemental
 2509 parameters as part of this Operation.

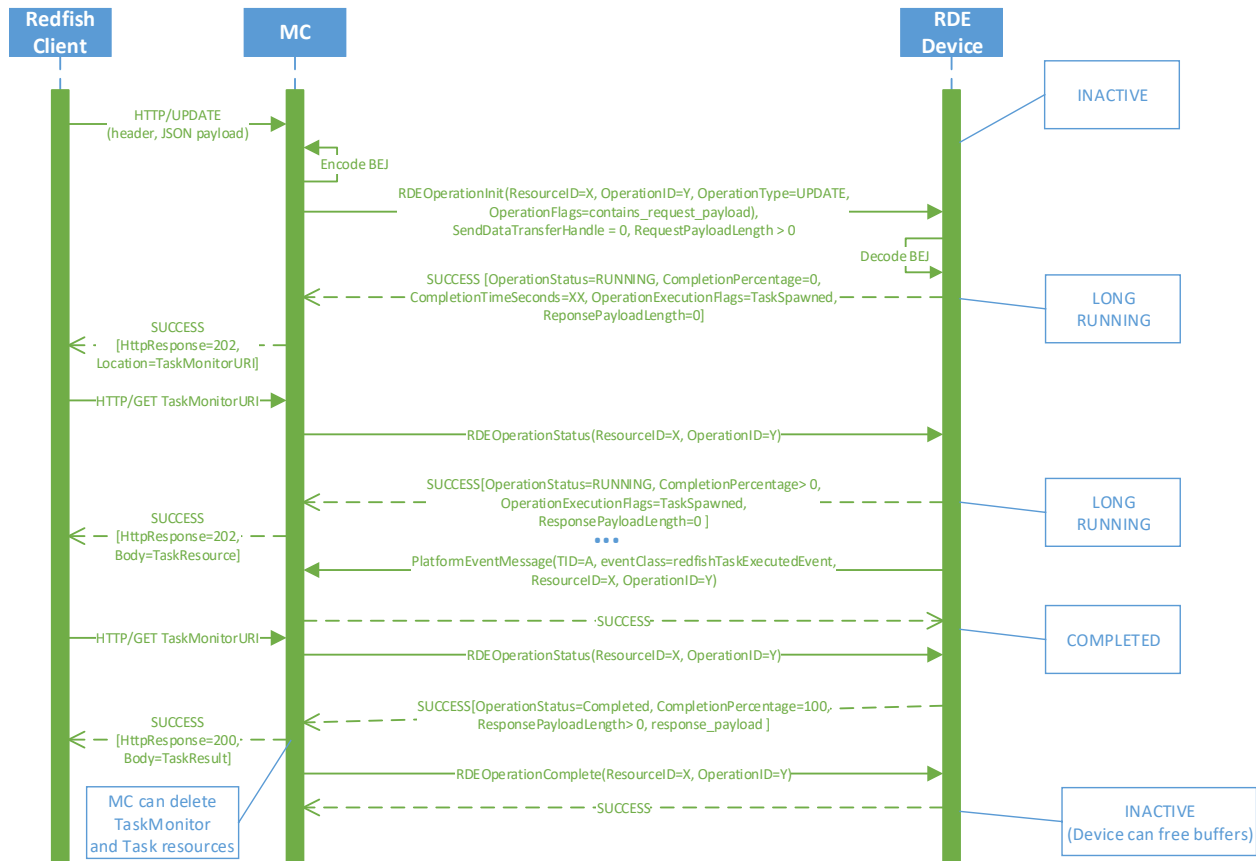
2510 When the RDE Device receives the RDEOperationInit request command, it has everything it needs to
 2511 begin work on the Operation. In this case, the RDE Device determines that performing the write will take
 2512 longer than PT1, so the RDE Device spawns a long-running Task to process the write asynchronously
 2513 and sends TaskSpawned in the OperationExecutionFlags to inform the MC.

2514 When it discovers that the RDE Device spawned a long-running Task, the MC adds a member to the
 2515 Task collection it maintains and synthesizes a TaskMonitor URI to send back to the client in a location
 2516 response header. At this point, the client can issue an HTTP GET to retrieve a status update on the Task;
 2517 when it does so, the MC sends RDEOperationStatus to the RDE Device to get the status update and
 2518 sends it back to the client as the result of the GET operation.

2519 At some point, the asynchronous Task finishes executing. When this happens, the RDE Device issues a
 2520 PlatformEventMessage to send a TaskCompletion event to the MC. (This presupposes that the RDE

2521 Device and the MC both support asynchronous eventing. Were this not the case, the RDE Device would
 2522 still generate the TaskCompletion event, but would wait for the MC to invoke the
 2523 PollForPlatformEventMessage command to report the event.) Regardless of which way the MC gets the
 2524 event, it then sends the RDEOperationStatus command one last time in order to retrieve the final results
 2525 from the Operation. The next time the client performs a GET on the TaskMonitor, the MC can send back
 2526 the final results of the Operation. Finally, the MC finalizes the Operation via RDEOperationComplete at
 2527 which point the MC can delete the Task collection member and the TaskMonitor URI and the RDE Device
 2528 can free up any buffers associated with the Operation and/or Task.

2529



2530

2531

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

2532

9.2.2 Operation/Task overview workflow diagrams (Operation perspective)

2533 This clause describes the operating behavior for MCs and RDE Devices over the lifecycle of Operations
 2534 from an Operation-centric perspective. The workflow diagrams are split between simpler, short-lived
 2535 Operations and those that spawn a Task to be processed asynchronously. These workflow diagrams are
 2536 intended to capture the standard flow for the execution of most Operations, but do not cover every
 2537 possible error condition. For full precision, refer to clause 9.2.3.

2538

9.2.2.1 Operation overview workflow diagram

2539

Table 46 details the information presented visually in Figure 13.

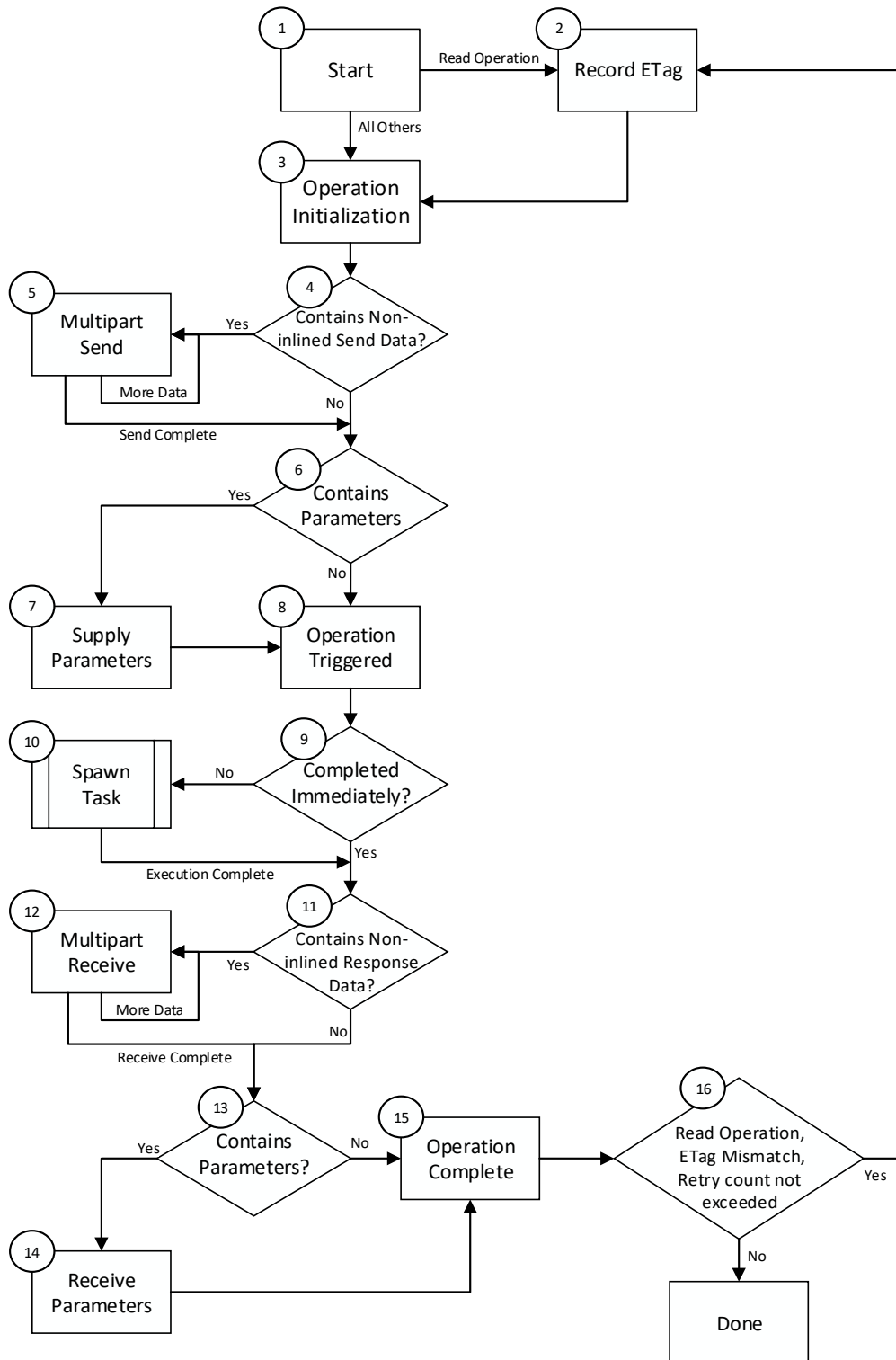
2540

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 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)	Unconditional	3
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_CHK	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_CHK	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	If the RDE Device is able to complete the Operation “quickly”	11
		Otherwise	10

Step	Description	Condition	Next Step
	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.		
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 sublife cycle.	Once the Task finishes executing	11
11 – RCV_PAYLOAD_CHUNK	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: <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_CHUNK	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	Unconditional	15

Step	Description	Condition	Next Step
	<p>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.</p>		
15 – COMPLETE	<p>The MC sends the RDEOperationComplete command to finalize the Operation</p>	n/a	n/a
16 – CMP_DIGEST	<p>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).</p>	<p>Read operation and mismatched ETags and retry count not exceeded</p>	2
		<p>Not a read, no ETag collected, the ETags match, or retry count exceeded</p>	n/a: Done



2541

2542

Figure 13 – RDE Operation lifecycle overview (holistic perspective)

2543

9.2.2.2 Task overview workflow diagram

2544

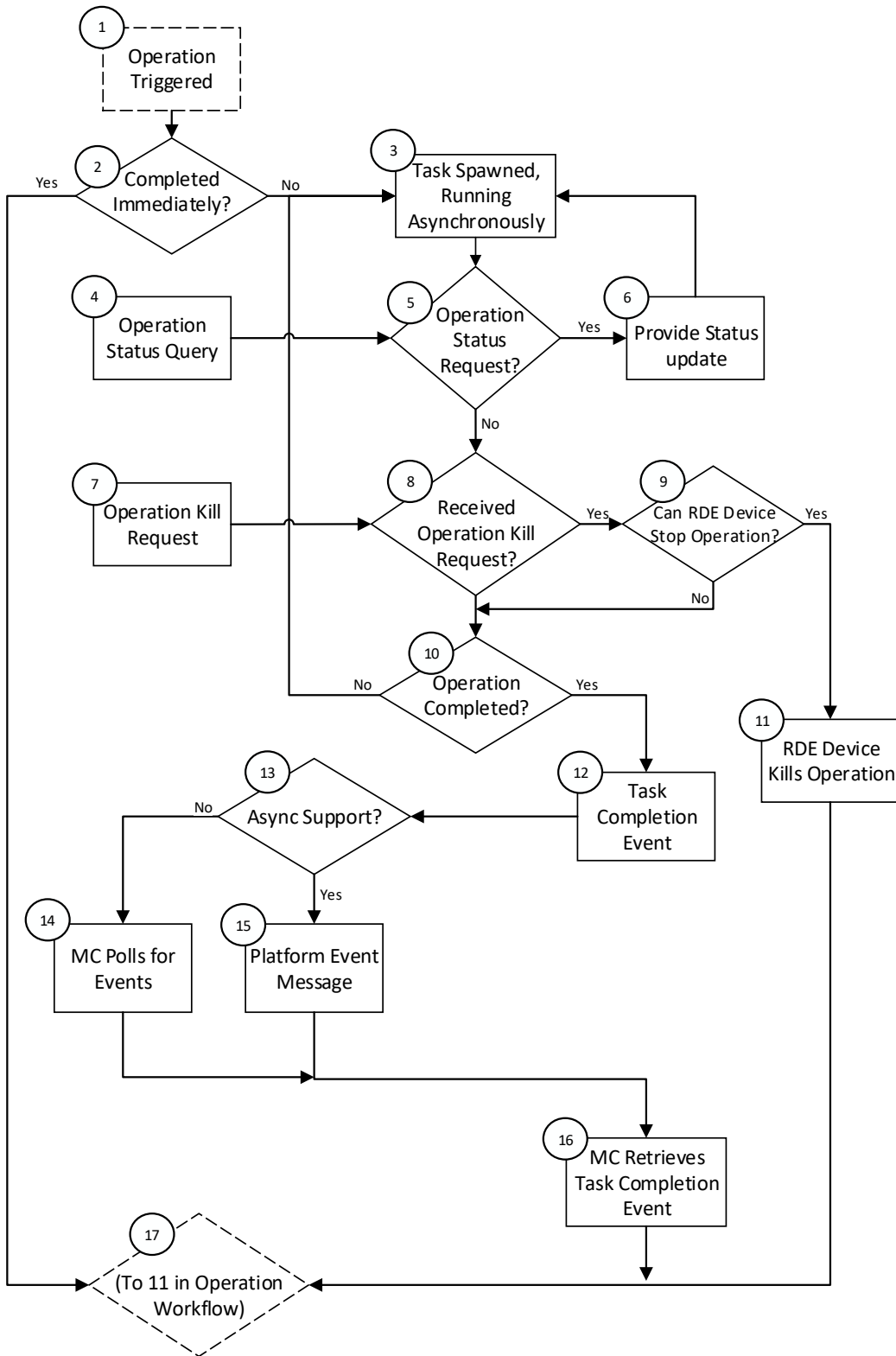
Table 47 details the information presented visually in Figure 14.

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_STATU S	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

2546



2547

2548

2549

Figure 14 – RDE Task lifecycle overview (holistic perspective)

2550 9.2.3 RDE Operation state machine (RDE Device perspective)

2551 The following clauses describe the operating behavior for the lifecycle of Operations and Tasks from an
2552 RDE Device-centric perspective. Table 48 details the information presented visually in Figure 15. The
2553 states presented in this state machine are not (collectively) the total state for the RDE Device, but rather
2554 the state for the Operation. The total state for the RDE Device would involve separate instances of the
2555 Task/Operation state machine replicated once for each of the concurrent Operations that the RDE Device
2556 and the MC negotiated to support at registration time.

2557 9.2.3.1 State definitions

2558 The following states shall be implemented by the RDE Device for each Operation it is supporting:

- 2559 • INACTIVE
 - 2560 – INACTIVE is the default Operation state in which the RDE Device shall start after
 - 2561 initialization. In this state, the RDE Device is not processing an Operation as it has not
 - 2562 received an RDEOperationInit command from the MC.
- 2563 • NEED_INPUT
 - 2564 – After receiving the RDEOperationInit command, the RDE Device moves to this state if it is
 - 2565 expecting additional Operation-specific parameters or a payload that was not inlined in the
 - 2566 RDEOperationInit command.
- 2567 • TRIGGERED
 - 2568 – Once the RDE Device receives everything it needs to execute an Operation, it begins
 - 2569 executing it immediately. If the triggering command – the command that supplied the last
 - 2570 bit of data needed to execute the Operation – was RDEOperationInit or
 - 2571 SupplyCustomRequestParameters, the response message to the triggering command
 - 2572 reflects the initial results for the Operation. However, if the triggering command was a
 - 2573 RDEMultipartSend, initial results are deferred until the MC invokes the
 - 2574 RDEOperationStatus command. This state captures the case where the Operation was
 - 2575 triggered by a RDEMultipartSend and the MC has not yet sent an RDEOperationStatus
 - 2576 command to get initial results. In this state, the RDE Device may execute the Operation;
 - 2577 alternatively, it may wait to receive RDEOperationStatus to begin execution.
- 2578 • TASK_RUNNING
 - 2579 – If the RDE Device cannot complete the Operation within the timeframe needed for the
 - 2580 response to the command that triggered it, the RDE Device spawns a Task in which to
 - 2581 execute the Operation asynchronously.
- 2582 • HAVE_RESULTS
 - 2583 – When execution of the Operation produces a response parameters or a response payload
 - 2584 that does not fit in the response message for the command that triggered the Operation (or
 - 2585 detected its completion, if a Task was spawned or if there was a payload but no custom
 - 2586 request parameters), the RDE Device remains in this state until the MC has collected all of
 - 2587 these results.
- 2588 • COMPLETED
 - 2589 – The RDE Device has completed processing of the Operation and awaits acknowledgment
 - 2590 from the MC that it has received all Operation response data. This acknowledgment is
 - 2591 done by the MC issuing the RDEOperationComplete command. When the RDE Device
 - 2592 receives this command, it may discard any internal records or state it has maintained for
 - 2593 the Operation.

- 2594 • FAILED
 - 2595 – The MC has explicitly killed the Operation or an error prevented execution of the
 - 2596 Operation.
- 2597 • ABANDONED
 - 2598 – If MC fails to progress the Operation through this state machine, the RDE Device may
 - 2599 abort the Operation and mark it as abandoned.

2600 **9.2.3.2 Operation lifecycle state machine**

2601 Figure 15 illustrates the state transitions the RDE Device shall implement. Each bubble represents a
 2602 particular state as defined in the previous clause. Upon initialization, system reboot, or an RDE Device
 2603 reset the RDE Device shall enter the INACTIVE state.

2604 **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, HaveCustomResponseParameters bit in OperationExecutionFlags not set	INACTIVE
	RDEOperationInit - RDE Device not ready - RDE Device does wish to specify a deferral timeframe	ERROR_NOT_READY, HaveCustomResponseParameters 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

Current State	Trigger	Response	Next State
	RDEOperationStatus	OPERATION_INACTIVE	INACTIVE
	RDEOperationInit; <ul style="list-style-type: none"> - valid request - Operation Flags indicate request non-inlined payload or parameters to be sent from MC to RDE Device 	Success	NEED_INPUT
	RDEOperationInit; <ul style="list-style-type: none"> - 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; <ul style="list-style-type: none"> - 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; <ul style="list-style-type: none"> - valid request - Operation Flags indicate no request payload to be sent from MC to RDE Device (or request payload 	Success	COMPLETED

Current State	Trigger	Response	Next State
	inlined in RDEOperationInit request message) <ul style="list-style-type: none"> - request flags indicate no supplemental parameters needed - 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 		
	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; <ul style="list-style-type: none"> - neither run_to_completion nor discard_record flag set 	Success	FAILED
	RDEOperationKill; <ul style="list-style-type: none"> - run_to_completion flag not set - discard_record flag set 	Success	INACTIVE
	RDEOperationKill; <ul style="list-style-type: none"> - run_to_completion flag set - discard_record flag not set 	ERROR_INVALID_DATA	NEED_INPUT
	RDEOperationKill; <ul style="list-style-type: none"> - 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

Current State	Trigger	Response	Next State
	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 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	ERROR_ETAG_MATCH	FAILED

Current State	Trigger	Response	Next State
	ETAG_IF_NONE_MATCH and an ETAG matches		
	SupplyCustomRequestParameters; - request contains unsupported RDE custom header	ERROR_UNRECOGNIZED_CUSTOM_HEADER	FAILED
	SupplyCustomRequestParameters; - no payload data remaining to be sent - Error occurs in processing of Operation	Error specific to type of failure encountered	FAILED
	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
2 - TRIGGERED	RDEOperationInit, same rdeOpID	ERROR_OPERATION_EXISTS; no disruption to existing Operation	TRIGGERED
	RDEOperationInit, different rdeOpID	Success or ERROR_CANNOT_CREATE_OPERATION, depending on whether	The new Operation is tracked in a separate copy of the state

Current State	Trigger	Response	Next State
		the RDE Device has another slot to execute an Operation	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
	RDEOperationKill <ul style="list-style-type: none"> - discard_results flag set - any other flag set 	ERROR_INVALID_DATA	TRIGGERED
	RDEOperationKill <ul style="list-style-type: none"> - discard_results flag set - no other flag set 	ERROR_UNEXPECTED	TRIGGERED
	RDEOperationKill <ul style="list-style-type: none"> - run_to_completion flag set - discard_record flag not set 	ERROR_INVALID_DATA	TRIGGERED
	RDEOperationKill; <ul style="list-style-type: none"> - Operation executing; Operation can be killed - 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 - 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 <ul style="list-style-type: none"> - 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; <ul style="list-style-type: none"> - RDE Device cannot complete Operation within PT1 	OPERATION_TASK_RUNNING	TASK_RUNNING

Current State	Trigger	Response	Next State
	RDEOperationStatus; <ul style="list-style-type: none"> - RDE Device completes Operation within PT1 - payload to be retrieved from RDE Device or response parameters present 	Success	HAVE_RESULTS
	RDEOperationStatus; <ul style="list-style-type: none"> - RDE Device completes 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 	Success	COMPLETED
	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

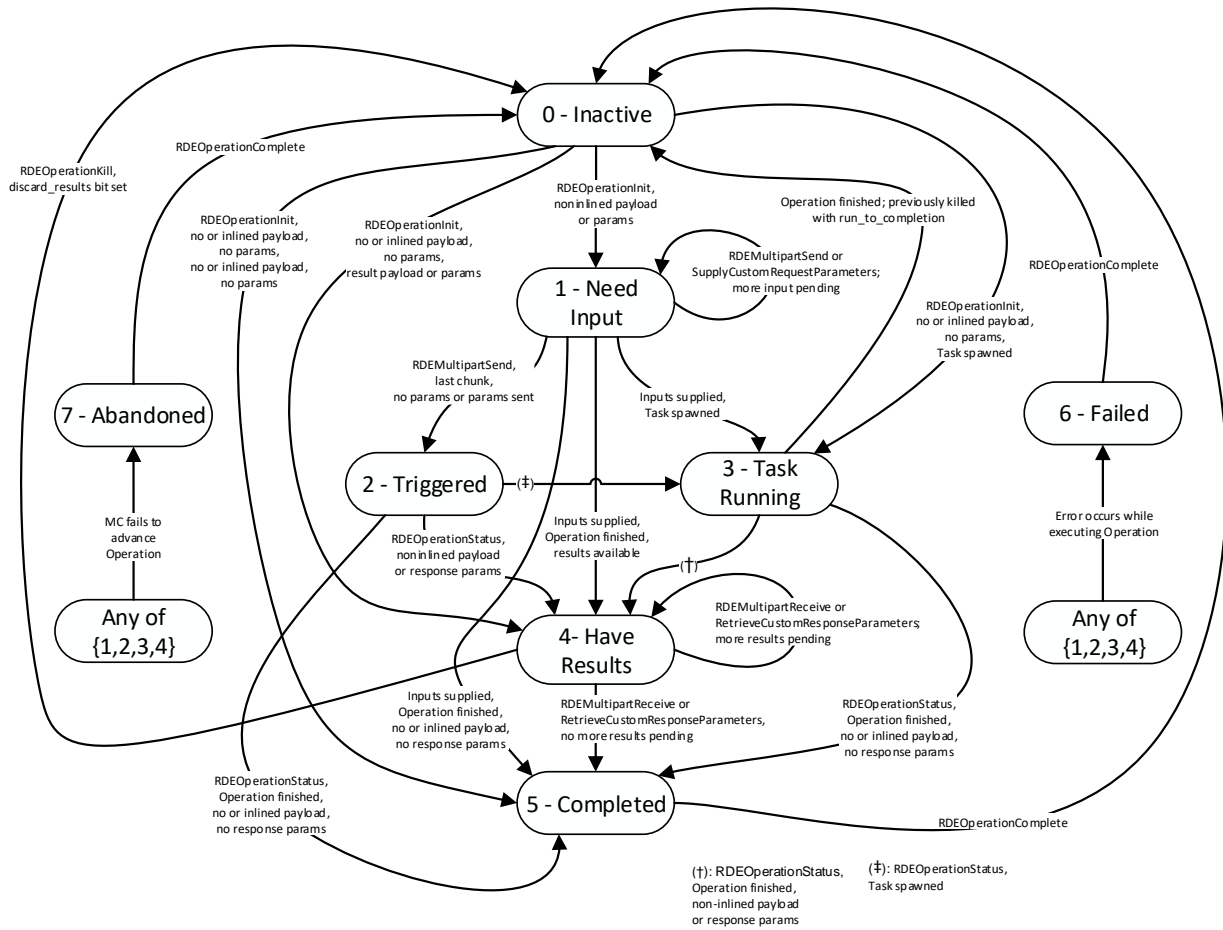
Current State	Trigger	Response	Next State
	RDEOperationKill <ul style="list-style-type: none"> - Operation executing - Operation can be killed - run_to_completion flag not set - discard_record flag set 	Success	INACTIVE
	RDEOperationKill <ul style="list-style-type: none"> - Operation executing - Operation can be killed - both run_to_completion and discard_record flags set 	Success	TASK_RUNNING
	RDEOperationKill; <ul style="list-style-type: none"> - 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; <ul style="list-style-type: none"> - 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; <ul style="list-style-type: none"> - Operation killed 	None	INACTIVE
	Execution finished; <ul style="list-style-type: none"> - Task Completion Event received by MC; - T_{abandon} timeout waiting for RDEOperationStatus command 	None	ABANDONED
	RDEOperationStatus; <ul style="list-style-type: none"> - execution not yet finished 	OPERATION_TASK_RUNNING	TASK_RUNNING
	RDEOperationStatus; <ul style="list-style-type: none"> - execution finished - payload to be retrieved from RDE Device or response parameters present 	OPERATION_HAVE_RESULTS	HAVE_RESULTS
	RDEOperationStatus; <ul style="list-style-type: none"> - execution finished - no payload or payload fits in response message such 	OPERATION_COMPLETED	COMPLETED

Current State	Trigger	Response	Next State
	<p>that total response message size is within negotiated maximum transfer chunk</p> <ul style="list-style-type: none"> - no response parameters 		
	RDEmultipartSend, RDEmultipartReceive, RDEOperationComplete	ERROR_UNEXPECTED	TASK_RUNNING
	Any other Operation command	ERROR	TASK_RUNNING
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 <ul style="list-style-type: none"> - discard_results flag set - any other flag set 	ERROR_INVALID_DATA	HAVE_RESULTS
	RDEOperationKill <ul style="list-style-type: none"> - discard_results flag set - no other flag set 	SUCCESS	INACTIVE
	RDEOperationKill <ul style="list-style-type: none"> - run_to_completion flag set - discard_record flag not set 	ERROR_INVALID_DATA	HAVE_RESULTS
	RDEOperationKill; <ul style="list-style-type: none"> - any other combination of run_to_completion and discard_record flags set 	ERROR_OPERATION_UNKILLABLE	HAVE_RESULTS
	RDEOperationStatus	OPERATION_HAVE_RESULTS	HAVE_RESULTS
	RDEmultipartReceive; <ul style="list-style-type: none"> - MC aborts transfer 	Do not send data; Success; Prepare to restart transfer with next RDEmultipartReceive command	HAVE_RESULTS
	RDEmultipartReceive; <ul style="list-style-type: none"> - transfer error 	Error specific to type of transfer failure encountered	HAVE_RESULTS (MC may retry receive or abandon Operation)
	RDEmultipartReceive; <ul style="list-style-type: none"> - more data to transfer from the RDE Device to the MC after this chunk 	Send data; Success	HAVE_RESULTS

Current State	Trigger	Response	Next State
	RDEMultipartReceive; <ul style="list-style-type: none"> - 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; <ul style="list-style-type: none"> - no more data to transfer from the RDE Device to the MC after this chunk - no response parameters present 	Send data; Success	COMPLETED
	T _{abandon} 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

Current State	Trigger	Response	Next State
	RDEOperationKill - discard_results flag set - any other flag set	ERROR_INVALID_DATA	COMPLETED
	RDEOperationKill - discard_results flag set - no other flag set	ERROR_UNEXPECTED	COMPLETED
	RDEOperationKill - run_to_completion flag set - discard_record flag not set	ERROR_INVALID_DATA	COMPLETED
	RDEOperationKill; - any other combination of run_to_completion and discard_record flags set	ERROR_OPERATION_UNKILLABLE	COMPLETED
	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

Current State	Trigger	Response	Next State
	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
	RDEOperationKill - discard_results flag set - no other flag set	ERROR_UNEXPECTED	ABANDONED
	RDEOperationKill - run_to_completion flag set - discard_record flag not set	ERROR_INVALID_DATA	ABANDONED
	RDEOperationKill; - 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



2605

2606

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

2607 **9.3 Event lifecycle**

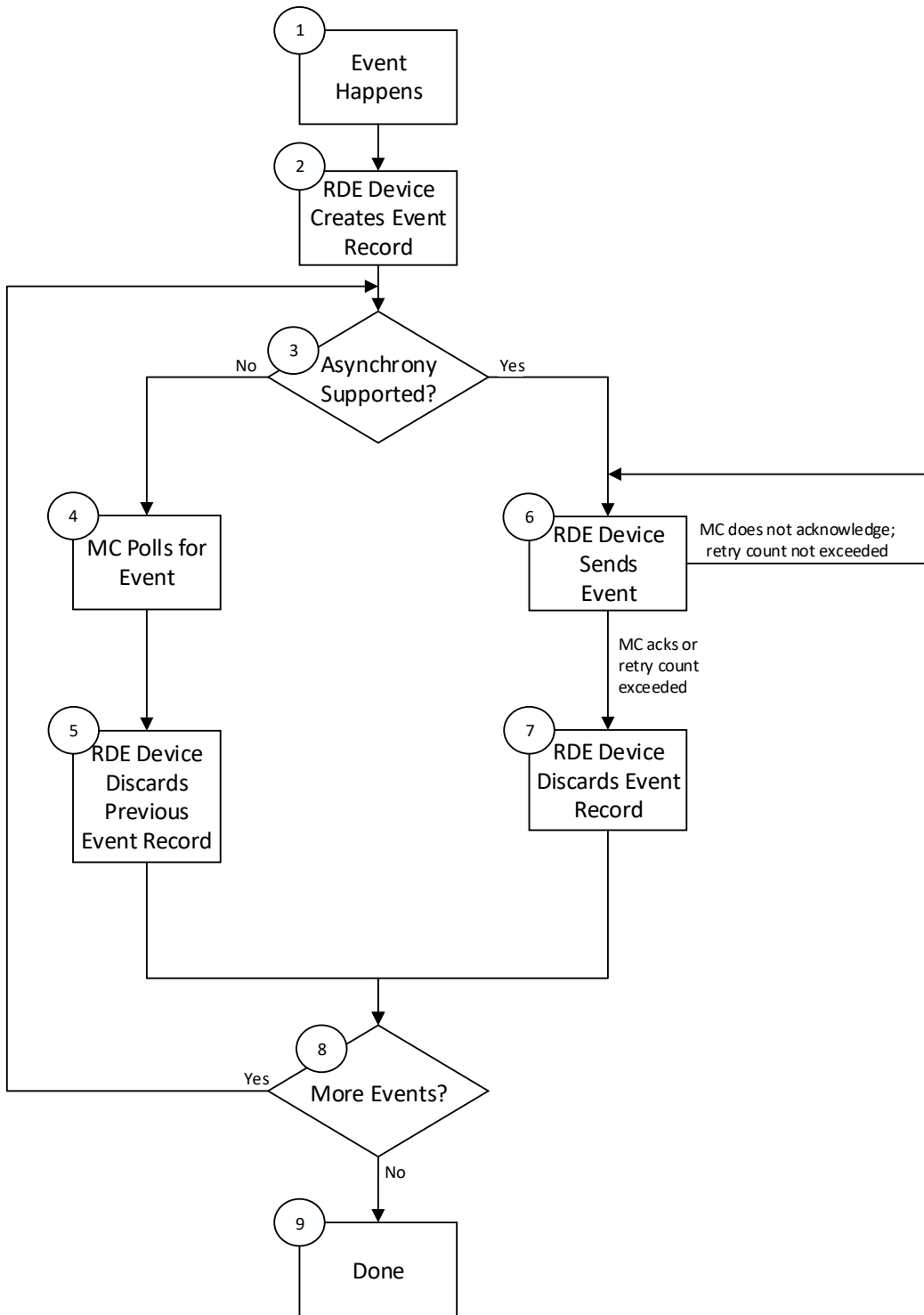
2608 Table 49 describes the operating behavior for MCs and RDE Devices over the lifecycle of Events
 2609 depicted visually in Figure 16. This sequence applies to both Task completion Events and schema-based
 2610 Events. MC and RDE Device implementations of RDE shall comply with the sequences presented here.

2611

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

Current State	Description	Condition	Next Step
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	-



2612

2613

Figure 16 – Redfish event lifecycle overview

2614 **10 PLDM commands for Redfish Device Enablement**

2615 This clause provides the list of command codes that are used by MCs and RDE Devices that implement
 2616 PLDM Redfish Device Enablement as defined in this specification. The command codes for the PLDM
 2617 messages are given in Table 50. RDE Devices and MCs shall implement all commands where the entry
 2618 in the “Command Requirement for RDE Device” or “Command Requirement for MC”, respectively, is
 2619 listed as Mandatory. RDE Devices and MCs may optionally implement any commands where the entry in
 2620 the “Command Requirement for RDE Device” or “Command Requirement for MC”, respectively, is listed
 2621 as Optional.

2622 **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
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

2623 Notes:

- 2624 1) Either RDEmultipartSend or PLDM common MultipartSend is required if the RDE Device intends to
2625 support write Operations. RDE versions of bulk transfer commands shall be used if either the RDE
2626 Device or the MC does not support PLDM common versions; if both the RDE Device and the MC
2627 advertise support for PLDM common versions of bulk transfer commands (via the PLDM Base
2628 NegotiateTransferSize command), the RDE versions shall not be used.
- 2629 2) SetEventReceiver is mandatory if the RDE Device intends to support asynchronous messaging for
2630 Events via PlatformEventMessage.
- 2631 3) RDE Devices and MCs must support either PlatformEventMessage or
2632 PollForPlatformEventMessage in order to enable Event support.
- 2633 4) SupplyCustomResponseParameter is required if the RDE Device ever sets the
2634 HaveCustomResponseParameters flag in the OperationExecutionFlags field of the response
2635 message for a triggering command.

2636 **11 PLDM for Redfish Device Enablement – Discovery and schema**
2637 **commands**

2638 This clause describes the commands that are used by RDE Devices and MCs that implement the
2639 discovery and schema management commands defined in this specification. The command codes for the
2640 PLDM messages are given in Table 50.

2641 **11.1 NegotiateRedfishParameters command (0x01) format**

2642 This command enables the MC to negotiate general Redfish parameters with an RDE Device. The MC
2643 shall send this command to the RDE Device prior to any other RDE command. An RDE Device that

2644 supports multiple mediums shall provide the same response to this command independent of the medium
 2645 on which this command was issued.

2646 When the RDE Device receives a request with data formatted per the Request Data section below, it shall
 2647 respond with data formatted per the Response Data section. For a non-SUCCESS CompletionCode, only
 2648 the CompletionCode field of the Response Data shall be returned.

2649 **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> <p>[15:9] - reserved</p> <p>[8] - BEJ v1.1 encoding and decoding supported; 1b = yes</p> <p>[7] - events_supported; 1b = yes. Must be 1b if MC supports Redfish Events or Long-running Tasks.</p> <p>[6] - action_supported; 1b = yes</p> <p>[5] - replace_supported; 1b = yes</p> <p>[4] - update_supported; 1b = yes</p> <p>[3] - delete_supported; 1b = yes</p> <p>[2] - create_supported; 1b = yes</p> <p>[1] - read_supported; 1b = yes. All MCs that implement PLDM for Redfish Device Enablement shall support read Operations</p> <p>[0] - head_supported; 1b = yes</p>
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> <p>[7:3] - reserved</p> <p>[2] - bej_1_1_support: the RDE Device supported encoding and decoding BEJ version 1.1</p> <p>[1] - expand_support: the RDE Device can process a \$expand request query parameter (expressed via the LinkExpand field of the SupplyCustomRequestParameters command)</p> <p>[0] - atomic_resource_read: the RDE Device can respond to a read of an entire resource atomically, guaranteeing consistency of the read</p>

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> <p>[15:8] - reserved</p> <p>[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</p> <p>[6] - action_supported; 1b = yes</p> <p>[5] - replace_supported; 1b = yes</p> <p>[4] - update_supported; 1b = yes</p> <p>[3] - delete_supported; 1b = yes</p> <p>[2] - create_supported; 1b = yes</p> <p>[1] - read_supported; 1b = yes. All RDE Devices shall support read Operations</p> <p>[0] - head_supported; 1b = yes</p>
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>

2650 **11.2 NegotiateMediumParameters command (0x02) format**

2651 This command enables the MC to negotiate medium-specific parameters with an RDE Device. The MC
 2652 should invoke this command on each communication medium (e.g., RBT, SMBus, PCIe VDM) on which it
 2653 intends to interface with the RDE Device. The MC shall send this command over the transport for a
 2654 particular medium to negotiate parameters for that medium. When the RDE Device receives a request
 2655 with data formatted per the Request Data section below, it shall respond with data formatted per the
 2656 Response Data section. For a non-SUCCESS CompletionCode, only the CompletionCode field of the
 2657 Response Data shall be returned.

2658

Table 52 – NegotiateMediumParameters command format

Type	Request data
uint32	<p>MCMaximumTransferChunkSizeBytes</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 MCMaximumTransferChunkSizeBytes, 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>

2659 11.3 GetSchemaDictionary command (0x03) format

2660 This command enables the MC to retrieve a dictionary (full or truncated; see clause 7.2.3) associated with
 2661 a Redfish Resource PDR. After invoking the GetSchemaDictionary command, the MC shall, upon receipt
 2662 of a successful completion code and a valid read transfer handle, invoke one or more
 2663 RDEMultipartReceive commands (clause 13.2) to transfer data for the dictionary from the RDE Device.
 2664 The MC shall only have one dictionary, schema, or message registry retrieval in process from a given
 2665 RDE Device at any time. In the event that the MC begins a dictionary, schema, or message registry
 2666 retrieval when a previous retrieval has not yet completed (i.e., more chunks of dictionary or schema data
 2667 remain to be retrieved), the previous retrieval is implicitly aborted and the RDE Device may discard any
 2668 data associated with the transfer.

2669 MCs are discouraged from invoking the GetSchemaDictionary command in the middle of processing an
 2670 RDE Operation (excluding when it is running asynchronously as a long-running task). Instead, whenever
 2671 possible, they should run the Operation back to the INACTIVE state and only then retrieve dictionaries
 2672 needed to finalize processing of Operation results. (Ideally, these dictionaries would have been cached
 2673 before the Operation was initialized.) Neither the GetSchemaDictionary command nor any
 2674 RDEMultipartReceive commands used to retrieve a dictionary shall be construed as resetting the
 2675 abandonment timer ($T_{abandon}$, see clause 7.6).

2676 When the RDE Device receives a request with data formatted per the Request Data section below, it shall
 2677 respond with data formatted per the Response Data section if it supports the command. For a non-
 2678 SUCCESS CompletionCode, only the CompletionCode field of the Response Data shall be returned.

2679

Table 53 – GetSchemaDictionary command format

Type	Request data
uint32	<p>ResourceID</p> <p>The ResourceID of any resource in the Redfish Resource PDR from which to retrieve the dictionary. A ResourceID 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</p> <p>The class of schema being requested</p>
Type	Response data
enum8	<p>CompletionCode</p> <p>value: { PLDM_BASE_CODES, ERROR_UNSUPPORTED, ERROR_NO_SUCH_RESOURCE }</p> <p>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</p> <p>The format of the dictionary as specified in the dictionary's VersionTag, defined in clause 7.2.3.2.</p>
uint32	<p>TransferHandle</p> <p>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>

2680 **11.4 GetSchemaURI command (0x04) format**

2681 This command enables the MC to retrieve the formal URI for one of the RDE Device's schemas.

2682 When the RDE Device receives a request with data formatted per the Request Data section below, it shall
 2683 respond with data formatted per the Response Data section if it supports the command. For a non-
 2684 SUCCESS CompletionCode, only the CompletionCode field of the Response Data shall be returned.

2685

Table 54 – GetSchemaURI command format

Type	Request data
uint32	ResourceID The ResourceID of a resource in a Redfish Resource PDR from which to retrieve the URI. A ResourceID 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.
schemaClass	RequestedSchemaClass The class of schema being requested
uint8	OEMExtensionNumber Shall be zero for a standard DMTF-published schema, or the one-based OEM extension to a standard schema
Type	Response data
enum8	CompletionCode value: { PLDM_BASE_CODES, ERROR_UNSUPPORTED, ERROR_NO_SUCH_RESOURCE } 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.
uint8	StringFragmentCount 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.
varstring	SchemaURI [0] URI string fragment for the schema. The reassembled string shall be the canonical URI for the JSON Schema used by the RDE Device.
...	...
varstring	SchemaURI [N - 1] URI string fragment for the schema. The reassembled string shall be the canonical URI for the JSON Schema used by the RDE Device.

2686 11.5 GetResourceETag command (0x05) format

2687 This command enables the MC to retrieve a hashed summary of the data contained immediately within a
2688 resource, including all OEM extensions to it, or of all data within an RDE Device. The retrieved ETag shall
2689 reflect the underlying data as specified in the Redfish specification ([DSP0266](#)).

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 In the event that the RDE Device cannot provide a response to this command within the PT1 time period
2694 (defined in [DSP0240](#)), the RDE Device may provide completion code ETAG_CALCULATION_ONGOING
2695 and continue the process of generating the ETag. The MC may then poll for the completed ETag by
2696 repeating the same GetResourceETag command that it gave that previously yielded this result. The RDE
2697 Device in turn shall signal whether it has completed the calculation by responding with a completion code
2698 of either SUCCESS (the calculation is done) or ETAG_CALCULATION_ONGOING (otherwise). It is
2699 recommended that the MC delay for an integer multiple of PT1 between retry attempts.

2700 Following an invocation of this command that results in a completion code of
2701 ETAG_CALCULATION_ONGOING, any other RDE command, including an invocation of
2702 GetResourceETag with a different request message, shall be interpreted by the RDE Device as implicitly

2703 canceling the pending GetResourceETag command and cause it to stop generating the ETag. The RDE
 2704 Device shall then proceed to respond to the newly arrived command normally.

2705 NOTE ETags provided via this command are not escaped for inclusion in JSON data. MCs should be aware that
 2706 performing a raw comparison of an ETag retrieved from this command with one received as part of BEJ-encoded
 2707 JSON data will result in a mismatch as the ETag format requires characters that must be escaped in JSON data.

2708 **Table 55 – GetResourceETag command format**

Type	Request data
uint32	ResourceID The ResourceID 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.

2709 **11.6 GetOEMCount command (0x06) format**

2710 This command enables the MC to retrieve the number of OEM extensions for a schema.

2711 When the RDE Device receives a request with data formatted per the Request Data section below, it shall
 2712 respond with data formatted per the Response Data section if it supports the command. For a non-
 2713 SUCCESS CompletionCode, only the CompletionCode field of the Response Data shall be returned.

2714 **Table 56 – GetOEMCount command format**

Type	Request data
uint32	ResourceID The ResourceID of the resource in the Redfish Resource PDR from which to retrieve the OEM count. A ResourceID 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. NOTE Redfish does not allow OEM extensions to Annotation and Registry schemas.
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.

2715 **11.7 GetOEMName command (0x07) format**

2716 This command enables the MC to retrieve information about the name associated with an OEM extension
 2717 to a schema (including schemas for which OEM information is available in a Redfish Resource PDR).

2718 RDE Devices shall enumerate OEM extensions in lexicographic order.

2719 When the RDE Device receives a request with data formatted per the Request Data section below, it shall
 2720 respond with data formatted per the Response Data section if it supports the command. For a non-
 2721 SUCCESS CompletionCode, only the CompletionCode field of the Response Data shall be returned.

2722 **Table 57 – GetOEMName command format**

Type	Request data
uint32	ResourceID The ResourceID of any resource in the Redfish Resource PDR from which to retrieve an OEM name. A ResourceID 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

2723 **11.8 GetRegistryCount command (0x08) format**

2724 This command enables the MC to retrieve the number of message registries supported by an RDE
 2725 Device.

2726 When the RDE Device receives a request with data formatted per the Request Data section below, it shall
 2727 respond with data formatted per the Response Data section if it supports the command. For a non-
 2728 SUCCESS CompletionCode, only the CompletionCode field of the Response Data shall be returned.

2729 **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

2730 **11.9 GetRegistryDetails command (0x09) format**

2731 This command enables the MC to retrieve information about a message registry an RDE Device supports.

2732 RDE Devices shall enumerate message registries in lexicographic order and return message registry
 2733 versions in reverse numeric order (most recent versions listed first). The RDE Device shall truncate the
 2734 list and decrease the count as needed to ensure that the response message fits within the negotiated
 2735 message size, thereby omitting mention of support for older versions.

2736 When the RDE Device receives a request with data formatted per the Request Data section below, it shall
 2737 respond with data formatted per the Response Data section if it supports the command. For a non-
 2738 SUCCESS CompletionCode, only the CompletionCode field of the Response Data shall be returned.

2739 **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

2740 **11.10 SelectRegistryVersion command (0x0A) format**

2741 This command enables the MC to specify the version of a supported Redfish message registry that the
 2742 RDE device should use. By default, the RDE Device shall utilize the latest version of the registry that it
 2743 supports.

2744 When the RDE Device receives a request with data formatted per the Request Data section below, it shall
 2745 respond with data formatted per the Response Data section if it supports the command.

2746 **Table 60 – SelectRegistryVersion command format**

Type	Request data
uint8	RegistryIndex 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.
ver32	RegistryVersion Version of the registry to be used
Type	Response data
enum8	CompletionCode value: { PLDM_BASE_CODES } ERROR_INVALID_DATA: The supplied index does not correspond to a supported registry or the supplied version is not supported

2747 **11.11 GetMessageRegistry command (0x0B) format**

2748 This command enables the MC to retrieve the formal JSON registry for a Redfish message registry
 2749 supported by the RDE device. After invoking the GetMessageRegistry command, the MC shall, upon
 2750 receipt of a successful completion code and a valid read transfer handle, invoke one or more
 2751 RDEMultipartReceive commands (clause 13.2) to transfer data for the registry from the RDE Device. The
 2752 MC shall only have one dictionary, schema, or message registry retrieval in process from a given RDE
 2753 Device at any time. In the event that the MC begins a dictionary, schema, or message registry retrieval
 2754 when a previous retrieval has not yet completed (i.e., more chunks of dictionary or schema data remain to
 2755 be retrieved), the previous retrieval is implicitly aborted and the RDE Device may discard any data
 2756 associated with the transfer.

2757 When the RDE Device receives a request with data formatted per the Request Data section below, it shall
 2758 respond with data formatted per the Response Data section if it supports the command. For a non-
 2759 SUCCESS CompletionCode, only the CompletionCode field of the Response Data shall be returned.

2760 **Table 61 – GetMessageRegistry command format**

Type	Request data
uint8	RegistryIndex 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.
Type	Response data
enum8	CompletionCode value: { PLDM_BASE_CODES } ERROR_INVALID_DATA: The supplied index does not correspond to a supported registry

Type	Response data (continued)
uint8	<p>SchemaFormat</p> <p>Bitwise OR of two values:</p> <p>Text format: { RAW_UTF8 = 0; GZIP_UTF8 = 1 }</p> <p>Schema format: { JSON = 0x10; CSDL = 0x20; YAML = 0x30 }</p> <p>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</p> <p>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>

2761 **11.12 GetSchemaFile command (0x0C) format**

2762 This command enables the MC to retrieve the formal schema for a Redfish resource supported by the
 2763 RDE device. After invoking the GetSchemaFile command, the MC shall, upon receipt of a successful
 2764 completion code and a valid read transfer handle, invoke one or more RDEMultipartReceive commands
 2765 (clause 13.2) to transfer data for the schema from the RDE Device. The MC shall only have one
 2766 dictionary, schema, or message registry retrieval in process from a given RDE Device at any time. In the
 2767 event that the MC begins a dictionary, schema, or message registry retrieval when a previous retrieval
 2768 has not yet completed (i.e., more chunks of dictionary or schema data remain to be retrieved), the
 2769 previous retrieval is implicitly aborted and the RDE Device may discard any data associated with the
 2770 transfer. MCs should reference the version and signature of schemas, as documented in Redfish
 2771 Resource PDRs, wherever possible to avoid duplicate download of schema files.

2772 When the RDE Device receives a request with data formatted per the Request Data section below, it shall
 2773 respond with data formatted per the Response Data section if it supports the command. For a non-
 2774 SUCCESS CompletionCode, only the CompletionCode field of the Response Data shall be returned.

2775 **Table 62 – GetSchemaFile command format**

Type	Request data
uint32	<p>ResourceID</p> <p>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</p> <p>The class of schema being requested</p>
uint8	<p>OEMOffset</p> <p>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</p> <p>value: { PLDM_BASE_CODES, ERROR_NO_SUCH_RESOURCE } ERROR_INVALID_DATA: The supplied OEMOffset is not valid</p>
uint8	<p>SchemaFormat</p> <p>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</p> <p>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>

2776 12 PLDM for Redfish Device Enablement – RDE Operation and Task 2777 commands

2778 This clause describes the Task commands that are used by RDE Devices and MCs that implement
2779 Redfish Device Enablement as defined in this specification. The command numbers for the PLDM
2780 messages are given in Table 50.

2781 12.1 RDEOperationInit command (0x10) format

2782 This command enables the MC to initiate a Redfish Operation with an RDE Device on behalf of a client.
2783 After invoking the RDEOperationInit command, the MC may, upon receipt of a successful completion
2784 code, invoke one or more RDEMultipartSend commands (clause 13.1) to transfer payload data of type
2785 bejEncoding to the RDE Device. The MC shall only use RDEMultipartSend to transfer the payload data if
2786 that data cannot fit in the request message of the RDEOperationInit command. After any payload has
2787 been transferred, the MC may invoke the SupplyCustomRequestParameters command if additional
2788 parameters are required. See clause 9 for more details on the Operation lifecycle.

2789 After the RDE Device receives the RDEOperationInit command, if flags are not set to indicate that it
2790 should expect either payload data or custom request parameters, the RDE Device is triggered and shall
2791 begin execution of the Operation. Similarly, if the flags are set to expect a payload but not parameters,
2792 and the payload is contained inline in the request message, the RDE Device is implicitly triggered and
2793 shall begin execution of the Operation.

2794 If triggered, the RDE Device shall respond with results if it is able to complete the Operation within the
2795 time period required for a response to this message. If there is a response payload that fits within the
2796 ResponsePayload field while maintaining a message size compatible with the negotiated maximum chunk
2797 size (see NegotiateMediumParameters, clause 11.2), the RDE Device shall include it within this
2798 response. Only if including a response payload would cause the message to exceed the negotiated chunk
2799 size may the RDE Device flag it for transfer via RDEMultipartReceive.

2800 When the RDE Device receives a request with data formatted per the Request Data section below, it shall
2801 respond with data formatted per the Response Data section. Even with a non-SUCCESS
2802 CompletionCode, all fields of the Response Data shall be returned.

Table 63 – RDEOperationInit command format

Type	Request data
uint32	<p>ResourceID</p> <p>The resourceID of a resource in the Redfish Resource PDR for the data that is the target of this operation</p>
rdeOpID	<p>OperationID</p> <p>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</p> <p>The type of Redfish Operation being performed.</p> <p>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</p> <p>Flags associated with this Operation:</p> <p>[7:4] - reserved for future use</p> <p>[3] - excerpt_flag; if 1b, the RDE Device should perform an excerpt read (see 7.2.4.3.6)</p> <p>[2] - contains_custom_request_parameters; if 1b, the RDE Device should expect to receive a SupplyCustomRequestParameters command request before it may trigger the Operation</p> <p>[1] - contains_request_payload; if 0b, the Operation does not require data to be sent</p> <p>[0] - locator_valid; if 0b, the locator in the OperationLocator field shall be ignored</p>
uint32	<p>SendDataTransferHandle</p> <p>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</p> <p>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</p> <p>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</p> <p>BEJ locator indicating where the new Operation is to take place within the resource specified in ResourceID. May not be supported for all Operations. This field shall be omitted if the OperationLocatorLength field above is set to zero.</p>
null or bejEncoding	<p>RequestPayload</p> <p>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.</p>

Type	Response data
enum8	<p>CompletionCode</p> <p>value: { PLDM_BASE_CODES, ERROR_CANNOT_CREATE_OPERATION, ERROR_NOT_ALLOWED, ERROR_WRONG_LOCATION_TYPE, ERROR_OPERATION_EXISTS, ERROR_UNSUPPORTED, ERROR_NO_SUCH_RESOURCE }</p> <p>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.</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> <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> <p>[7: 6] - reserved for future use</p> <p>[5] - head access; 1b = access allowed</p> <p>[4] - delete access; 1b = access allowed</p> <p>[3] - create access; 1b = access allowed</p> <p>[2] - replace access; 1b = access allowed</p> <p>[1] - update access; 1b = access allowed</p> <p>[0] - read access; 1b = access allowed</p> <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>

2804 **12.2 SupplyCustomRequestParameters command (0x11) format**

2805 This command enables the MC to send custom HTTP/HTTPS X- headers and other uncommon request
 2806 parameters to an RDE Device to be applied to an Operation if the client’s HTTP operation contains any
 2807 such parameters. The MC must not use this command to submit any headers for which a standard
 2808 handling is defined in either this specification or [DSP0266](#). If the client’s HTTP operation does not contain
 2809 the parameters conveyed in this command, the MC shall not send this command as part of its processing
 2810 of the Operation.

2811 The MC shall only invoke this command in the event that at least one custom header or uncommon
 2812 request parameter needs to be transferred to the RDE Device. When sent, the

2813 **SupplyCustomRequestParameters** command shall be invoked after the MC sends the
2814 RDEOperationInit command.

2815 After the RDE Device receives the SupplyCustomRequestParameters command, if flags from the original
2816 RDEOperationInit command (see clause 12.1) were not set to indicate that it should expect payload data
2817 or if the RDE Device has already received payload data, the RDE Device shall consider itself triggered
2818 and begin execution of the Operation.

2819 If triggered, the RDE Device shall respond with results if it is able to complete the Operation within the
2820 time period required for a response to this message. If there is a response payload that fits within the
2821 ResponsePayload field while maintaining a message size compatible with the negotiated maximum chunk
2822 size (see clause 11.2), the RDE Device shall include it within this response. Only if including a response
2823 payload would cause the message to exceed the negotiated chunk size may the RDE Device flag it for
2824 transfer via RDEmultipartReceive.

2825 The size of the request message is limited to the negotiated maximum chunk size (see clause 11.2). If the
2826 client supplied sufficiently many custom request headers and/or ETags that the request message would
2827 exceed this negotiated size, the MC shall abort the request and perform the following steps:

- 2828 1) Use the RDEOperationKill (see clause 12.6) and then RDEOperationComplete (see clause
2829 12.4) commands to abort and finalize the Operation if it had already been initiated via
2830 RDEOperationInit (see clause 12.1).
- 2831 2) Return to the client HTTP/HTTPS error code 431, Request Header Fields Too Large.
- 2832 3) Cease processing of the client request.

2833 When the RDE Device receives a request with data formatted per the Request Data section below, it shall
2834 respond with data formatted per the Response Data section. Even with a non-SUCCESS
2835 CompletionCode, all fields of the Response Data shall be returned.

2836 **Table 64 – SupplyCustomRequestParameters command format**

Type	Request data
uint32	ResourceID The resourceID of a resource in the Redfish Resource PDR for the instance to which custom headers should be supplied
rdeOpID	OperationID Identification number for this Operation; must match the one used for all commands relating to this Operation.
uint16	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. 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. 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 bejResourceLinkExpansion format specification.

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 7.2.4.3.1.</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> <p>[7:6] - reserved for future use</p> <p>[5] - head access; 1b = access allowed</p> <p>[4] - delete access; 1b = access allowed</p> <p>[3] - create access; 1b = access allowed</p> <p>[2] - replace access; 1b = access allowed</p> <p>[1] - update access; 1b = access allowed</p> <p>[0] - read access; 1b = access allowed</p> <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>

2837 **12.3 RetrieveCustomResponseParameters command (0x12) format**

2838 This command enables the MC to retrieve custom HTTP/HTTPS headers or other uncommon response
 2839 parameters from an RDE Device to be forwarded to the client that initiated a Redfish operation. The MC
 2840 shall only invoke this command when the **HaveCustomResponseParameters** flag in the response
 2841 message for a triggered RDE command indicates that it is needed.

2842 The RDE Device shall not supply more response headers than would allow the response message to fit in
 2843 the negotiated maximum transfer chunk size (see clause 11.2).

2844 When the RDE Device receives a request with data formatted per the Request Data section below, it shall
 2845 respond with data formatted per the Response Data section. For a non-SUCCESS CompletionCode, only
 2846 the CompletionCode field of the Response Data shall be returned.

2847 **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
...	...

2848 12.4 RDEOperationComplete command (0x13) format

2849 This command enables the MC to inform an RDE Device that it considers an Operation to be complete,
 2850 including failed and abandoned Operations. The RDE Device in turn may discard any internal records for
 2851 the Operation.

2852 When the RDE Device receives a request with data formatted per the Request Data section below, it shall
 2853 respond with data formatted per the Response Data section.

2854

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 }

2855 **12.5 RDEOperationStatus command (0x14) format**

2856 This command enables the MC to query an RDE Device for the status of an Operation. It is additionally
 2857 used to collect the initial response when an RDE Operation is triggered by a RDEmultipartSend command
 2858 or after a Task finishes asynchronous execution.

2859 When providing result data for an Operation that has finished executing, if there is a response payload
 2860 that fits within the ResponsePayload field while maintaining a message size compatible with the
 2861 negotiated maximum chunk size (see NegotiateMediumParameters, clause 11.2), the RDE Device shall
 2862 include it within this response. Only if including a response payload would cause the message to exceed
 2863 the negotiated chunk size may the RDE Device flag it for transfer via RDEmultipartReceive.

2864 When the RDE Device receives a request with data formatted per the Request Data section below, it shall
 2865 respond with data formatted per the Response Data section. Even with a non-SUCCESS
 2866 CompletionCode, all fields of the Response Data shall be returned.

2867

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
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> <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>

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> <p>[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> <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
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>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>

Type	Response data (continued)
null or bejEncoding	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.

2868 12.6 RDEOperationKill command (0x15) format

2869 This command enables the MC to request that an RDE Device terminate an Operation. The RDE Device
2870 shall kill the Operation if the Operation can be killed; however, the MC must be aware that not all
2871 Operations can be terminated.

2872 When the RDE Device receives a request with data formatted per the Request Data section below, it shall
2873 respond with data formatted per the Response Data section if it supports the command.

2874 **Table 68 – RDEOperationKill 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
bitfield8	KillFlags Flags for killing the Operation: [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	CompletionCode value: { PLDM_BASE_CODES, ERROR_OPERATION_ABANDONED, ERROR_OPERATION_FAILED, ERROR_OPERATION_UNKILLABLE, ERROR_NO_SUCH_RESOURCE, ERROR_UNEXPECTED }

2875 12.7 RDEOperationEnumerate command (0x16) format

2876 This command enables the MC to request that an RDE Device enumerate all Operations that are
2877 currently active (not in state INACTIVE in the Operation lifecycle state machine of clause 9.2.3.2). It is

2878 expected that the MC will typically use this command during its initialization to discover any Operations
 2879 that spawned Tasks that were active through a shutdown.

2880 NOTE When instantiating Operations, the RDE Device shall not create a new Operation if including the total data for
 2881 all Operations would cause the response message for this command to exceed the negotiated maximum transfer
 2882 chunk size (see clause 11.2) for any of the mediums on which the MC has communicated with the RDE Device.

2883 If the RDE Device accepts operations from protocols other than Redfish, it should make them visible as
 2884 RDE Operations while they are active by enumerating them in response to this command.

2885 When the RDE Device receives a request with data formatted per the Request Data section below, it shall
 2886 respond with data formatted per the Response Data section if it supports the command. For a non-
 2887 SUCCESS CompletionCode, only the CompletionCode field of the Response Data shall be returned.

2888 **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 }

2889 **13 PLDM for Redfish Device Enablement – Utility commands**2890 **13.1 RDEmultipartSend command (0x30) format**

2891 This command enables the MC to send a large volume of data to an RDE Device. In the event of a data
 2892 checksum error, the MC may reissue the first RDEmultipartSend command with the initial data transfer
 2893 handle; the RDE Device shall recognize this to mean that the transfer failed and respond as if this were
 2894 the first transfer attempt. If the MC chooses not to restart the transfer, or in any other error occurs, the MC
 2895 should abandon the transfer. In the latter case, if the transfer is part of an Operation, the MC shall
 2896 explicitly abort and then finalize the Operation via the RDEOperationKill and RDEOperationComplete
 2897 commands (see clauses 12.6 and 12.4).

2898 Similarly, in the event of transient transfer errors for individual chunks of the data, the MC may retry those
 2899 chunks by reissuing the RDEmultipartSend command corresponding to those chunks provided it has not
 2900 yet issued a RDEmultipartSend command for a subsequent chunk. When the RDE Device receives a
 2901 request with data formatted per the Request Data section below, it shall respond with data formatted per
 2902 the Response Data section. For a non-SUCCESS CompletionCode, only the CompletionCode field of the
 2903 Response Data shall be returned.

2904 NOTE In versions of this specification prior to v1.1.0, this command was named MultipartSend.

2905 **Table 70 – RDEmultipartSend command format**

Type	Request data
uint32	<p>DataTransferHandle</p> <p>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.</p> <p>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</p> <p>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</p> <p>An indication of current progress within the transfer. The value START_AND_END indicates that the entire transfer consists of a single chunk.</p> <p>value: { START = 0, MIDDLE = 1, END = 2, START_AND_END = 3 }</p>
uint32	<p>NextDataTransferHandle</p> <p>The handle for the next chunk of data for this transfer; zero (0x00000000) if no further data.</p>
uint32	<p>DataLengthBytes</p> <p>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]</p> <p>The first byte of the current chunk of data. Shall be omitted if only the DataIntegrityChecksum is present.</p>
...	...
uint8	<p>Data [N-1]</p> <p>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>

2906 **13.2 RDEMultiPartReceive command (0x31) format**

2907 This command enables the MC to receive a large volume of data from an RDE Device. In the event of a
 2908 data checksum error, the MC may reissue the first RDEMultiPartReceive command with the initial data
 2909 transfer handle; the RDE Device shall recognize this to mean that the transfer failed and respond as if this
 2910 were the first transfer attempt. If the MC chooses not to restart the transfer, or in any other error occurs,
 2911 the MC should abandon the transfer. In the latter case, if the transfer is part of an Operation, the MC shall
 2912 explicitly abort and finalize the Operation via the RDEOperationKill and then RDEOperationComplete
 2913 commands (see clauses 12.6 and 12.4).

2914 Similarly, in the event of transient transfer errors for individual chunks of the data, the MC may retry those
 2915 chunks by reissuing the RDEMultiPartReceive command corresponding to those chunks provided it has
 2916 not yet issued a RDEMultiPartReceive command for a subsequent chunk.

2917 When the RDE Device receives a request with data formatted per the Request Data section below, it shall
 2918 respond with data formatted per the Response Data section if it supports the command. For a non-
 2919 SUCCESS CompletionCode, only the CompletionCode field of the Response Data shall be returned.

2920 NOTE In versions of this specification prior to v1.1.0, this command was named MultiPartReceive.

2921

Table 71 – RDEMultiPartReceive command format

Type	Request data
uint32	<p>DataTransferHandle</p> <p>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</p> <p>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</p> <p>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</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> <p>If the transfer is aborted, the RDE Device shall acknowledge this status by returning SUCCESS.</p>
enum8	<p>TransferFlag</p> <p>value: { START = 0, MIDDLE = 1, END = 2, START_AND_END = 3 }</p> <p>This field shall be omitted for a non-SUCCESS CompletionCode or if the transfer has been aborted</p>
uint32	<p>NextDataTransferHandle</p> <p>The handle for the next chunk of data for this transfer; zero (0x00000000) if no further data</p> <p>This field shall be omitted for a non-SUCCESS CompletionCode or if the transfer has been aborted</p>
uint32	<p>DataLengthBytes</p> <p>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).</p> <p>This field shall be omitted for a non-SUCCESS CompletionCode or if the transfer has been aborted</p>
uint8	<p>Data [0]</p> <p>The first byte of current chunk of data. Shall be omitted if only the DataIntegrityChecksum is present.</p> <p>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]</p> <p>The last byte of the current chunk of data. Shall be omitted if only the DataIntegrityChecksum is present.</p> <p>This field shall be omitted for a non-SUCCESS CompletionCode or if the transfer has been aborted</p>
uint32	<p>DataIntegrityChecksum</p> <p>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.</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>

2922 **14 Additional Information**

2923 **14.1 RDE Multipart transfers**

2924 The various commands defined in clauses 10 and 12 support bulk transfers via the RDEMultipartSend
 2925 and RDEMultipartReceive commands defined in clause 13. The RDEMultipartSend and
 2926 RDEMultipartReceive commands use flags and data transfer handles to perform multipart transfers. A
 2927 data transfer handle uniquely identifies the next part of the transfer. The data transfer handle values are
 2928 implementation specific. For example, an implementation can use memory addresses or sequence
 2929 numbers as data transfer handles.

2930 NOTE If both the RDE Device and the MC support use of PLDM common multipart transfers, those versions of the
 2931 commands shall be used in lieu of the RDE versions. The following notes apply:

- 2932 • All transfers shall consist of a single portion, beginning at offset zero and transferring the entire buffer
- 2933 • The TransferContext field, which is defined in [DSP0240](#) to be protocol specific, shall be supplied with the
 2934 OperationID that would have been used with an RDE version of a multipart transfer
- 2935 • Handling of aborted transfers, which is defined in [DSP0240](#) to be protocol specific, shall follow the notes
 2936 provided within this specification for multipart transfers.

2937 **14.1.1 Flag usage for RDEMultipartSend**

2938 The following list shows some requirements for using TransferOperationFlag, TransferFlag, and
 2939 DataTransferHandle in RDEMultipartSend data transfers:

- 2940 • To prepare a large data send for use in an RDE command, a DataTransferHandle shall be sent
 2941 by the MC in the request message of the RDEOperationInit command.
- 2942 • To reflect a data transfer (re)initiated with a RDEMultipartSend command, the
 2943 TransferOperation shall be set to XFER_FIRST_PART in the response message.
- 2944 • For transferring a part after the first part of data, the TransferOperation shall be set to
 2945 XFER_NEXT_PART and the DataTransferHandle shall be set to the NextDataTransferHandle
 2946 that was obtained in the request for the previous RDEMultipartSend command for this data
 2947 transfer.

- 2948
- 2949
- The TransferFlag specified in the request for a RDEMultipartSend command has the following meanings:
 - 2950 – START, which is the first part of the data transfer
 - 2951 – MIDDLE, which is neither the first nor the last part of the data transfer
 - 2952 – END, which is the last part of the data transfer
 - 2953 – START_AND_END, which is the first and the last part of the data transfer. In this case, the transfer consists of a single chunk
 - 2954
 - For a RDEMultipartSend, the requester shall consider a data transfer complete when it receives a success CompletionCode in the response to a request in which the TransferFlag was set to End or StartAndEnd.
 - 2955
 - 2956
 - 2957

2958 14.1.2 Flag usage for RDEMultipartReceive

2959 The following list shows some requirements for using TransferOperationFlag, TransferFlag, and
2960 DataTransferHandle in RDEMultipartReceive data transfers:

- 2961 • To prepare a large data transfer receive for use in an RDE command, a DataTransferHandle
2962 shall be sent by the RDE Device in the response message to the RDEOperationInit,
2963 SupplyCustomRequestParameters, or RDEOperationStatus command after an Operation has
2964 finished execution and results are ready for pick-up.
- 2965 • To initiate a data transfer with a RDEMultipartReceive command, the TransferOperation shall
2966 be set to XFER_FIRST_PART in the request message.
- 2967 • For transferring a part after the first part of data, the TransferOperation shall be set to
2968 XFER_NEXT_PART and the DataTransferHandle shall be set to the NextDataTransferHandle
2969 that was obtained in the response to the previous RDEMultipartReceive command for this data
2970 transfer.
- 2971 • The TransferFlag specified in the response of a RDEMultipartReceive command has the
2972 following meanings:
 - 2973 – START, which is the first part of the data transfer
 - 2974 – MIDDLE, which is neither the first nor the last part of the data transfer
 - 2975 – END, which is the last part of the data transfer
 - 2976 – START_AND_END, which is the first and the last part of the data transfer
- 2977 • For a RDEMultipartReceive, the requester and responder shall consider a data transfer
2978 complete when the TransferFlag in the response is set to END or START_AND_END. After this
2979 point, the transfer may not be restarted without repeating the invoking commands, such as
2980 GetSchemaDictionary for a multipart transfer of a dictionary.

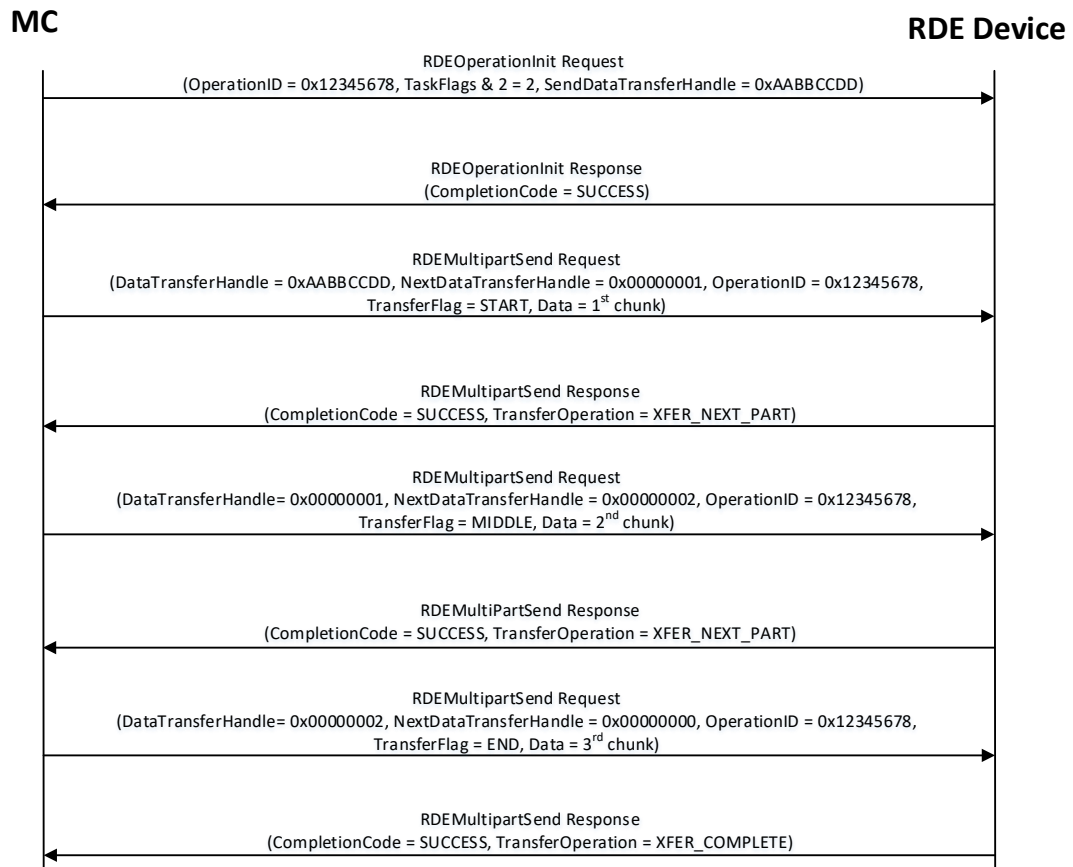
2981 14.1.3 RDE Multipart transfer examples

2982 The following examples show how the multipart transfers can be performed using the generic mechanism
2983 defined in the commands.

2984 In the first example, the MC sends data to the RDE Device as part of a Redfish Update operation.
2985 Following the RDEOperationInit command sequence, the MC effects the transfer via a series of
2986 RDEMultipartSend commands. Figure 17 shows the flow of the data transfer.

2987 In the second example, the MC retrieves the dictionary for a schema. The request is initiated via the
2988 GetSchemaDictionary command and then effected via one or more RDEMultipartReceive commands.

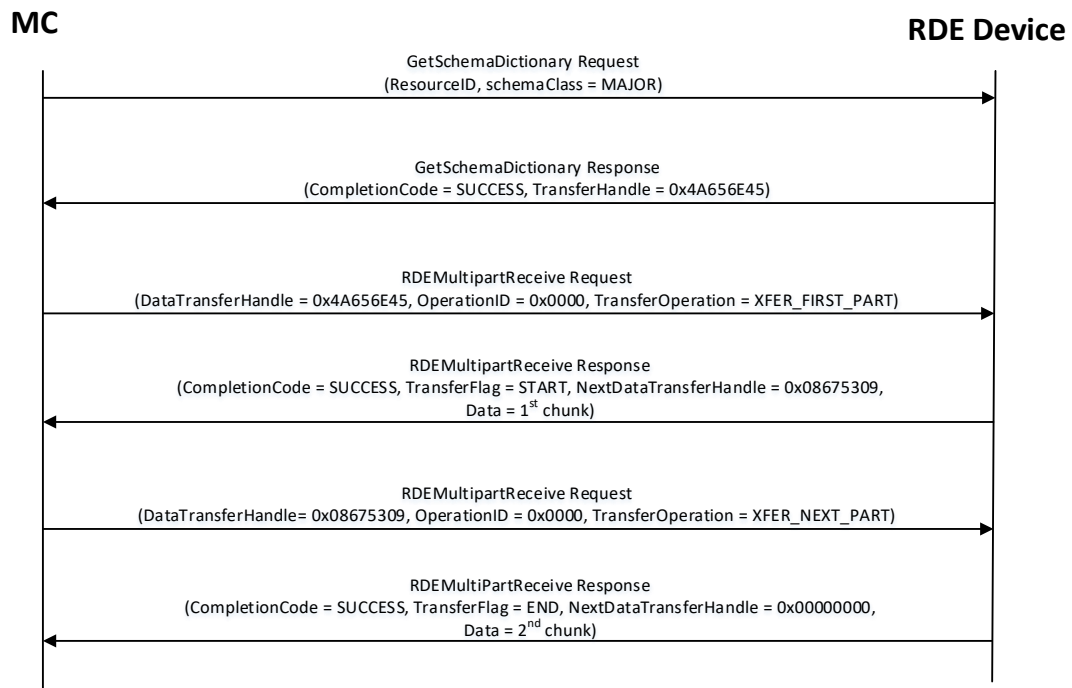
2989 Figure 18 shows the flow of the data transfer.



2990

2991

Figure 17 – RDEMultiPartSend example



2992

2993

Figure 18 – RDEMultiPartReceive example

2994 **14.2 Implementation notes**

2995 Several implementation notes apply to manufacturers of RDE Devices or of management controllers.

2996 **14.2.1 Schema updates**

2997 If one or more schemas for an RDE Device are updated, the RDE Device may communicate this to the
 2998 MC by triggering an event for the affected PDRs. When the MC detects a PDR update, it shall reread the
 2999 affected PDRs.

3000 **14.2.2 Storage of dictionaries**

3001 It is not necessary for the MC to maintain all dictionaries in memory at any given time. It may flush
 3002 dictionaries at will since they can be retrieved on demand from the RDE Devices via the
 3003 GetSchemaDictionary command (clause 11.2). However, if the MC has to retrieve a dictionary “on
 3004 demand” to support a Redfish query, this will likely incur a performance delay in responding to the client.
 3005 For MCs with highly limited memory that cannot retain all the dictionaries they need to support, care must
 3006 thus be exercised in the runtime selection of dictionaries to evict. Such caching considerations are
 3007 outside the scope of this specification.

3008 **14.2.3 Dictionaries for related schemas**

3009 MCs must not assume that sibling instances of Redfish Resource PDRs in a hierarchy (such as collection
 3010 members) use the same version of a schema. They could, for example, correspond to individual elements
 3011 from an array of hardware (such as a disk array) built by separate manufacturers and supporting different
 3012 versions of a major schema or with different OEM extensions to it. However, at such time as the MC has
 3013 verified that two siblings do in fact use the same schemas, there is no reason to store multiple copies of
 3014 the dictionary corresponding to that schema. Of course, sibling instances of resources stored within the

3015 same PDR share all dictionaries; it is only with instances of resources from separate PDRs that this
3016 applies.

3017 Similarly, it is expected to be fairly commonplace that the system managed by an MC could have multiple
3018 RDE Devices of the same class, such as multiple network adapters or multiple RAID array controllers. In
3019 such cases, however, there is no guarantee that each such RDE Device will support the same version of
3020 any given Redfish schema.

3021 To handle such cases, MCs have two choices. The most straightforward approach is to simply maintain
3022 each dictionary associated with the RDE Device it came from. This of course has space implications. A
3023 more practical approach is to store one copy of the dictionary for each version of the schema and then
3024 keep track of which version of the dictionary to use with which RDE Device. Because RDE Devices may
3025 support only subsets of the properties in resources, care must be taken when employing this approach to
3026 ensure that all supported properties are covered in the dictionaries selected. This may be done by
3027 merging dictionaries at runtime, though details of how to merge dictionaries are out of scope for this
3028 specification. In particular, OEM sections of dictionaries are not generally able to be merged as the
3029 sequence numbers for the names of the different OEM extensions themselves are likely to overlap.

3030 However, an even better approach is available. In Redfish schemas, so long as only the minor and
3031 release version numbers change, schemas are required to be fully backward compatible with earlier
3032 revisions. Individual properties and enumeration values may be added but never removed. The MC can
3033 therefore leverage this to retain only the newest instance of dictionary for each major version supported
3034 by RDE Devices. Again, the fact that RDE Devices may support only subsets of the properties in a
3035 resource means that care must be taken to ensure dictionary support for all the properties used across all
3036 RDE Devices that implement any given schema.

3037 **14.2.4 [MC] HTTP/HTTPS POST Operations**

3038 As specified in [DSP0266](#), a Redfish POST Operation can represent either a Create Operation or an
3039 Action. To distinguish between these cases, the MC may examine the URI target supplied with the
3040 operation. If it points to a collection, the MC may assume that the Operation is a Create; if it points to an
3041 action, the MC may assume the Operation is an Action. Alternatively, the MC may presuppose that the
3042 POST is a Create Operation and if it receives an ERROR_WRONG_LOCATION_TYPE error code from
3043 the RDE Device, retry the Operation as an Action. This second approach reduces the amount of URI
3044 inspection the MC has to perform in order to proxy the Operation at the cost of a small delay in
3045 completion time for the Action case. (The supposition that POSTs correspond to Create Operations could
3046 of course be reversed, but it is expected that Actions will be much rarer than Create Operations.)
3047 Implementers should be aware that such delays could cause a client-side timeout.

3048 Another clue that could be used to differentiate between POSTs intended as create operations vs POSTs
3049 intended as actions would be trial encodings of supplied payload data. If there is no payload data, then
3050 the request is either in error or an action. In this case, the payload should be encoded with the dictionary
3051 for the major schema associated with target resource. On the other hand, if the payload is intended for a
3052 create operation, the correct dictionary to use would be the collection member dictionary, which may be
3053 retrieved via the GetSchemaDictionary command (clause 11.2), specifying
3054 COLLECTION_MEMBER_TYPE as the dictionary to retrieve.

3055 **14.2.4.1 Support for Actions**

3056 When a Redfish client issues a Redfish Operation for an Action, the URI target for the Action will be a
3057 POST of the form /redfish/v1/{path to root of RDE Device component}/{path to RDE Device owned
3058 resource}/Actions/schema_name.action_name. To process this, the MC may translate {path to root of
3059 RDE Device component} and {path to RDE Device owned resource} normally to identify the PDR against
3060 which the Operation should be executed. (If the URI is not in this format, this is another indication that the
3061 POST operation is probably a CREATE.) After it has performed this step, the MC can then check its PDR
3062 hierarchy to find the Redfish Action PDR containing an action named schema_name.action_name. If it

3063 doesn't find one, the MC shall respond with HTTP status code 404, Not Found and stop processing the
3064 Operation.

3065 After the correct Action is located, the MC can translate any request parameters supplied with the Action.
3066 To do so, it should look within the dictionary at the point beginning with the named action, and then
3067 navigate into the Parameters set under the action. From there, standard encoding rules apply. When
3068 supplying a locator for the Action to the RDE Device as part of the RDEOperationInit command, the MC
3069 shall not include the Parameters set as one of the sequence numbers comprising the locator; rather, it
3070 shall stop with the sequence number for the property corresponding to the Action's name.

3071 After the Action is complete, it may contain result parameters. If present, definitions for these will be found
3072 in the dictionary in a ReturnType set parallel to the Parameters set that contained any request
3073 parameters. If an Action does not contain explicit result parameters, the ReturnType set will generally not
3074 be present in the dictionary. The structure of the ReturnType set mirrors exactly that of the Parameters
3075 set.

3076 **14.2.5 Consistency checking of read Operations**

3077 Because the collection of data contained within a schema cannot generally be read atomically by RDE
3078 Devices, issues of consistency arise. In particular, if the RDE Device reads some of the data, performs an
3079 update, and then reads more data, there is no guarantee that data read in the separate "chunks" will be
3080 mutually consistent. While the level of risk that this could pose for a client consumer of the data may vary,
3081 the threat will not. The problem is exacerbated when reads must be performed across multiple resources
3082 in order to satisfy a client request: The window of opportunity for a write to slip in between distinct
3083 resource reads is much larger than the window between reads of individual pieces of data in a single
3084 resource.

3085 To resolve the threat of inconsistency, MCs should utilize a technique known as consistency checking.
3086 Before issuing a read, the MC should retrieve the ETag for the schema to be read, using the
3087 GetResourceETag command (clause 11.5). For a read that spans multiple resources, the global ETag
3088 should be read instead, by supplying 0xFFFFFFFF for the ResourceID in the command. The MC should
3089 then proceed with all of the reads and then check the ETag again. If the ETag matches what was initially
3090 read, the MC may conclude that the read was consistent and return it to the client. Otherwise, the MC
3091 should retry. It is expected that consistency failures will be very rare; however, if after three attempts, the
3092 MC cannot obtain a consistent read, it should report error 500, Internal Server Error to the client.

3093 NOTE For reads that only span a single resource, if the RDE Device asserts the **atomic_resource_read** bit in the
3094 **DeviceCapabilitiesFlags** response message to the NegotiateRedfishParameters command (clause 11.1), the MC
3095 may skip consistency checking.

3096 **14.2.6 [MC] Placement of RDE Device resources in the outward-facing Redfish** 3097 **URI hierarchy**

3098 In the Redfish Resource PDRs and Redfish Entity Association PDRs that an RDE Device presents, there
3099 will normally be one or a limited number that reflect EXTERNAL (0x0000) as their ContainingResourceID.
3100 These resources need to be integrated into the outward-facing Redfish URI hierarchy. Resources that do
3101 not reflect EXTERNAL as their ContainingResourceID do not need to be placed by the MC; it is the RDE
3102 Device's responsibility to make sure that they are accessible via some chain of Redfish Resource and
3103 Redfish Entity Association PDRs (including PDRs chained via @link properties) that ultimately link to
3104 EXTERNAL.

3105 When retrieving these PDRs for RDE Device components, the MC should read the
3106 ProposedContainingResourceName from the PDR. While following this recommendation is not
3107 mandatory, the MC should use it to inform a placement decision. If the MC does not follow the placement
3108 recommendation, it should read the MajorSchemaName field to identify the type of RDE Device they
3109 correspond to. Within the canon of standard Redfish schemas, there are comparatively few that reside at
3110 the top level, and each has a well-defined place it should appear within the hierarchy. The MC should

3111 thus make a simple map of which top-level schema types map to which places in the hierarchy and use
3112 this to place RDE Devices. In making these placement decisions, the MC should take information about
3113 the hardware platform topology into account so as to best reflect the overall Redfish system.

3114 It may happen that the MC encounters a schema it does not recognize. This can occur, for example, if a
3115 new schema type is standardized after the MC firmware is built. The handling of such cases is up to the
3116 MC. One possibility would be to place the schema in the OEM section under the most appropriate
3117 subobject. For an unknown DMTF standard schema, this should be the OEM/DMTF object. (To tell that a
3118 schema is DMTF standard, the MC may retrieve the published URI via GetSchemaURI command of
3119 clause 11.4, download the schema, and inspect the schema, namespace, or other content.)

3120 Naturally, wherever the MC places the RDE Device component, it shall add a link to the RDE Device
3121 component in the JSON retrieved by a client from the enclosing location.

3122 **14.2.7 LogEntry and LogEntryCollection resources**

3123 RDE Devices that support the LogEntry and LogEntryCollection resources must be aware that large
3124 volumes of LogEntries can overwhelm the 16 bit ResourceID space available for identifying Redfish
3125 Resource PDRs. To handle this case, it is recommended that RDE Devices provide a PDR for the
3126 LogEntryCollection but do NOT provide PDRs for the individual LogEntry instances. Instead, RDE
3127 Devices that support these schemas should also support the link expansion query parameter (see \$levels
3128 in [DSP0266](#) and the LinkExpand parameter from SupplyCustomRequestParameters in clause 12.2). This
3129 means that they should fill out the related resource links in the “Members” section of the response with
3130 bejResourceLinkExpansion data in which the encoded ResourceID is set to zero to ensure that the MC
3131 gets the COLLECTION_MEMBER_TYPE dictionary from the LogEntryCollection.

3132 **14.2.8 On-demand pagination**

3133 In Redfish, certain read operations may produce a very large amount of data. For example, reading a
3134 collection with many members will produce output with size proportional to the number of members.
3135 Rather than overload clients with a huge transfer of data, Redfish Devices may paginate it into chunks
3136 and provide one page at a time with an @odata.nextlink annotation giving a URI from which to retrieve
3137 the next piece.

3138 RDE supports the same pagination approach. It is entirely at the RDE Device’s discretion whether to
3139 paginate and where to draw pagination boundaries. When the RDE Device wishes to paginate, it shall
3140 insert an @odata.nextlink annotation, using a deferred binding pagination reference (see
3141 \$LINK.PDR<resource-ID>.PAGE<pagination-offset>% in clause 8.3), filling in the next page number for
3142 the data being returned. When the MC decodes this deferred binding, it shall create a temporary URI for
3143 the pagination and expose this pagination URI in the decoded JSON response it sends back to the client.
3144 Naturally, the encoded pagination URI must be decodable to extract the page number. Finally, when the
3145 client attempts a read from the pagination URI, the MC shall extract out the page number and send it to
3146 the RDE Device via the PaginationOffset field in the request message for the
3147 SupplyCustomRequestParameters command (clause 12.2).

3148 **14.2.9 Considerations for Redfish clients**

3149 No changes to behavior are required of Redfish clients in order to interact with BEJ-based RDE Devices;
3150 the details of providing them to the client are completely transparent from the client perspective. In fact, a
3151 fundamental design goal of this specification is that it should be impossible for a client to tell whether a
3152 Redfish message was ultimately serviced by an RDE Device that operates in JSON over HTTP/HTTPS or
3153 BEJ over PLDM.

3154

3155 14.2.10 OriginOfCondition in Redfish events

3156 The OriginOfCondition field in the Redfish event schema contains a link reference to a Redfish resource
3157 associated with a Redfish event. In typical use cases, resource data is read upon receiving the event to
3158 determine the resource state when the event transpired. This can happen either explicitly, from the client
3159 performing a read on the OriginOfCondition resource, or implicitly, if IncludeOriginOfCondition is set in the
3160 EventDestination when the client registered for Redfish events.

3161 RDE version 1.1 does not provide support for a device to populate the OriginOfCondition field with full
3162 resource data. However, an MC that wishes to minimize the timing window for the race condition may
3163 perform the appropriate read immediately upon receiving the Redfish event.

3164 14.2.11 [MC] Merging dictionaries with OEM extensions

3165 When merging dictionaries, MCs should consider that OEM extensions to Redfish schemas are
3166 enumerated alphabetically. In particular, the root objects (sets) of extensions (which come immediately
3167 under "OEM" inside the root object of the host schema) are likely to have conflicting sequence numbers if
3168 different sets of extensions appear in two different dictionaries for a given host schema.

3169 Additionally, no attempt has been made in this specification to make registry dictionaries able to be
3170 merged.

**ANNEX A
(normative)**

Change log

3171
3172
3173
3174

3175

Version	Date	Description
1.0.0	2019-06-25	
1.0.1	2019-12-09	Errata update
1.1.0	2021-02-11	<ul style="list-style-type: none"> • Added support for nested annotations • Enhanced message registry support, including a new registry dictionary, BEJ encoding • Improved ability to identify OEM extensions to schemas • Added support for PLDM common multipart transfers

3176