



1
2
3
4
5

Document Identifier: DSP0218

Date: 2019-06-25

Version: 1.0.0

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

8 **Supersedes: None**

9 **Document Class: Normative**

10 **Document Status: Published**

11 **Document Language: en-US**

12

13 Copyright Notice

14 Copyright © 2018, 2019 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29 withdrawn or modified after publication, and shall be indemnified and held harmless by any party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.

36

37

38

39

40

41

CONTENTS

43	Foreword	8
44	Introduction.....	9
45	Document conventions.....	9
46	1 Scope	10
47	2 Normative references	11
48	3 Terms and definitions	12
49	4 Symbols and abbreviated terms.....	15
50	5 Conventions	15
51	5.1 Reserved and unassigned values.....	15
52	5.2 Byte ordering.....	15
53	5.3 PLDM for Redfish Device Enablement data types	15
54	5.3.1 varstring PLDM data type	16
55	5.3.2 schemaClass PLDM data type	17
56	5.3.3 nnint PLDM data type	17
57	5.3.4 bejEncoding PLDM data type	17
58	5.3.5 bejTuple PLDM data type	18
59	5.3.6 bejTupleS PLDM data type.....	18
60	5.3.7 bejTupleF PLDM data type.....	18
61	5.3.8 bejTupleL PLDM data type	19
62	5.3.9 bejTupleV PLDM data type.....	20
63	5.3.10 bejNull PLDM data type	20
64	5.3.11 bejInteger PLDM data type	20
65	5.3.12 bejEnum PLDM data type.....	21
66	5.3.13 bejString PLDM data type.....	21
67	5.3.14 bejReal PLDM data type.....	21
68	5.3.15 bejBoolean PLDM data type	22
69	5.3.16 bejBytestring PLDM data type	22
70	5.3.17 bejSet PLDM data type.....	22
71	5.3.18 bejArray PLDM data type.....	23
72	5.3.19 bejChoice data PLDM type.....	23
73	5.3.20 bejPropertyAnnotation PLDM data type	23
74	5.3.21 bejResourceLink PLDM data type	24
75	5.3.22 bejResourceLinkExpansion PLDM data type	24
76	5.3.23 bejLocator PLDM data type	24
77	5.3.24 rdeOpID PLDM data type	25
78	6 PLDM for Redfish Device Enablement version.....	25
79	7 PLDM for Redfish Device Enablement Overview	25
80	7.1 Redfish Provider architecture overview	26
81	7.1.1 Roles	27
82	7.2 Redfish Device Enablement concepts	27
83	7.2.1 RDE Device discovery and registration	28
84	7.2.2 Data instances of Redfish schemas: Resources	29
85	7.2.3 Dictionaries	33
86	7.2.4 Redfish Operation support.....	38
87	7.2.5 PLDM RDE Events	48
88	7.2.6 Task support	50
89	7.3 Type code	51
90	7.4 Transport protocol type supported.....	51
91	7.5 Error completion codes.....	51
92	7.6 Timing specification	53
93	8 Binary Encoded JSON (BEJ)	54

94	8.1	BEJ design principles.....	54
95	8.2	SFLV tuples	55
96	8.2.1	Sequence number.....	55
97	8.2.2	Format.....	55
98	8.2.3	Length	55
99	8.2.4	Value.....	55
100	8.3	Deferred binding of data	56
101	8.4	BEJ encoding	58
102	8.4.1	Conversion of JSON data types to BEJ.....	58
103	8.4.2	Resource links	59
104	8.4.3	Annotations	59
105	8.4.4	Choice encoding for properties that support multiple data types	60
106	8.4.5	Properties with invalid values	60
107	8.4.6	Properties missing from dictionaries.....	61
108	8.5	BEJ decoding.....	61
109	8.5.1	Conversion of BEJ data types to JSON.....	61
110	8.5.2	Annotations	62
111	8.5.3	Sequence numbers missing from dictionaries.....	63
112	8.5.4	Sequence numbers for read-only properties in modification Operations	63
113	8.6	Example encoding and decoding.....	63
114	8.6.1	Example dictionary.....	63
115	8.6.2	Example encoding	66
116	8.6.3	Example decoding	69
117	8.7	BEJ locators.....	72
118	9	Operational behaviors	72
119	9.1	Initialization (MC perspective).....	72
120	9.1.1	Sample initialization ladder diagram	72
121	9.1.2	Initialization workflow diagram	74
122	9.2	Operation/Task lifecycle.....	76
123	9.2.1	Example Operation command sequence diagrams.....	76
124	9.2.2	Operation/Task overview workflow diagrams (Operation perspective)	80
125	9.2.3	RDE Operation state machine (RDE Device perspective)	88
126	9.3	Event lifecycle	99
127	10	PLDM commands for Redfish Device Enablement.....	102
128	11	PLDM for Redfish Device Enablement – Discovery and schema commands	103
129	11.1	NegotiateRedfishParameters command format	103
130	11.2	NegotiateMediumParameters command format.....	105
131	11.3	GetSchemaDictionary command format.....	106
132	11.4	GetSchemaURI command format.....	107
133	11.5	GetResourceETag command format.....	108
134	12	PLDM for Redfish Device Enablement – RDE Operation and Task commands	109
135	12.1	RDEOperationInit command format.....	109
136	12.2	SupplyCustomRequestParameters command format	112
137	12.3	RetrieveCustomResponseParameters command format	116
138	12.4	RDEOperationComplete command format.....	117
139	12.5	RDEOperationStatus command format	118
140	12.6	RDEOperationKill command format.....	120
141	12.7	RDEOperationEnumerate command format.....	121
142	13	PLDM for Redfish Device Enablement – Utility commands	122
143	13.1	MultipartSend command format.....	122
144	13.2	MultipartReceive command format	124
145	14	Additional Information.....	126
146	14.1	Multipart transfers	126

147 14.1.1 Flag usage for MultipartSend..... 126
 148 14.1.2 Flag usage for MultipartReceive 127
 149 14.1.3 Multipart transfer examples 127
 150 14.2 Implementation notes..... 129
 151 14.2.1 Schema updates 129
 152 14.2.2 Storage of dictionaries 129
 153 14.2.3 Dictionaries for related schemas 129
 154 14.2.4 [MC] HTTP/HTTPS POST Operations..... 130
 155 14.2.5 Consistency checking of read Operations 131
 156 14.2.6 [MC] Placement of RDE Device resources in the outward-facing Redfish URI
 157 hierarchy 131
 158 14.2.7 LogEntry and LogEntryCollection resources 132
 159 14.2.8 On-demand pagination 132
 160 14.2.9 Considerations for Redfish clients 132
 161 ANNEX A (informative) Change log..... 133
 162

163 Figures

164	Figure 1 – RDE Roles	27
165	Figure 2 – Example linking of Redfish Resource and Redfish Entity Association PDRs	32
166	Figure 3 – Schema linking without Redfish entity association PDRs	32
167	Figure 4 – Dictionary binary format.....	37
168	Figure 5 – DummySimple schema.....	65
169	Figure 6 – DummySimple dictionary – binary form.....	66
170	Figure 7 – Example Initialization ladder diagram	74
171	Figure 8 – Typical RDE Device discovery and registration.....	75
172	Figure 9 – Simple read Operation ladder diagram.....	76
173	Figure 10 – Complex Read Operation ladder diagram	78
174	Figure 11 – Write Operation ladder diagram.....	79
175	Figure 12 – Write Operation with long-running Task ladder diagram	80
176	Figure 13 – RDE Operation lifecycle overview (holistic perspective)	84
177	Figure 14 – RDE Task lifecycle overview (holistic perspective)	87
178	Figure 15 – Operation lifecycle state machine (RDE Device perspective)	99
179	Figure 16 – Redfish event lifecycle overview.....	101
180	Figure 17 – MultipartSend example	128
181	Figure 18 – MultipartReceive example	129
182		

183 Tables

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

204 Table 21 – bejArray value encoding for BEJ..... 23

205 Table 22 – bejChoice value encoding for BEJ..... 23

206 Table 23 – bejPropertyAnnotation value encoding for BEJ 23

207 Table 24 – bejPropertyAnnotation value encoding example 24

208 Table 25 – bejResourceLink value encoding for BEJ 24

209 Table 26 – bejResourceLinkExpansion value encoding for BEJ 24

210 Table 27 – bejLocator value encoding 25

211 Table 28 – rdeOpID data structure 25

212 Table 29 – Redfish dictionary binary format 34

213 Table 30 – Dictionary entry example for a property supporting multiple formats 37

214 Table 31 – Redfish operations 39

215 Table 32 – Redfish operation headers..... 41

216 Table 33 – Redfish operation request query options 46

217 Table 34 – PLDM for Redfish Device Enablement completion codes 51

218 Table 35 – HTTP codes for standard PLDM completion codes..... 53

219 Table 36 – Timing specification..... 53

220 Table 37 – Sequence number dictionary indication 55

221 Table 38 – JSON data types supported in BEJ 56

222 Table 39 – BEJ deferred binding substitution parameters 57

223 Table 40 – Message annotation related property BEJ locator encoding 60

224 Table 41 – DummySimple dictionary (tabular form) 65

225 Table 42 – Initialization Workflow 74

226 Table 43 – Operation lifecycle overview 81

227 Table 44 – Task lifecycle overview 85

228 Table 45 – Task lifecycle state machine 89

229 Table 46 – Event lifecycle overview 99

230 Table 47 – PLDM for Redfish Device Enablement command codes..... 102

231 Table 48 – NegotiateRedfishParameters command format..... 104

232 Table 49 – NegotiateMediumParameters command format 106

233 Table 50 – GetSchemaDictionary command format..... 106

234 Table 51 – GetSchemaURI command format..... 107

235 Table 52 – GetResourceETag command format 108

236 Table 53 – RDEOperationInit command format..... 109

237 Table 54 – SupplyCustomRequestParameters command format 113

238 Table 55 – RetrieveCustomResponseParameters command format 116

239 Table 56 – RDEOperationComplete command format 117

240 Table 57 – RDEOperationStatus command format 118

241 Table 58 – RDEOperationKill command format..... 121

242 Table 59 – RDEOperationEnumerate command format..... 122

243 Table 60 – MultipartSend command format..... 123

244 Table 61 – MultipartReceive command format 125

245

246

Foreword

247 The *Redfish Device Enablement Specification* (DSP0218) was prepared by the Platform Management
248 Components Intercommunications (PMCI Working Group) of the DMTF.

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

251 **Acknowledgments**

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

253 **Editor:**

- 254 • Bill Scherer – Hewlett Packard Enterprise

255 **Contributors:**

- 256 • Richelle Ahlvers – Broadcom Inc.
- 257 • Jeff Autor – Hewlett Packard Enterprise
- 258 • Patrick Caporale – Lenovo
- 259 • Mike Garrett – Hewlett Packard Enterprise
- 260 • Jeff Hilland – Hewlett Packard Enterprise
- 261 • Yuval Itkin – Mellanox Technologies
- 262 • Ira Kalman – Intel
- 263 • Eliel Louzoun – Intel
- 264 • Balaji Natrajan – Microchip Technology Inc.
- 265 • Edward Newman – Hewlett Packard Enterprise
- 266 • Zvika Perry Peleg – Cavium
- 267 • Scott Phuong, Cisco Systems, Inc.
- 268 • Jeffrey Plank – Microchip Technology Inc.
- 269 • Joey Rainville – Hewlett Packard Enterprise
- 270 • Patrick Schoeller – Hewlett Packard Enterprise
- 271 • Hemal Shah – Broadcom Inc.
- 272 • Bob Stevens – Dell
- 273 • Bill Vetter – Lenovo

274

Introduction

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

286 Document conventions

287 Clause naming conventions

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

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

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

298 Typographical conventions

299 This document uses the following typographical conventions:

- 300 • 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.

340 2 Normative references

341 The following referenced documents are indispensable for the application of this document. For dated or
342 versioned references, only the edition cited (including any corrigenda or DMTF update versions) applies.
343 For references without a date or version, the latest published edition of the referenced document
344 (including any corrigenda or DMTF update versions) applies. Earlier versions may not provide sufficient
345 support for this specification.

346 DMTF DSP0222, *Network Controller Sideband Interface (NC-SI) Specification 1.1*,
347 https://www.dmtf.org/sites/default/files/standards/documents/DSP0222_1.1.pdf

348 DMTF DSP0236, *MCTP Base Specification 1.2*,
349 http://dmtf.org/sites/default/files/standards/documents/DSP0236_1.2.pdf

350 DMTF DSP0240, *Platform Level Data Model (PLDM) Base Specification 1.0*,
351 http://dmtf.org/sites/default/files/standards/documents/DSP0240_1.0.pdf

352 DMTF DSP0241, *Platform Level Data Model (PLDM) Over MCTP Binding Specification 1.0*,
353 http://dmtf.org/sites/default/files/standards/documents/DSP0241_1.0.pdf

354 DMTF DSP0245, *Platform Level Data Model (PLDM) IDs and Codes Specification 1.3*,
355 http://dmtf.org/sites/default/files/standards/documents/DSP0245_1.3.pdf

356 DMTF DSP0248, *Platform Level Data Model (PLDM) for Platform Monitoring and Control Specification*
357 *1.1*, http://dmtf.org/sites/default/files/standards/documents/DSP0248_1.1.pdf

358 DMTF DSP0266, *Redfish Scalable Platforms Management API Specification 1.6*,
359 http://www.dmtf.org/sites/default/files/standards/documents/DSP0266_1.6.pdf

360 DMTF DSP0267, *PLDM for Firmware Update Specification 1.0*,
361 https://www.dmtf.org/sites/default/files/standards/documents/DSP0267_1.0.pdf

362 DMTF DSP4004, *DMTF Release Process 2.4*,
363 http://dmtf.org/sites/default/files/standards/documents/DSP4004_2.4.pdf

364 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)
365 [international.org/publications/files/ECMA-ST/ECMA-404.pdf](http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf)

366 IETF RFC2781, *UTF-16, an encoding of ISO 10646*, February 2000,
367 <http://www.ietf.org/rfc/rfc2781.txt>

368 IETF STD63, *UTF-8, a transformation format of ISO 10646* <http://www.ietf.org/rfc/std/std63.txt>

369 IETF RFC4122, *A Universally Unique Identifier (UUID) URN Namespace*, July 2005,
370 <http://www.ietf.org/rfc/rfc4122.txt>

371 IETF RFC4646, *Tags for Identifying Languages*, September 2006,
372 <http://www.ietf.org/rfc/rfc4646.txt>

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

386 **3 Terms and definitions**

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

389 The terms "shall" ("required"), "shall not", "should" ("recommended"), "should not" ("not recommended"),
390 "may", "need not" ("not required"), "can" and "cannot" in this document are to be interpreted as described
391 in ISO/IEC Directives, Part 2, Clause 7. The terms in parentheses are alternatives for the preceding term,
392 for use in exceptional cases when the preceding term cannot be used for linguistic reasons. Note that
393 ISO/IEC Directives, Part 2, Clause 7 specifies additional alternatives. Occurrences of such additional
394 alternatives shall be interpreted in their normal English meaning.

395 The terms "clause", "subclause", "paragraph", and "annex" in this document are to be interpreted as
396 described in ISO/IEC Directives, Part 2, Clause 6.

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

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

404 **3.1**

405 **Action**

406 Any standard Redfish action defined in a standard Redfish Schema or any custom OEM action defined in
407 an OEM schema extension

408 **3.2**

409 **Annotation**

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

- 412 **3.3**
413 **Client**
414 Any agent that communicates with a management controller to enable a user to manage Redfish-
415 compliant systems and RDE Devices
- 416 **3.4**
417 **Collection**
418 A Redfish container holding an array of independent Redfish resource Members that in turn are typically
419 represented by a schema external to the one that contains the collection itself.
- 420 **3.5**
421 **Device Component**
422 A top-level entry point into the schema hierarchy presented by an RDE Device
- 423 **3.6**
424 **Dictionary**
425 A binary lookup table containing translation information that allows conversion between BEJ and JSON
426 formats of data for a given resource
- 427 **3.7**
428 **Discovery**
429 The process by which an MC determines that an RDE Device supports PLDM for Redfish Device
430 Enablement
- 431 **3.8**
432 **Major Schema**
433 The primary schema defining the format of a collection of data, usually a published standard Redfish
434 schema.
- 435 **3.9**
436 **Member**
437 Any of the independent resources contained within a collection
- 438 **3.10**
439 **Metadata**
440 Information that describes data of interest, such as its type format, length in bytes, or encoding method
- 441 **3.11**
442 **OData**
443 The [Open Data protocol](#), a source of annotations in Redfish, as defined by OASIS.
- 444 **3.12**
445 **OEM Extension**
446 Any manufacturer-specific addition to major schema
- 447 **3.13**
448 **Property**
449 An individual datum contained within a Resource

- 450 **3.14**
451 **RDE Device**
452 Any PLDM terminus containing an RDE Provider that requires the intervention of an MC to receive
453 Redfish communications
- 454 **3.15**
455 **RDE Provider**
456 Any RDE Device that responds to RDE Operations. See also **Redfish Provider**.
- 457 **3.16**
458 **RDE Operation**
459 The sequence of PLDM messages and operations that represent a Redfish Operation being executed by
460 an MC and/or an RDE Device on behalf of a client. See also **Redfish Operation**.
- 461 **3.17**
462 **Redfish Operation**
463 Any Redfish operation transmitted via HTTP or HTTPS from a client to an MC for execution. See also
464 **RDE Operation**.
- 465 **3.18**
466 **Redfish Provider**
467 Any entity that responds to Redfish Operations. See also **RDE Provider**.
- 468 **3.19**
469 **Registration**
470 The process of enabling a compliant RDE Device with an MC to be an RDE Provider
- 471 **3.20**
472 **Resource**
473 A hierarchical set of data organized in the format specified in a Redfish Schema.
- 474 **3.21**
475 **Schema**
476 Any regular structure for organizing one or more fields of data in a hierarchical format
- 477 **3.22**
478 **Task**
479 Any Operation for which an RDE Device cannot complete execution in the time allotted to respond to the
480 PLDM triggering command message sent from the MC and for which the MC creates standard Redfish
481 Task and TaskMonitor objects
- 482 **3.23**
483 **Triggering Command**
484 The PLDM command that supplies the last bit of data needed for an RDE Device to begin execution of an
485 RDE Operation
- 486 **3.24**
487 **Truncated**
488 When applied to a dictionary, one that is limited to containing conversion information for properties
489 supported by an RDE Device

490 **4 Symbols and abbreviated terms**

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

493 **4.1**

494 **BEJ**

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

496 **4.2**

497 **JSON**

498 JavaScript Object Notation

499 **4.3**

500 **RDE**

501 Redfish Device Enablement

502 **5 Conventions**

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

505 **5.1 Reserved and unassigned values**

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

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

510 **5.2 Byte ordering**

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

514 **5.3 PLDM for Redfish Device Enablement data types**

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

517 **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

Data Type	Interpretation
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
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
bejResourceLink	Resource Link data encoded for BEJ per clause 5.3.21
bejResourceLinkExpansion	Resource Link data expanded to include schema data encoded for BEJ per clause 5.3.22
bejLocator	An intra-schema locator for Operation targeting; formatted per clause 5.3.23
rdeOpID	An Operation identifier used to link together the various command messages that comprise a single RDE Operation; formatted per clause 5.3.24

518 5.3.1 varstring PLDM data type

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

521 **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

522 **5.3.2 schemaClass PLDM data type**

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

- 525 • MAJOR: the main schema containing the data for a Redfish resource. This class covers the
526 vast majority of schemas for Redfish resources.
- 527 • EVENT: the standard DMTF-published event schema, for occurrences that clients may wish to
528 be notified about.
- 529 • ANNOTATION: the standard DMTF-published annotation schema that captures metadata about
530 a major schema or payload.
- 531 • ERROR: the standard DMTF-published error schema that documents an extended error when a
532 Redfish operation cannot be completed.
- 533 • COLLECTION_MEMBER_TYPE: for resources that correspond to Redfish collections, this
534 class enables access to the major schema for members of that collection from the context of the
535 collection resource. (Unlike regular resources, collections in Redfish are unversioned and
536 contain multiple members.)

537 **Table 3 – schemaClass enumeration**

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

538 **5.3.3 nnint PLDM data type**

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

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

544 **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)

545 **5.3.4 bejEncoding PLDM data type**

546 The bejEncoding PLDM data type captures an overall hierarchical BEJ-encoded block of hierarchical
547 data.

548

Table 5 – bejEncoding data structure

Type	Description
ver32	BEJ Version; shall be 1.0.0 (0xF1F0F000) for this specification
uint16	Reserved for BEJ flags
schemaClass	Defines the primary schema type for the data encoded in bejTuple below. Shall not be ANNOTATION
bejTuple	The encoded tuple data, defined in clause 5.3.5

549 **5.3.5 bejTuple PLDM data type**

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

551

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

552 **5.3.6 bejTupleS PLDM data type**553 The bejTupleS PLDM data type captures the Sequence Number BEJ tuple element described in clause
554 8.2.1

555

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.

556 **5.3.7 bejTupleF PLDM data type**

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

558

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 flag **. 1b indicates the flag is set</p> <p>[0] - deferred_binding flag*. 1b indicates the flag is set</p>

559 * The deferred_binding flag shall only be set in conjunction with BEJ String data and shall never be set
 560 when encoding the format of a property inside a dictionary. See clause 8.3.

561 ** The read_only_property flag shall only be set when encoding the format of a property inside a
 562 dictionary. See clause 7.2.3.2.

563 *** The nullable_property flag shall only be set when encoding the format of a property inside a dictionary.
 564 See clause 7.2.3.2.

565

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 – 1101b	Reserved	n/a
1110b	BEJ Resource Link	bejResourceLink
1111b	BEJ Resource Link Expansion	bejResourceLinkExpansion

566 **5.3.8 bejTupleL PLDM data type**

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

568

Table 10 – bejTupleL encoding for BEJ

Type	Description
nnint	Length in bytes of value tuple field

569 **5.3.9 bejTupleV PLDM data type**

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

571 **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.

572 **5.3.10 bejNull PLDM data type**

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

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

Type	Description
(none)	No fields

575 **5.3.11 bejInteger PLDM data type**

576 Integer data shall be encoded as the shortest sequence of bytes (little endian) that represent the value in
577 twos complement encoding. This implies that if the value is positive and the high bit (0x80) of the MSB in
578 an unsigned representation would be set, the unsigned value will be prefixed with a new null (0x00) MSB
579 to mark the value as explicitly positive.

580 **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)

581 **5.3.12 bejEnum PLDM data type**

582 **Table 14 – bejEnum value encoding for BEJ**

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

583 **5.3.13 bejString PLDM data type**

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

585 **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

586 **5.3.14 bejReal PLDM data type**

587 BEJ encoding for *whole*, *fract*, and *exp* that represent the base 10 encoding $whole.fract \times 10^{exp}$.

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

590 **Table 16 – 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)

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

595 As an example, Table 17 shows the encoding of the JSON number “1.0005e+10”:

596 **Table 17 – 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)

597 5.3.15 bejBoolean PLDM data type

598 The bejBoolean PLDM data type captures boolean data.

599 **Table 18 – bejBoolean value encoding for BEJ**

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

600 5.3.16 bejBytestring PLDM data type

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

603 **Table 19 – bejBytestring value encoding for BEJ**

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

604 5.3.17 bejSet PLDM data type

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

607 **Table 20 – 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)

608 **5.3.18 bejArray PLDM data type**

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

611 **Table 21 – 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)

612 **5.3.19 bejChoice data PLDM type**

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

616 **Table 22 – bejChoice value encoding for BEJ**

Type	Description
bejTuple	Selected option

617 **5.3.20 bejPropertyAnnotation PLDM data type**

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

622 **Table 23 – 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
bejTupleF	Format for annotation data applying to the property indicated by the sequence number above. 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
bejTupleV	Annotation data applying to the property indicated by the sequence number above

623 As an example, Table 24 shows the encoding of the annotation:

624 "Status@Redfish.RequiredOnCreate" : false

625

Table 24 – bejPropertyAnnotation value encoding example

Type	Bytes	Description
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

626

5.3.21 bejResourceLink PLDM data type

627

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

629

630

Table 25 – bejResourceLink value encoding for BEJ

Type	Description
nnint	ResourceID of Redfish Resource PDR for linked schema

631

5.3.22 bejResourceLinkExpansion PLDM data type

632

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

634

635

636

Table 26 – bejResourceLinkExpansion value encoding for BEJ

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

637

5.3.23 bejLocator PLDM data type

638

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

639

640

641

642

643

644

Table 27 – 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]
...	...
bejTupleS	Sequence number [N - 1]

645 **5.3.24 rdeOpID PLDM data type**

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

648

Table 28 – rdeOpID data structure

Type	Description
uint16	OperationIdentifier Numeric identifier for the Operation. Operation identifiers with the MSB set (1b) are reserved for use by the MC when it instantiates Operations. Operation identifiers with the MSB 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.

649 **6 PLDM for Redfish Device Enablement version**

650 The version of this Platform Level Data Model (PLDM) for Redfish Device Enablement Specification shall
651 be 1.0.0 (major version number 1, minor version number 0, update version number 0, and no alpha
652 version).

653 In response to the GetPLDMVersion command described in [DSP0240](#), the reported version for Type 6
654 (PLDM for Redfish Device Enablement, this specification) shall be encoded as 0xF1F0F000.

655 **7 PLDM for Redfish Device Enablement Overview**

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

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

666 Implementing the Redfish protocol as defined by [DSP0266](#) is a big challenge when passing requests to
667 and from devices such as network adapters that have highly limited processing capabilities and memory

668 space. Redfish's messages are prohibitively large because they are encoded for human readability in
669 HTTP/HTTPS using JavaScript Object Notation (JSON). This specification details a compressed
670 encoding of Redfish payloads that is suitable for such devices. It further identifies a common method to
671 use PLDM to communicate these messages between a management controller and the devices that host
672 the data the operations target. The functionality of providing a complete Redfish service is distributed
673 across components that function in different roles; this is discussed in more detail in clause 7.1.1.

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

- 681 • Binary Encoded JSON (BEJ)
 - 682 ○ Simplified compact binary format for communicating Redfish JSON data payloads
 - 683 ○ Captures essential schema information into a compact binary dictionary so that it does
 - 684 not need to be transferred as part of message payloads
 - 685 ○ Defined locators allow for selection of a specific object or property inside the schema's
 - 686 data hierarchy to perform an operation
 - 687 ○ Encoders and decoders account for the unordered nature of BEJ and JSON properties
 - 688
- 689 • RDE Device Registration for Redfish
 - 690 ○ A mechanism to determine the schemas the RDE Device supports, including OEM
 - 691 custom extensions
 - 692 ○ A mechanism to determine parameters for limitations on the types of communication the
 - 693 RDE Device can perform, the number of outstanding operations it can support, and other
 - 694 management parameters
 - 695
- 696 • Messaging Support for Redfish Operations via BEJ
 - 697 ○ Read, Update, Post, Create, Delete Operations
 - 698 ○ Asynchrony support for Operations that spawn long-running Tasks
 - 699 ○ Notification Events for completion of long-running Tasks and for other RDE Device-
 - 700 specific happenings¹
 - 701 ○ Advanced operations such as pagination and ETag support

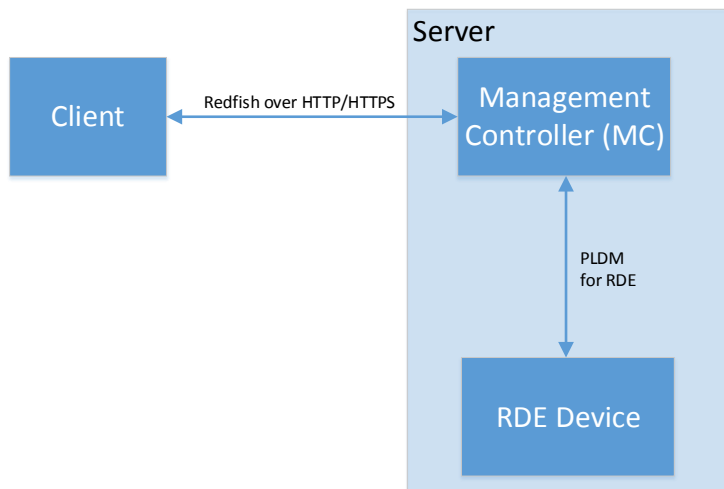
702 7.1 Redfish Provider architecture overview

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

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

710 **7.1.1 Roles**

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



712

713

Figure 1 – RDE Roles

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

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

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

731 **7.2 Redfish Device Enablement concepts**

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

733 7.2.1 RDE Device discovery and registration

734 The processes by which an RDE Device becomes known to the MC and thus visible to clients are known
735 as Discovery and Registration. Discovery consists of the MC becoming aware of an RDE Device and
736 recognizing that it supports Redfish management. Registration consists of the MC interrogating specific
737 details of the RDE Device's Redfish capabilities and then making it visible to external clients. An example
738 ladder diagram and a typical workflow for the discovery and registration process may be found in clause
739 9.1.

740 7.2.1.1 RDE Device discovery

741 The first step of the discovery process begins when the MC detects the presence of a PLDM capable
742 device on a particular medium. The technique by which the MC determines that a device supports PLDM
743 is outside the scope of this specification; details of this process may be found in the PLDM base
744 specification ([DSP0240](#)). Similarly, the technique by which the MC may determine that a device found on
745 one medium is the same device it has previously found on another medium is outside the scope of this
746 specification.

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

753 Next, the MC may use the base PLDM GetPLDMCommands command once for each of the Monitoring
754 and Control and Redfish Device Enablement PLDM Types to verify that the RDE Device supports the
755 required commands. The required commands for each PLDM Type are listed in Table 47. As with the
756 GetPLDMTypes command, use of this command is optional if the MC has some other technique to
757 understand which commands the RDE Device supports. At this point, RDE Device discovery at the PLDM
758 level is complete.

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

- 763 • The RDE Device's RDE Provider name
- 764 • The RDE Device's support for concurrency. This is the number of Operations the RDE Device
765 can support simultaneously
- 766 • RDE feature support

767 The final step in discovery is for the MC to invoke the NegotiateMediumParameters command (clause
768 11.2) in order to negotiate communication details for the RDE Device. The MC invokes this command on
769 each medium it plans to communicate with the RDE Device on as it discovers the RDE Device on that
770 medium. Medium details include the following:

- 771 • The size of data that can be sent in a single message on the medium

772 7.2.1.2 RDE Device registration

773 In the registration process, the MC interrogates the RDE Device about the hierarchy of Redfish resources
774 it supports in order to act as a proxy, transparently mirroring them to external clients. The MC may skip
775 registration of the RDE Device if the PDR/Dictionary signature retrieved via the

776 NegotiateRedfishParameters command matches one previously retrieved and the MC still has the PDRs
777 and dictionaries cached.

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

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

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

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

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

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

816 **7.2.2 Data instances of Redfish schemas: Resources**

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

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

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

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

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

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

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

- 845 1) The resource is contained within the RDE Device. If it is not, it does not need to be part of the
846 RDE Device's management topology model.
- 847 2) The resource is subordinate to another resource contained within the RDE Device. If it is not,
848 the resource can be linked directly to the resource outside the RDE Device by setting its parent
849 field to EXTERNAL.
- 850 3) The resource needs to be linked to another resource outside the RDE Device.

851 7.2.2.1 Alignment of resources

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

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

861 7.2.2.2 Example linking of PDRs within RDE Devices

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

864 The example in Figure 2 models a simple rack-mounted server with local RAID storage. In this example,
865 we see a Redfish Resource PDR offering an instance of the standard Redfish Storage resource, with

866 ResourceID 123. This PDR has ContainingResourceID (abbreviated ContainingRID in the figure) set to
867 EXTERNAL as the RDE Device should be subordinate to the Storage Collection under ComputerSystem.

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

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

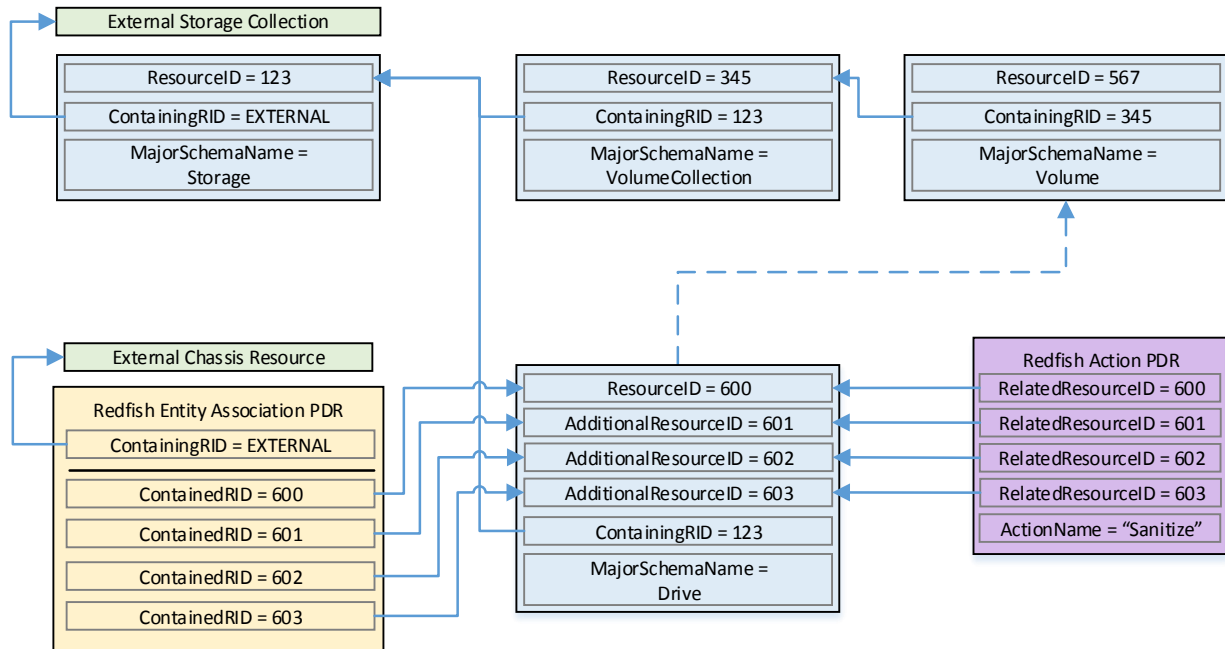
875 The connections discussed so far are all direct parent linkages in the Redfish Resource PDRs because
876 the links they represent are the direct subordinate resource links from the standard Redfish storage
877 model. However, the Redfish storage model also includes notations that drives are related to (contained
878 within) a volume and that drives are related to (present inside) a chassis. These resource relations can be
879 modeled using Redfish Entity Association PDRs if the MC is managing the links. Alternatively, they can
880 be implicitly managed by the RDE Device. In this case, the RDE Device will expose the links itself by
881 filling in a Links section of the relevant resource data with references to the linked resources. While the
882 RDE Device could in theory provide a Redfish Entity Association PDR for this case, it serves no purpose
883 for the MC.

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

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

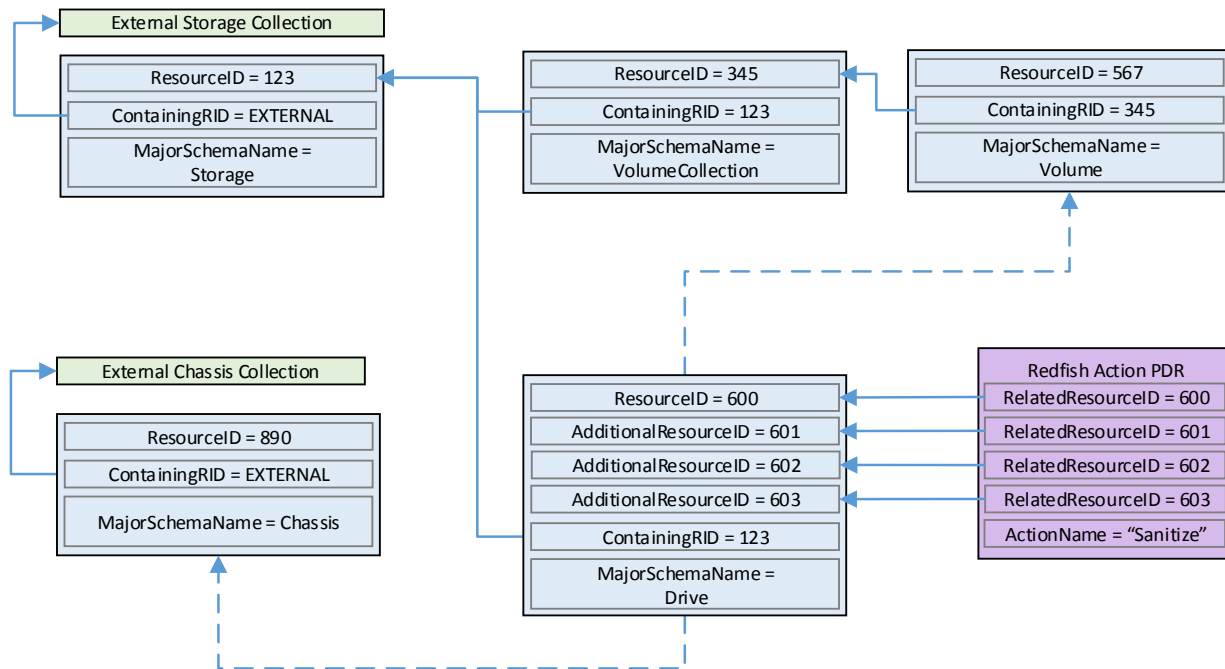
894 Finally, each of the drives supports a Sanitize operation. This is shown by instantiating a Redfish Action
895 PDR naming the Sanitize action and linking it to each of the drives.

896 As an alternative to the PDR layout of Figure 2, in Figure 3, the RDE Device exposes its own chassis
897 resource (labeled as Resource ID 890) rather than having the drives be part of an external chassis. The
898 PDR for this chassis resource shows ContainingResourceID EXTERNAL to demonstrate that it belongs in
899 the system chassis collection resource. With this modification, the links between the chassis resource and
900 the drives can be managed internally by the RDE Device and hence no Redfish Entity Association PDR is
901 necessary.



902
903

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



905
906

907 **Figure 3 – Schema linking without Redfish entity association PDRs**

908 7.2.3 Dictionaries

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

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

- 917 • Standard Redfish data schema (aka the major schema)
- 918 • Standard Redfish Event schema
- 919 • Standard Redfish Annotation schema
- 920 • Standard Redfish Error schema

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

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

924 Dictionaries for standard Redfish schemas are published on the DMTF Redfish website at
925 <http://redfish.dmtf.org/dictionaries>. Naturally, these dictionaries do not include OEM extensions. RDE
926 Devices may support their resources with either the standard dictionaries or with custom dictionaries that
927 may include OEM extensions, and that may also be truncated to contain only entries for properties
928 supported by the RDE Device.

929 7.2.3.1 Canonizing a schema into a dictionary

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

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

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.

- 949 4) In the event that a property is deleted from a schema, its sequence number shall not be reused;
950 the sequence number for the deleted property shall forever remain allocated to that property.
- 951 5) As with properties, the values of an enumeration shall collectively receive an independent set of
952 sequence numbers ranging from zero to $N - 1$, where N is the number of enumeration values.
953 Sequence numbers for enumeration values not belonging to the same enumeration are not
954 related in any way.
- 955 6) For the initial version of a Redfish schema, sequence numbers for enumeration values shall be
956 assigned according to a strict alphabetical ordering of the enumeration values from the schema.
- 957 7) In order to preserve backwards compatibility with earlier versions of schemas, for subsequent
958 revisions of Redfish schemas, the sequence numbers for enumeration values added in that
959 revision shall be assigned sequence numbers N to $N + A - 1$, where N is the number of
960 sequence numbers assigned in the previous revision and A is the number of enumeration
961 values added in the present revision. The new sequence numbers shall be assigned according
962 to a strict alphabetical ordering of their value strings from the schema.
- 963 8) In the event that an enumeration value is deleted from a schema, its sequence number shall not
964 be reused; the sequence number for the deleted enumeration value shall forever remain
965 allocated to that enumeration value.

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

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

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

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

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

985 7.2.3.2 Dictionary binary format

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

987 **Table 29 – Redfish dictionary binary format**

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

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

Type	Dictionary Data
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.

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

991

DWORD	Byte offset			
	+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

DWORD	Byte offset			
	+0	+1	+2	+3
...	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}			

992 **Figure 4 – Dictionary binary format**

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

995 **7.2.3.2.1 Hierarchical organization of entries**

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

1000 NOTE While not mandatory, it is acceptable that multiple dictionary entries may point to a common complex
 1001 subtype to allow reuse of that information and reduce the overall size of the dictionary. For example,
 1002 Resource.status is commonly used multiple times within the same schema, so having a single offset for it
 1003 can trim some length from the dictionary.

1004 **7.2.3.3 Properties that support multiple formats**

1005 For properties that support multiple formats, the dictionary shall contain an entry linking the property
 1006 name string to the BEJ Choice format. This choice entry shall in turn link to a series of anonymous child
 1007 entries (name offset = 0x0000) that are of the various data formats supported by the property. For
 1008 example, if a TCP/IP hostname property supports both string (“www.dmtf.org”) and numeric (the 32-bit
 1009 equivalent of 72.47.235.184) values, the dictionary might contain rows such as the following:

1010 **Table 30 – Dictionary entry example for a property supporting multiple formats**

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

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

1011 NOTE Following the rules for sequence number assignment (see clause 7.2.3.1), each cluster of properties
 1012 contained within a given set and each cluster of enumeration values are numbered separately. Hence
 1013 sequence numbers may be repeated within a dictionary.

1014 An exception to this rule is that properties that support null and exactly one other data format shall be
 1015 collapsed into a single entry in the dictionary listing only the non-null data format. The nullable_property
 1016 bit in the bejTupleF value of the format entry in the dictionary shall be set to 1b in this case. This case is
 1017 common in the standard Redfish schemas, where most properties are nullable. This is flagged with the
 1018 “nullable” keyword in the CSDL schemas, but in the JSON schemas, it manifests as the supported type
 1019 list for the property consisting of NULL and either a solitary second type or a collection of strings that form
 1020 an enumeration.

1021 7.2.3.4 Annotation dictionary format

1022 Standard Redfish annotations are derived from three sources: the Redfish, odata, and message
 1023 schemas. The annotations that can be part of a JSON payload are collected together into the redfish-
 1024 payload-annotations.vX.Y.Z.json schema file. This clause details special notes that apply to building the
 1025 annotation dictionary:

- 1026 • The dictionary entries for properties in the annotation dictionary shall include the entire name of
 1027 the annotation, beginning with the ‘@’ sign and including both the annotation source (one of
 1028 redfish, message, or odata) and the annotation’s name itself. For example, the dictionary Name
 1029 field for the @odata.id property shall be an offset to the string “@odata.id”.
- 1030 • The dictionary entries for patternProperties in the annotation dictionary shall be stripped of the
 1031 wildcard patterns before the ‘@’ sign and of the trailing ‘\$’ sign but shall otherwise be treated
 1032 identically to standard properties. For example, the dictionary Name field for the “^[a-zA-Z_][a-
 1033 zA-Z0-9_]*”?@Message.ExtendedInfo\$” patternProperty shall be an offset to the string
 1034 “@Message.ExtendedInfo”.
- 1035 • In accordance with the rules presented in clause 7.2.3, the top-level entries for annotations
 1036 (those containing the names of the annotations themselves) shall be sorted alphabetically
 1037 together for the initial version of the schema’s dictionary, and shall be appended to the list with
 1038 each schema revision. Stated explicitly, the annotations from the properties and
 1039 patternProperties shall be comingled together within the entries for each revision of the
 1040 dictionary.
- 1041 • Dictionary entries for children properties of annotations, such as the anonymous string value
 1042 array entries for @Redfish.AllowableValues shall be structured and formatted per the rules
 1043 presented in clause 7.2.3.

1044 7.2.4 Redfish Operation support

1045 Redfish Operations are sent from a client to a Redfish Provider that is able to process them and respond
 1046 appropriately. These operations are encoded in JSON and transported via either the HTTP or the HTTPS
 1047 protocol.

1048 In this specification, the MC is the Redfish Provider that the client sends operations to. However, rather
 1049 than responding directly, the MC is a proxy that conveys these operations to the RDE Devices that
 1050 maintain the data and can provide responses to client requests. The proxied operations (that are
 1051 transmitted to the RDE Device as RDE Operations) are encoded in BEJ (clause 8) and transported via
 1052 PLDM. The MC, in its role as proxy Redfish Provider for the RDE Devices, translates the JSON/HTTP(S)
 1053 requests from the client into BEJ/PLDM for the RDE Device, and then translates the BEJ/PLDM response
 1054 from the RDE Device into a JSON/HTTP(S) response for the client.

1055 **7.2.4.1 Primary Operations**

1056 There are seven primary Redfish Operations. These are summarized in Table 31.

1057 **Table 31 – 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 either 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

1058 The only Redfish Operation that is required to be supported in RDE is Read; however, it is expected that
 1059 implementations will support Update as well. Create and Delete are conditionally required for RDE
 1060 Devices that contain collections; Action is conditionally required for RDE Devices that support Redfish
 1061 schema-defined actions. The Head and Replace Redfish Operations are strictly optional.

1062 **7.2.4.1.1 HTTP/HTTPS and Redfish**

1063 A full discussion of the HTTP/HTTPS protocol is beyond the scope of this specification; however, a
 1064 minimalist overview of key concepts relevant to Redfish Device Enablement follows. Readers are directed
 1065 to [DSP0266](#) for more detailed information on the usage of HTTP and HTTPS with Redfish and to
 1066 standard documentation for more general information on the HTTP/HTTPS protocols themselves.

1067 **7.2.4.1.1.1 Redfish Operation requests**

1068 Every Redfish request has a target URI to which it should be applied; this URI is the target of the
 1069 HTTP/HTTPS verb listed in Table 31. The URI may consist of several parts of interest for purposes of this
 1070 specification: a prefix that points to the RDE Device being managed, a subpath within the RDE Device
 1071 management topology, a specific resource selection preceded by an octothorp character (#), and one or
 1072 more query options preceded by a question mark (?) character.

1073 Many, but not all, Redfish requests have a JSON payload associated with them. For example, a POST
 1074 operation to create a new child element in a collection would normally contain a JSON payload for the
 1075 data being supplied for that new child element.

1076 Finally, every Redfish HTTP/HTTPS request will contain a series of headers, each of which modifies it in
 1077 some fashion.

1078 7.2.4.1.1.2 Redfish Operation responses

1079 The response to a Redfish HTTP/HTTPS request will also contain several elements. First, the response
1080 will contain a status code that represents the result of the operation. Like for requests, [DSP0266](#) defines
1081 several response headers that may need to be supplied in conjunction with a Redfish response. Finally, a
1082 JSON payload may be present such as in the case of a read operation.

1083 7.2.4.1.1.3 Generic handling of Redfish Operations

1084 Generically, to handle processing of a Redfish HTTP/HTTPS request, the MC will typically implement the
1085 following steps (This overview ignores error conditions, timeouts, and long-lived Tasks. A much more
1086 detailed treatment may be found in clause 9.):

- 1087 1) Parse the prefix of the supplied URI to pinpoint the RDE Device that the operation targets
- 1088 2) Parse the RDE Device portion of the URI to identify the specific place in the RDE Device's
1089 management topology targeted by the operation
- 1090 3) Identify the Redfish Resource PDR that represents that portion of the data
- 1091 4) Using the HTTP/HTTPS verb and other request information, determine the type of Redfish
1092 operation that the client is trying to perform
- 1093 5) Translate any request headers (clause 7.2.4.2) and query options (clause 7.2.4.3) into
1094 parameters to the corresponding PLDM request message(s)
- 1095 6) Translate the JSON payload, if present, into a corresponding BEJ (clause 8) payload for the
1096 request, using a dictionary appropriate for the target Redfish Resource PDR
- 1097 7) Send the PLDM for Redfish Device Enablement RDEOperationInit command (clause 12.1) to
1098 begin the Operation
- 1099 8) Send any BEJ payload to the RDE Device via one or more PLDM for Redfish Device
1100 Enablement MultipartSend commands (clause 13.1) unless it was small enough to be inlined in
1101 the RDEOperationInit command
- 1102 9) Send any request parameters to the RDE Device via the PLDM for Redfish Device Enablement
1103 SupplyCustomRequestParameters command (clause 12.2)
- 1104 10) If there was a payload but no request parameters, send the RDEOperationStatus command
1105 (clause 12.5)
- 1106 11) Retrieve and decode any BEJ-encoded JSON data for any Operation response payloads via
1107 one or more PLDM for Redfish Device Enablement MultipartReceive commands (clause 13.2)
- 1108 12) Retrieve any response parameters via the PLDM for Redfish Device Enablement
1109 RetrieveCustomResponseHeaders command (clause 12.3)
- 1110 13) Send the PLDM for Redfish Device Enablement RDEOperationComplete command (clause
1111 12.4) to inform the RDE Device that it may discard any data structures associated with the Task
- 1112 14) Translate the BEJ response payload, if present, into JSON format for return to the client, using
1113 an appropriate dictionary
- 1114 15) Prepare and send the final response to the client, adding the various HTTP/HTTPS response
1115 headers (clause 7.2.4.2) appropriate to the type of Redfish operation that was just performed

1116 7.2.4.2 Redfish operation headers

1117 Several HTTP/HTTPS transport layer headers modify Redfish operations when translated in the context
1118 of RDE Operations. These are summarized in Table 32. Implementation notes for how the MC and RDE
1119 Device shall support some of these modifiers – when attached to Redfish operations – may be found in

1120 the indicated subsections. For headers not listed here, the implementation is outside the scope of this
 1121 specification; implementers shall refer to [DSP0266](#) and standard HTTP/HTTPS documentation for more
 1122 information on processing these headers.

1123 **Table 32 – Redfish operation headers**

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

1124 **7.2.4.2.1 If-Match request header**

1125 The MC shall support the If-Match header when applied to Redfish HTTP/HTTPS PUT and PATCH
 1126 operations; support for other Redfish operations is optional.

1127 The parameter for this header is an ETag.

1128 In order to support this header, the MC shall convey the supplied ETag to the RDE Device via the
 1129 ETag[0] field of the PLDM SupplyCustomRequestParameters command (clause 12.2) request message
 1130 and supply the value ETAG_IF_MATCH for the ETagOperation field of the same message. For this
 1131 header, the MC shall supply the value 1 for the ETagCount field of the request message.

1132 When the RDE Device receives an ETAG_IF_MATCH within the ETagOperation field in the
 1133 SupplyCustomRequestParameters command, it shall verify that the ETag matches the current state of the
 1134 targeted schema data instance before proceeding with the RDE Operation. In the event of a mismatch, it
 1135 shall respond to the SupplyCustomRequestParameters command with completion code
 1136 ERROR_ETAG_MATCH.

1137 In the event that both an If-Match and If-None-Match request header are supplied by the client, the MC
1138 shall respond with HTTP status code 400 – Bad Request – to the client and stop processing the request.
1139 The MC shall not send such a malformed request to the RDE Device.

1140 **7.2.4.2.2 If-None-Match request header**

1141 The MC may optionally support the If-None-Match header when applied to Redfish HTTP/HTTPS PUT
1142 and PATCH operations.

1143 The parameter for this header is a comma-separated list of ETags.

1144 In order to support this header, the MC shall convey the supplied ETag(s) to the RDE Device via the
1145 ETag[i] fields of the PLDM SupplyCustomRequestParameters command (clause 12.2) request message
1146 and supply the value ETAG_IF_NONE_MATCH for the ETagOperation field of the same message. For
1147 this header, the MC shall supply the value N for the ETagCount field of the request message where N is
1148 the number of entries in the comma-separated list.

1149 When the RDE Device receives an ETAG_IF_NONE_MATCH within the ETagOperation field in the
1150 SupplyCustomRequestParameters command, it shall verify that none of the supplied ETags matches the
1151 current state of the targeted schema data instance before proceeding with the RDE Operation. In the
1152 event of a match, it shall respond to the SupplyCustomRequestParameters command with completion
1153 code ERROR_ETAG_MATCH.

1154 In the event that both an If-Match and If-None-Match request header are supplied by the client, the MC
1155 shall respond with HTTP status code 400 – Bad Request – to the client and stop processing the request.
1156 The MC shall not send such a malformed request to the RDE Device.

1157 **7.2.4.2.3 Custom HTTP headers**

1158 The MC shall support custom headers when applied to any Redfish HTTP/HTTPS operation. For
1159 purposes of this specification, the term custom headers shall refer to any HTTP/HTTPS header for which
1160 no standard handling is described either in this specification or in [DSP0266](#). Per the HTTP/HTTPS
1161 specifications, custom headers typically have their header name prefixed with “X-“.

1162 The parameters for custom headers will vary by actual header type.

1163 In order to support custom headers, the MC shall bundle them into the request message for an invocation
1164 of the SupplyCustomRequestParameters command (clause 12.2). To do so, the MC shall set the
1165 HeaderCount request parameter to the number of custom request parameters. For each custom request
1166 parameter *n*, the MC shall set HeaderName[*n*] and HeaderParameter[*n*] to the name and value of the
1167 request parameter, respectively.

1168 When the RDE Device receives custom request parameters, it may perform any custom handling for the
1169 parameter. If it does not support a specific custom request parameter received, the RDE Device shall
1170 respond with the ERROR_UNRECOGNIZED_CUSTOM_HEADER completion code.

1171 Similarly, when the RDE Device has custom response parameters to send back to a client, it shall set the
1172 HaveCustomResponseParameters flag in the OperationExecutionFlags response field of the
1173 RDEOperationInit, SupplyCustomRequestParameters, or RDEOperationStatus command to ask the MC
1174 to retrieve these parameters. Then, in response to the RetrieveCustomResponseParameters command
1175 (clause 12.3), the RDE Device shall set the ResponseHeaderCount field to the number of custom
1176 response headers it wants to send back to the client. For each custom response parameter *n*, the RDE
1177 Device shall set HeaderName[*n*] and HeaderParameter[*n*] to the name and value of the response
1178 parameter, respectively.

1179 Following completion of the main Operation, the MC shall check the HaveCustomResponseParameters
1180 flag in the OperationExecutionFlags response field to see if the RDE Device is supplying custom
1181 response headers. If the flag is set (with value 1b), the MC shall use the
1182 RetrieveCustomResponseParameters command (clause 12.3) to recover them from the RDE Device. The
1183 MC shall then append the recovered headers to the Redfish Operation response.

1184 7.2.4.2.4 ETag response header

1185 The MC shall provide an ETag header in response to every Redfish HTTP/HTTPS GET or HEAD
1186 operation.

1187 The parameter for this header is an ETag.

1188 In order to support this header, the RDE Device shall generate a digest of the schema data instance after
1189 each modification to the data in accordance with [RFC 7232](#). When the MC begins a GET or HEAD
1190 operation to the RDE Device via a PLDM RDEOperationInit command (clause 12.1), the RDE Device
1191 shall populate the ETag field in the response message to the command where the RDE Operation has
1192 completed (one of RDEOperationInit, SupplyCustomRequestParameters, or RDEOperationStatus) with
1193 this digest.

1194 When it receives an ETag field in the response message for a completed RDE Operation, the MC shall
1195 then populate this header with the digest it receives.

1196 7.2.4.2.5 Link response header

1197 The MC shall provide one or more Link headers in response to every Redfish HTTP/HTTPS GET and
1198 HEAD operation as described in [DSP0266](#).

1199 The parameter for this header is a URI.

1200 This header has three forms as described in [DSP0266](#); all three shall be supported by MCs. The handling
1201 for these three forms is detailed in the next three clauses.

1202 No special action is needed on the part of an RDE Device to support any form of the link response
1203 header.

1204 7.2.4.2.5.1 Schema form

1205 The MC shall provide a link header with “rel=describedby” to provide a schema link for the data that is or
1206 would be returned in response to a Redfish HTTP/HTTPS GET or HEAD operation. The MC may obtain
1207 this link in any of several manners:

- 1208 • An @odata.context annotation in read data may contain the schema reference.
- 1209 • The MC may have the schema reference cached.
- 1210 • The MC may retrieve the schema reference directly from the PDR encapsulating the instance of
1211 the schema data by invoking the PLDM GetSchemaURI command (clause 11.4).

1212 An example of a schema form link header is as follows; readers are referred to [DSP0266](#) for more detail:

1213

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

1214 7.2.4.2.5.2 Annotation form

1215 The MC should provide a link header to provide an annotation link for the data that is or would be
1216 returned in response to a Redfish HTTP/HTTPS GET or HEAD operation. The MC may obtain this link in
1217 any of several manners:

- 1218 • The MC may inspect annotations to determine whether @odata or @Redfish annotations are
1219 used.
- 1220 • The MC may retrieve the schema reference directly from the PDR encapsulating the instance of
1221 the schema data by invoking the PLDM GetSchemaURI command (clause 11.4)

1222 An example of an annotation form link header is as follows; readers are referred to [DSP0266](#) for more
1223 detail:

1224

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

1225 7.2.4.2.5.3 Passthrough form

1226 The MC shall translate link annotations returned from the RDE Device in response to a Redfish
1227 HTTP/HTTPS GET operation into link headers. In this form, the MC shall also include the schema path to
1228 the link.

1229 An example of a passthrough form link header is as follows; readers are referred to [DSP0266](#) for more
1230 detail:

1231

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

1232 7.2.4.2.6 Location response header

1233 The MC shall provide a Location header in response to every Redfish HTTP/HTTPS POST that effects a
1234 successful create operation. The MC shall also provide a Location header in response to every Redfish
1235 Operation that spawns a long-running Task when executed as an RDE Operation.

1236 The parameter for this header is a URI.

1237 In order to support this header for completed create operations, the RDE Device shall populate the
1238 NewResourceID response parameter in the response message for the
1239 RetrieveCustomResponseParameters command (clause 12.3) with the Resource ID of the newly created
1240 collection element. Upon receipt, the MC shall combine this resource ID with the topology information
1241 contained in the Redfish Resource PDRs for the targeted PDR up through the device component root to
1242 create a local URI portion that it shall then combine with its external management URI for the RDE Device
1243 to build a complete URI for the newly added collection element. The MC shall then populate this header
1244 with the resulting URI.

1245 In order to support this header for Redfish Operations that spawn long-running Tasks when executed as
1246 RDE Operations, the MC shall generate a TaskMonitor URL for the Operation and populate the Location
1247 header with the generated URL. See clause 7.2.6 for more details.

1248 7.2.4.2.7 Cache-Control response header

1249 The MC shall provide a Cache-Control header in response to every Redfish HTTP/HTTPS GET or HEAD
1250 operation.

1251 In order to support this header for HTTP/HTTPS GET operations, the RDE Device shall mark the
1252 CacheAllowed flag in the OperationExecutionFlags field of the response message for the triggering
1253 command for the read or head Operation with an indication of the caching status of data read.

1254 When the MC reads the CacheAllowed flag in the OperationExecutionFlags field of the response
1255 message for a completed RDE Operation, it shall populate the Cache-Control response header with an
1256 appropriate value. Specifically, if the RDE Device indicates that the data is cacheable, the MC shall
1257 interpret this as equivalent to the value "public" as defined in [RFC 7234](#); otherwise, the MC shall interpret
1258 this as equivalent to the value "no-store" as defined in [RFC 7234](#).

1259 **7.2.4.2.8 Allow response header**

1260 The MC shall provide an Allow header in response to every Redfish HTTP/HTTPS operation that is
1261 rejected by the RDE Device specifically for the reason of being a disallowed operation, giving the
1262 ERROR_NOT_ALLOWED completion code (clause 7.5). The MC shall additionally provide an Allow
1263 response header in response to every GET (or HEAD, if supported) Redfish Operation.

1264 In order to support this header, when the RDE Device responds to an RDE command with
1265 ERROR_NOT_ALLOWED, it shall populate the PermissionFlags field of its response message with an
1266 indication of the operations that are permitted.

1267 When the MC reads the PermissionFlags field of the response message for a completed RDE Operation,
1268 the MC shall populate this header with the supplied information.

1269 **7.2.4.2.9 Retry-After response header**

1270 The MC shall provide a Retry-After header in response to every non-HEAD Redfish Operation that when
1271 conveyed to the RDE Device results in any transient failure (ERROR_NOT_READY; see clause 7.5).

1272 The parameter for this header is the length of time in seconds the client should wait before retrying the
1273 request.

1274 When the RDE Device needs to defer an RDE Operation, it shall return ERROR_NOT_READY in
1275 response to the RDEOperationInit command that begins the Operation. The RDE Device must now
1276 choose whether to supply a specific deferral timeframe or to use the default deferral timeframe. To specify
1277 a specific deferral timeframe, the RDE Device shall also set the HaveCustomResponseParameters flag in
1278 the OperationExecutionFlags response field of the RDEOperationInit command to inform the MC that it
1279 should retrieve deferral information. Then, if it did set the HaveCustomResponseParameters flag, in
1280 response to the RetrieveCustomResponseParameters command (clause 12.3), the RDE Device shall set
1281 the DeferralTimeframe and DeferralUnits parameters appropriately to indicate how long it is requesting
1282 the client to wait before resubmitting the request.

1283 As an alternative to specifying a deferral timeframe via the response message for
1284 RetrieveCustomResponseParameters, the RDE Device may skip setting the
1285 HaveCustomResponseParameters flag in the OperationExecutionFlags response field of the
1286 RDEOperationInit command to request that the MC supply a default deferral timeframe on its behalf.

1287 When it receives the response to the RDEOperationInit command, the MC shall check the
1288 HaveCustomResponseParameters flag in the OperationExecutionFlags response field to see if the RDE
1289 Device has an extended response. If the flag is set (with value 1b), the MC shall use the
1290 RetrieveCustomResponseParameters command (clause 12.3) to recover the deferral timeframe from the
1291 DeferralTimeframe and DeferralUnits fields of the response message. If the flag was not set, or if the RDE
1292 Device supplied an unknown deferral timeframe (0xFF), the MC shall use a default value of 5 seconds. It
1293 shall then populate this header with the deferral value.

1294 Both the MC and RDE Device shall be prepared for possibility that the client may retry the operation
1295 before this deferral timeframe elapses: Operations can be re-initiated by impatient end users.

1296 **7.2.4.3 Redfish Operation request query options**

1297 In addition to HTTP/HTTPS headers, the standard Redfish management protocol defines several query
 1298 options that a client may specify in a URI to narrow the request in Redfish GET Operations. For any query
 1299 option not listed here, the MC may support it in a fashion as described in [DSP0266](#).

1300 **Table 33 – Redfish operation request query options**

Query Option	Clause	Description	Example
\$skip	7.2.4.3.1	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)

1301 **7.2.4.3.1 \$skip query option**

1302 The MC should support \$skip query options when provided as part of a target URI for a Redfish
 1303 HTTP/HTTPS GET operation.

1304 The parameter for this query option is an integer representing the number of members of a resource
 1305 collection to skip over. See [DSP0266](#) for more details on the usage of \$skip.

1306 To support this query option, the MC shall supply the \$skip parameter in the CollectionSkip field of the
 1307 SupplyCustomRequestParameters (clause 12.2) request message. In the event that this query option is
 1308 not supplied as part of the target URI for an HTTP/HTTPS GET operation, the MC shall supply a value of
 1309 zero in this field if it otherwise needs to supply extended request parameters; it shall not send the
 1310 SupplyCustomRequestParameters just to supply a value of zero for the CollectionSkip field.

1311 When processing an RDE read Operation for a resource collection, the RDE Device shall check the
 1312 CollectionSkip parameter from the SupplyCustomRequestParameters request message to determine the
 1313 number of members to skip over in its response, per [DSP0266](#). In the event that the MC did not indicate
 1314 the presence of extended request parameters, the RDE Device shall interpret this as a CollectionSkip
 1315 value of zero. If the parameter for \$skip exceeds the number of elements in the collection, the RDE
 1316 Device shall return ERROR_OPERATION_FAILED and, in accordance with the Redfish standard
 1317 [DSP0266](#) respond with an annotation specifying that the value is invalid (see
 1318 QueryParameterOutOfRange in the Redfish base message registry).

1319 7.2.4.3.2 \$stop query option

1320 The MC should support \$stop query options when provided as part of the target URI for a Redfish
1321 HTTP/HTTPS GET operation.

1322 The parameter for this query option is an integer representing the number of members of a resource
1323 collection to return. See [DSP0266](#) for more details on the usage of \$stop. If the parameter for \$stop
1324 exceeds the remaining number of members in a resource collection, the number returned shall be
1325 truncated to those remaining.

1326 To support this query option, the MC shall supply the \$stop parameter in the CollectionTop field of the
1327 SupplyCustomRequestParameters (clause 12.2) request message. In the event that this query option is
1328 not supplied as part of the target URI for an HTTP/HTTPS GET operation, the MC shall supply a value of
1329 0xFFFF in this field; it shall not send the SupplyCustomRequestParameters just to supply a value of
1330 unlimited for the CollectionTop field.

1331 When processing an RDE read Operation for a resource collection, the RDE Device shall check the
1332 CollectionTop parameter from the SupplyCustomRequestParameters request message to determine the
1333 number of members to respond with, per [DSP0266](#). The RDE Device shall interpret a value of 0xFFFF as
1334 indicating that there is no limit to the number of members it should return for the referenced resource
1335 collection. In the event that the MC did not indicate the presence of extended request parameters, the
1336 RDE Device shall interpret this as a CollectionTop value of unlimited.

1337 7.2.4.3.3 \$expand query option

1338 The MC should support \$expand query options when provided as part of the target URI for a Redfish
1339 HTTP/HTTPS GET operation.

1340 The parameter for this query option is a string representing the links (Navigation properties) to expand in
1341 place, “gluing together” the results of multiple reads into a single JSON response payload. This parameter
1342 may be an absolute string specifying the exact link to be expanded, or it may be any of three wildcards.
1343 The first wildcard, an asterisk (*), means that all links should be expanded. The second wildcard, a dot (.),
1344 means that subordinate links (those that are directly referenced i.e., not in the Links Property section of
1345 the resource) should be expanded. The third wildcard, a tilde (~), means that dependent links (those that
1346 are not directly referenced i.e., in the Links Property section of the resource) should be expanded. See
1347 [DSP0266](#) for more details on the usage of \$expand.

1348 No special action is required of the MC to support this query option other than tracking that it is present
1349 for use with the \$levels and \$select qualifiers. If the \$levels query option qualifier is not present in
1350 conjunction with the \$expand query option, the MC shall treat this as equivalent to \$levels=1.

1351 No action is needed on the part of an RDE Device to support this query option.

1352 7.2.4.3.4 \$levels query option qualifier

1353 The MC should support the \$levels qualifier to the \$expand query option when provided as part of the
1354 target URI for a Redfish HTTP/HTTPS GET operation or when provided implicitly by having \$expand
1355 provided as part of a Redfish HTTP/HTTPS GET operation without having the \$levels query option
1356 qualifier supplied.

1357 The parameter for this query option is an integer representing the number of schema links to expand into.
1358 If no \$level qualifier is present, the MC shall interpret this as equivalent to \$levels=1.

1359 To support this parameter, the MC can select between two choices: passing it on to the RDE Device or
1360 supporting it itself. The method by which this choice is made is implementation-specific and out of scope
1361 for this specification. If the RDE Device indicates that it cannot support \$levels expansion by setting the
1362 expand_support bit to zero in the DeviceCapabilitiesFlags in the response message to the

- 1363 NegotiateRedfishParameters command (clause 11.1), or if the expansion type is not “All Links” (see
1364 clause 7.2.4.3.3), the MC shall not select passing it to the RDE Device.
- 1365 If the MC chooses to pass this query option to the RDE Device, it shall transmit the supplied value to the
1366 RDE Device via the SupplyCustomRequestParameters command in the LinkExpand parameter.
- 1367 If the MC chooses to handle this query option itself, it shall recursively issue reads to “expand out” data
1368 for links embedded in data it reads. Such links may be identified during the BEJ decode process as tuples
1369 with a format of bejResourceLink (clause 5.3.21). The corresponding value of the node represents the
1370 Resource ID for the Redfish Resource PDR representing the data to embed within the structure of data
1371 already read. The \$levels qualifier dictates the depth of recursion for this process.
- 1372 When the RDE Device receives a LinkExpand value of greater than zero in extended request parameters
1373 as part of an RDE read operation, it shall “expand out” all resource links (as defined in [DSP0266](#)) to the
1374 indicated depth by encoding them as bejResourceLinkExpansions in the response BEJ data for the
1375 command. If the RDE Device previously did not set the expand_support flag in the
1376 DeviceCapabilitiesFlags field of the NegotiateRedfishParameters command, it may instead ignore the
1377 value (treating it as zero).
- 1378 Implementers should refer to [DSP0266](#) for more details and caveats to be applied when expanding links
1379 with \$levels > 1.
- 1380 **7.2.4.3.5 \$select query option qualifier**
- 1381 The MC may support \$select as a qualifier to the \$expand query option or as a standalone query option,
1382 provided in either case as part of the target URI for a Redfish HTTP/HTTPS GET operation.
- 1383 The parameter for this query option is a string containing a comma-separated list of properties to be
1384 retrieved from the GET operation; the caller is asking that all other properties be suppressed. See
1385 [DSP0266](#) for more details on the usage of \$select.
- 1386 If it supports this parameter, the MC should perform the GET operation normally up to the point of
1387 retrieving BEJ-formatted data from the RDE Device. When decoding the BEJ data, however, the MC
1388 should silently discard any property not part of the \$select list.
- 1389 No action is needed on the part of an RDE Device to support this query option.
- 1390 **7.2.4.4 HTTP/HTTPS status codes**
- 1391 The MC shall comply with [DSP0266](#) in all matters pertaining to the HTTP/HTTPS status codes returned
1392 for Redfish GET, PATCH, PUT, POST, DELETE, and HEAD operations. Typical status codes for
1393 operational errors may be found in clause 7.5.
- 1394 **7.2.4.5 Multihosting and Operations**
- 1395 A single RDE Device may find that it is attached to multiple MCs. This can introduce complications from
1396 concurrency if conflicting Operations are issued and requires an RDE Device to decide whether an
1397 Operation should be visible to an MC other than the one that issued it. Support for multiple MCs is out of
1398 scope for this specification. In particular, the behavior of the RDE Device in the face of concurrent
1399 commands from multiple MCs is undefined.
- 1400 **7.2.5 PLDM RDE Events**
- 1401 An Event is an abstract representation of any happening that transpires in the context of the RDE Device,
1402 particularly one that is outside of the normal command request/response sequence. A Redfish Message
1403 Event consists of JSON data that includes elements such as the index of a standardized text string and a

1404 collection of parameters that provide clarification of the specifics of the Event that has transpired. The full
1405 schema for Events may be found in the standard Redfish Message schema; additionally, OEM extensions
1406 to this schema are possible.

1407 In this specification, a second class of events, Task Executed Events, allow RDE Devices to report that a
1408 Task has finished executing and that the MC should retrieve Operation results. The data for these events
1409 includes elements such as the Operation identifier and the resource with which the Operation is
1410 associated.

1411 As with any other PLDM eventing, the RDE Device advertises that it supports Events by listing support for
1412 the PLDM for Platform Monitoring and Control SetEventReceiver command (see [DSP0248](#)). The MC, for
1413 its part, may then select between two methods by which it will know that Events are available. If the MC
1414 configured the RDE Device to use asynchronous events through the SetEventReceiver command, the
1415 RDE Device shall use the PLDM for Platform Monitoring and Control PlatformEventMessage command
1416 (see [DSP0248](#)) to inform the MC by sending the Event directly. Otherwise, the RDE Device can be
1417 configured to polling mode using the same SetEventReceiver command. The MC uses the PLDM for
1418 Platform Monitoring and Control PollForPlatformEventMessage command (see [DSP0248](#)) for this
1419 purpose. The selection of any polling interval is determined by the MC and is outside the scope of this
1420 specification.

1421 Whether retrieved synchronously or asynchronously, once the MC gets the Event, it may process it.
1422 Redfish Message Events are packaged using the redfishMessageEvent eventClass; Task Executed
1423 Events are packaged using the redfishTaskExecutedEvent eventClass (see [DSP0248](#) for both
1424 eventClasses).

1425 Handling of Task Executed Events is described with Tasks in clause 7.2.6. For Redfish Message Events,
1426 the MC may decode the BEJ-formatted payload of Event data using the appropriate Event schema
1427 dictionary specific to the PDR from which the message was sent.

1428 For a more detailed view of the Event lifecycle, see clause 9.3.

1429 NOTE Events are optional in standard Redfish; however, support for Task Executed Events is mandatory in this
1430 specification if the RDE Device supports asynchronous execution for long-running Operations.

1431 7.2.5.1 [MC] Event subscriptions

1432 In Redfish, a client may request to be notified whenever a Redfish Event occurs. Per [DSP0266](#), to do so,
1433 the client uses a Redfish CREATE operation to add a record to the EventSubscription collection. This
1434 record in turn contains information on the various Event types that the client wishes to receive Events for.
1435 To unsubscribe, the client uses a Redfish DELETE operation to remove its record. Among other
1436 properties, the EventSubscription record contains a URI to which the Event should be forwarded. MCs
1437 that support Events shall support at least one Redfish event subscription.

1438 Event types are global across all schemas; there is no provision at this time ([DSP0266](#) v1.6) in Redfish
1439 for a client to subscribe to just one schema at a time. Further, there is generally no capacity for an RDE
1440 Device to send an HTTP/HTTPS record directly to an external recipient. Events are optional in Redfish;
1441 however, if the MC chooses to provide Event subscription support, it must comply with the following
1442 requirements:

- 1443 • The MC shall provide full support for the EventSubscription collection as a Redfish Provider per
1444 [DSP0266](#).
- 1445 • When it receives an Event subscription request (in the form of a Redfish CREATE operation on
1446 the EventSubscription collection), the MC shall parse the EventType array property of the
1447 request to identify the type or types of Events the client is interested in receiving
- 1448 • When the MC receives a Redfish Message Event from an RDE Device, it shall check the
1449 EventType of the Event received against the desired EventType for each active client. For each
1450 match, the MC shall forward the Event (translating any @Message.ExtendedInfo annotations, of
1451 course, from BEJ to JSON) to the client as a standard Redfish Provider for the Event service.

1452 7.2.6 Task support

1453 In PLDM for Redfish Device Enablement, every Redfish HTTP/HTTPS operation is effected as an RDE
1454 Operation. Most Operations, once sent to the RDE Device for execution, may be executed quickly and
1455 the results sent directly in the response message to the request message that triggered them.

1456 It may however transpire that in order for an RDE Device to complete an Operation, it requires more time
1457 than the available window within which the RDE Device is required to send a response. In this case, the
1458 RDE Device has two possible paths to follow. If the current number of extant Tasks is less than the RDE
1459 Device/MC capability intersection (as determined from the call to NegotiateRedfishParameters; see
1460 clause 11.1), the RDE Device shall mark the Operation as a long-running Task and execute it
1461 asynchronously. Otherwise, the RDE Device shall return ERROR_CANNOT_CREATE_TASK in its
1462 response message to indicate that no new Task slots are available (see clause 7.5).

1463 While the internal data structures used by an RDE Device to manage an Operation are outside the scope
1464 of this specification, they should include at a minimum the rdeOpID assigned (usually by the MC) when
1465 the Operation was first created. This allows the MC to reference the Task in subsequent commands to kill
1466 it (RDEOperationKill, clause 12.6) or query its status (RDEOperationStatus, clause 12.5).

1467 For its part, the MC shall provide full support for the Task collection as a Redfish Provider per [DSP0266](#).
1468 When the MC finds that an Operation has spawned a Task, it shall perform the following steps in order to
1469 comply with the requirements of [DSP0266](#):

- 1470 1) The MC shall instantiate a new TaskMonitor URL and a new member of the Task collection. The
1471 TaskMonitor URL should incorporate or reference (such as via a lookup table) the following data so
1472 that it can map from the TaskMonitor URL back to the correct Redfish resource – and thus the
1473 correct dictionary – for providing status query updates:
 - 1474 a) The ResourceID for the resource to which the RDE Operation was targeted
 - 1475 b) The rdeOpID for the Operation itself
- 1476 2) The MC shall return response code 202, Accepted to the client and include the Location response
1477 header populated with the TaskMonitor URL.
- 1478 3) In response to a subsequent Redfish GET Operation applied to the TaskMonitor URL or to the Task
1479 collection member, the MC shall invoke the RDEOperationStatus (see clause 12.5) command to
1480 obtain the latest status for the Operation and communicate it to the client in accordance with
1481 [DSP0266](#). If the GET was applied to a TaskMonitor URL and the Operation has completed, the MC
1482 shall supply the complete results to the client.
 - 1483 a) If the result of the RDEOperationStatus command was that the Operation has finished
1484 execution, the MC shall delete both the TaskMonitor URL and the Task collection member
1485 associated with the Operation.
- 1486 4) In response to a Redfish DELETE Operation applied to the TaskMonitor URL or to the Task
1487 collection member, the MC shall attempt to abort the associated Operation via the RDEOperationKill
1488 (see clause 12.6) command. It shall then remove both the TaskMonitor URL and the Task collection
1489 member.
- 1490 5) If the RDE Operation finishes before the client polls the TaskMonitor URL, the MC may collect and
1491 store the results of the Operation.
 - 1492 a) In accordance with [DSP0266](#), the MC should retain Operation results until the client retrieves
1493 them. It may refuse to accept further Operations until previous results have been claimed.
 - 1494 b) If the client attempts to collect Operation results after the MC has discarded them, the MC shall
1495 respond with an error HTTP status code as defined in [DSP0266](#).

1496 When the RDE Device finishes execution of a Task, it generates a Task Executed Event to inform the MC
 1497 of this status change. The MC can then retrieve the results (via RDEOperationStatus) and eventually
 1498 forward them to the client. To mark the Task as complete and allow the RDE Device to discard any
 1499 internal data structures used to manage the Task, the MC shall call RDEOperationComplete (clause
 1500 12.4).

1501 For a more detailed overview of the Operation/Task lifecycle from the MC's perspective, see clause
 1502 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
 1503 finite state machine for the Task lifecycle (from the RDE Device's perspective) may be found in clause
 1504 9.2.3.

1505 **7.3 Type code**

1506 Refer to [DSP0245](#) for a list of PLDM Type Codes in use. This specification uses the PLDM Type Code
 1507 000110b as defined in [DSP0245](#).

1508 **7.4 Transport protocol type supported**

1509 PLDM can support bindings over multiple interfaces; refer to [DSP0245](#) for the complete list. All transport
 1510 protocol types can be supported for the commands defined in Table 47.

1511 **7.5 Error completion codes**

1512 Table 34 lists PLDM completion codes for Redfish Device Enablement. The usage of individual error
 1513 completion codes are defined within each of the PLDM command clauses.

1514 **Table 34 – 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.
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
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
0x82	ERROR_NOT_ALLOWED	The client and/or MC is not allowed to perform the requested Operation.	405 Method Not Allowed
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
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

Value	Name	Description	HTTP Error Code
0x85	ERROR_OPERATION_UNKILLABLE	An attempt was made to kill an Operation that has already finished execution or that cannot be aborted.	409 Conflict
0x86	ERROR_OPERATION_EXISTS	An Operation initialization was attempted with an rdeOpID that is currently active.	N/A – MC retries with a new rdeOpID
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
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
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
0x90	ERROR_UNRECOGNIZED_CUSTOM_HEADER	The RDE Device received a custom X-header (via SupplyCustomRequestParameters) that it does not support	412 Precondition Failed
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)
0x92	ERROR_NO_SUCH_RESOURCE	An Operation command was invoked with a resource ID that does not exist	404, Not Found

1515 HTTP Error codes returned when Operations complete with standard PLDM completion codes shall be as
 1516 follows:

1517

Table 35 – 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

1518 **7.6 Timing specification**

1519 Table 36 below defines timing values that are specific to this document. The table below defines the
 1520 timing parameters defined for the PLDM Redfish Specification. In addition, all timing parameters listed in
 1521 [DSP0240](#) for command timeouts, command response times, and number of retries shall also be followed.

1522

Table 36 – 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.

Timing specification	Symbol	Min	Max	Description
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.

1523 8 Binary Encoded JSON (BEJ)

1524 This clause defines a binary encoding of Redfish JSON data that will be used for communicating with
 1525 RDE Devices. At its core, BEJ is a self-describing binary format for hierarchical data that is designed to
 1526 be straightforward for both encoding and decoding. Unlike in ASN.1, BEJ uses no contextual encodings;
 1527 everything is explicit and direct. While this requires the insertion of a bit more metadata into BEJ encoded
 1528 data, the tradeoff benefit is that no lookahead is required in the decoding process. The result is a
 1529 significantly streamlined representation that fits in a very small memory footprint suitable for modern
 1530 embedded processors.

1531 8.1 BEJ design principles

1532 The core design principles for BEJ are focused around it being a compact binary representation of JSON
 1533 that is easy for low-power embedded processors to encode, decode, and manipulate. This is important
 1534 because these ASICs typically have highly limited memory and power budgets; they must be able to
 1535 process data quickly and efficiently. Naturally, it must be possible to fully reconstruct a textual JSON
 1536 message from its BEJ encoding.

1537 The following design principles guided the development of BEJ:

- 1538 1) It must be possible to support full expressive range of JSON.
- 1539 2) The encoding should be binary and compact, with as much of the encoding as possible
 1540 dedicated to the JSON data elements. The amount of space afforded to metadata that conveys
 1541 elements such as type format and hierarchy information should be carefully limited.
- 1542 2) There is no need to support multiple encoding techniques for one type of data; there is therefore
 1543 no need to distinguish which encoding technique is in use.
- 1544 3) Schema information – such as the names of data items – does not need to be encoded into BEJ
 1545 because the recipient can use a prior knowledge of the data organization to determine semantic
 1546 information about the encoded data. In contrast to JSON, which is unordered, BEJ must adopt
 1547 an explicit ordering for its data to support this goal.
- 1548 4) The need for contextual awareness should be minimized in the encoding and decoding process.
 1549 Supporting context requires extra lookup tables (read: more memory) and delays processing

1550 time. Everything should be immediately present and directly decodable. Giving up a few bytes
 1551 of compactness in support of this goal is a worthwhile tradeoff.

1552 **8.2 SFLV tuples**

1553 Each piece of JSON data is encoded as a tuple of PLDM type bejTuple and consists of the following:

- 1554 1) Sequence number: the index within the canonical schema at the current hierarchy level for the
 1555 datum. For collections and arrays, the sequence number is the 0-based array index of the
 1556 current element.
- 1557 2) Format: the type of data that is encoded.
- 1558 3) Length: the length in bytes of the data.
- 1559 4) Value: the actual data, encoded in a format-specific manner.

1560 These tuple elements collectively describe a single piece of JSON data; each piece of JSON data is
 1561 described by a separate tuple. Requirements for each tuple element are detailed in the following clauses.

1562 SFLV tuples are represented by elements of the bejTuple PLDM type defined in clause 5.3.5.

1563 **8.2.1 Sequence number**

1564 The Sequence Number tuple field serves as a stand-in for the JSON property name assigned to the data
 1565 element the tuple encodes. Sequence numbers align to name strings contained within the dictionary for a
 1566 given schema. Sequence numbers are represented by elements of the bejTupleS PLDM type defined in
 1567 clause 5.3.6.

1568 The low-order bit of a sequence number shall indicate the dictionary to which it belongs according to the
 1569 following table:

1570 **Table 37 – 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

1571 **8.2.2 Format**

1572 The Format tuple field specifies the kind of data element that the tuple is representing.

1573 Formats are represented by elements of the bejTupleF PLDM type defined in clause 5.3.7.

1574 **8.2.3 Length**

1575 The Length tuple field details the length in bytes of the contents of the Value tuple field.

1576 Lengths are represented by elements of the bejTupleL PLDM type defined in clause 5.3.8.

1577 **8.2.4 Value**

1578 The Value tuple field contains an encoding of the actual data value for the JSON element described by
 1579 this tuple. The format of the value tuple field is variable but follows directly from the format code in the
 1580 Format tuple field.

1581 The following JSON data types are supported in BEJ:

1582 **Table 38 – JSON data types supported in BEJ**

BEJ Type	JSON Type	Description
Null	null	An empty data type
Integer	number	A whole number: any element of JSON type number that contains neither a decimal point nor an exponent
Enum	enum	An enumeration of permissible values in string format
String	string	A null-terminated UTF-8 text string
Real	number	A non-whole number: any element of JSON type number that contains at least one of a decimal point or an exponent
Boolean	boolean	Logical true/false
Bytestring	string (of base-64 encoded data)	Binary data
Set	No named type; data enclosed in { }	A named collection of data elements that may have differing types
Array	No named type; data enclosed in []	A named collection of zero or more copies of data elements of a common type
Choice	special	The ability of a named data element to be of multiple types
Property Annotation	special	An annotation targeted to a specific property, in the format 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

1583 If the deferred_binding flag (see the bejTupleF PLDM type definition in clause 5.3.7) is set, the string
 1584 encoded in the value tuple element contains substitution macros that the MC is to supply on behalf of the
 1585 RDE Device when populating a message to send back to the client. See clause 8.3 for more details.

1586 Values are represented by elements of the bejTupleV PLDM type defined in clause 5.3.9.

1587 **8.3 Deferred binding of data**

1588 The data returned to a client from a Redfish operation typically contains annotation metadata that specify
 1589 URIs and other bits of information that are assigned by the MC when it performs RDE Device discovery
 1590 and registration. In practice, the only way for an RDE Device to know the values for these annotations
 1591 would be for it to somehow query the MC about them. Instead, we define substitution macros that the
 1592 RDE Device may use to ask the MC to supply these bits of information on its behalf. RDE Devices shall
 1593 not invoke substitution macros for information that they know and can provide themselves.

1594 All substitution macros are bracketed with the percent sign (%) character. While it would in theory be
 1595 possible for the MC to check every string it decodes for the presence of this escape character, in practice

1596 that would be an inefficient waste of MC processing time. Instead, the RDE Device shall flag any string
 1597 containing substitution macros with the deferred binding bit to inform the MC of their presence; the MC
 1598 shall only perform macro substitution if the deferred binding bit is set. The MC shall support the deferred
 1599 bindings listed in Table 39.

1600

Table 39 – 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
%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
%S	The MC-assigned link to the ComputerSystem resource within which the RDE Device is located	/redfish/v1/Systems/437XR1138R2
%C	The MC-assigned link to the Chassis resource within which the RDE Device is located	/redfish/v1/Chassis/1U
%M	The metadata URL for the service	/redfish/v1/\$metadata
%T<resource-ID>.<n>	The MC-assigned target URI for the n th Action from the Redfish Action PDR or PDRs linked to a resource within a Redfish Resource PDR, or “/invalid.<resource-ID>.<n>” if no such action exists	/redfish/v1/Systems/437XR1138R2/Storage/1/Actions/Storage.SetEncryptionKey /invalid.123.6
%I<resource-ID>	The MC-assigned instance identifier for the collection element representing an RDE Device (specified by the resource ID of the target PDR), or “invalid” if the PDR does not correspond to a resource immediately contained within a collection managed by the MC	437XR1138R2 invalid
%U	The UEFI Device Path assigned to the RDE Device by the MC and/or BIOS	PciRoot(0x0)/Pci(0x1,0x0)/Pci(0x0,0x0)/Scsi(0xA,0x0)
%.	Terminates a previous substitution. Shall be used only in the event that numeric data immediately follows a %T, %P, or %L macro	n/a
Any other character preceded by a % character	None – the MC shall pass the sequence exactly as found	%p %X

1601 **8.4 BEJ encoding**

1602 This clause presents implementation considerations for the BEJ encoding process. For standard resource
1603 encoding (as opposed to annotations), the BEJ conversion dictionary is built to encode the same
1604 hierarchical data format as the schema itself. Implementations should therefore track their context inside
1605 the dictionary in parallel with tracking their location in the data to be encoded. While not mandatory, a
1606 recursive implementation will prove in most cases to be the easiest approach to realize this tracking.

1607 Like with JSON encodings of data, there is no defined ordering for properties in BEJ data; encoders are
1608 therefore free to encode properties in any order.

1609 **8.4.1 Conversion of JSON data types to BEJ**

1610 Recognition of [JSON](#) data types enables them to be encoded properly. In Redfish, every property is
1611 encoded in the format “property_name” : property_value. Whitespace between syntactic elements is
1612 ignored in JSON encodings.

1613 **8.4.1.1 JSON objects**

1614 A JSON object consists of an opening curly brace ('{'), a nonempty comma-separated list of properties,
1615 and then a closing curly brace ('}'). JSON objects shall be encoded as BEJ sets with the properties inside
1616 the curly braces encoded recursively as the value tuple contents of the BEJ set. Following the precedent
1617 established in JSON, the properties contained within a JSON object may be encoded in BEJ in any order.
1618 In particular, the encoding order for a collection of properties is not required to match their respective
1619 sequence numbers.

1620 **8.4.1.2 JSON arrays**

1621 A JSON array consists of an opening square brace ('['), a nonempty comma-separated list of JSON
1622 values all of a common data type (typically objects in Redfish), and then a closing square brace. JSON
1623 arrays shall be encoded as BEJ arrays with the data inside the square braces encoded recursively as
1624 instances of the value tuple contents of the BEJ array. The immediate contents of a JSON array shall be
1625 encoded in order corresponding to their array indices.

1626 The sequence numbers for BEJ array immediate child elements shall match the zero-based array index
1627 of the children. These sequence numbers are not represented in the dictionary; it is the responsibility of a
1628 BEJ encoder/decoder to understand that this is how array data instances are handled.

1629 **8.4.1.3 JSON numbers**

1630 In JSON, there is no distinction between integer and real data; both are collected together as the number
1631 type. For BEJ, numeric data shall be encoded as a BEJ integer if it contains neither a decimal point nor
1632 an exponentiation marker ('e' or 'E') and as a BEJ real otherwise.

1633 **8.4.1.4 JSON strings**

1634 When converting JSON strings to BEJ format, a null terminator shall be appended to the string.

1635 **8.4.1.5 JSON Boolean**

1636 In JSON, Boolean data consists of one of the two sentinels “true” or “false”. These sentinels shall be
1637 encoded as BEJ Boolean data with an appropriate value field.

1638 8.4.1.6 JSON null

1639 In JSON, null data consists of the sentinel “null”. This sentinel shall be encoded as BEJ Null data only if
1640 the datatype for the property in the schema is null. For a nullable property (identified via the third tag bit
1641 from the dictionary entry or by the schema), null data shall be encoded as its standard type (from the
1642 dictionary) with length zero and no value tuple element.

1643 8.4.2 Resource links

1644 Most schemas contain links to other schemas within their properties, formatted as @odata.id annotations.
1645 When encoding these links in BEJ, the bejResourceLink (simple links) or bejResourceLinkExpansion
1646 (links expanded to include the full resource data for the link target) type shall be used to encode the
1647 ResourceID of the Redfish Resource PDR for the link target. Either type may be supplied for a property or
1648 annotation indicated in the dictionary as being of type bejResourceLink.

1649 8.4.3 Annotations

1650 Redfish annotations may be recognized as properties with a name string containing the “at” sign ('@').
1651 Several annotations are defined in Redfish, including some that are mandatory for inclusion with any
1652 Redfish GET Operation. The RDE Device is responsible for ensuring that these mandatory annotations
1653 are included in the results of an RDE read Operation.

1654 Annotations in Redfish have two forms:

- 1655 • Standalone form annotations have the form “@annotation_class.annotation_name” :
1656 annotation_value.
 - 1657 ○ Example: “@odata.id”: “/redfish/v1/Systems/1/”
 - 1658 ○ Standalone annotations shall be encoded with the BEJ data type listed in the annotation
1659 dictionary in the row matching the annotation name string
- 1660 • Property annotation form annotations have the form
1661 “property@annotation_class.annotation_name” : annotation_value.
 - 1662 ○ Example: “ResetType@Redfish.AllowableValues” : [“On”, “PushPowerButton”]
 - 1663 ○ Property annotation form annotations shall be encoded with the BEJ Property Annotation
1664 data type; the annotation value shall be encoded as a dependent child of the annotation
1665 entry. See clause 5.3.20.

1666 NOTE Unlike major schema resource properties, annotations have a flat namespace from which sequence
1667 numbers are drawn. To identify the sequence number for an annotation, an encoder should start at the root
1668 of the annotation dictionary and then find the string matching the annotation name (including the '@' sign
1669 and the annotation source) within this set. In particular, the sequence number for an annotation is
1670 independent of the current encoding context.

1671 Special handling is required when the RDE Device sends a message annotation to the MC. The related
1672 properties property inside the annotation’s data structure is formatted as an array of strings, but the RDE
1673 Device has only sequence numbers to work with: the RDE Device may not be able to supply the property
1674 name for the sequence number. If the RDE Device knows the name of the related property that is
1675 relevant for the message annotation, it may supply the name directly as an array element. Otherwise, it
1676 shall encode into the array element a BEJ locator by concatenating the following string components:

1677

Table 40 – 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

1678 8.4.4 Choice encoding for properties that support multiple data types

1679 If the encoder finds a property that is listed in the dictionary as being of type BEJ choice, it shall encode
 1680 the property with type bejChoice in the BEJ format tuple element. The actual value and selected data type
 1681 shall be encoded as a dependent child of the tuple containing the bejChoice element. See clauses 5.3.19
 1682 and 7.2.3.3.

1683 8.4.5 Properties with invalid values

1684 If the MC is encoding an update request from a client that includes a property value that does not match a
 1685 required data type according to the dictionary it is translating from, the MC shall in accordance with the
 1686 Redfish standard [DSP0266](#) respond to the client with HTTP status code 400 and a
 1687 @Message.ExtendedInfo annotation specifying the property with the value format error (see
 1688 PropertyValueFormatError, PropertyValueTypeError in the Redfish base message registry). Similarly, if
 1689 the value supplied for a property such as an enumeration does not match any required values, the MC
 1690 shall in accordance with the Redfish standard [DSP0266](#) respond to the client with HTTP status code 400
 1691 and a @Message.ExtendedInfo annotation specifying the property with a value not in the accepted list
 1692 (see PropertyValueNotInList in the Redfish base message registry).

1693 8.4.6 Properties missing from dictionaries

1694 When encoding JSON data, an encoder may find that the name of a property does not correspond to a
1695 string found in the dictionary. If the encoder is the RDE Device, this should never happen as the RDE
1696 Device is responsible for the dictionary. This situation therefore represents a non-compliant RDE
1697 implementation.

1698 If the MC finds that a property does not correspond to a string found in the dictionary from an RDE
1699 Device, it should in accordance with the Redfish standard [DSP0266](#) respond to the client with HTTP
1700 status code 200 or 400 and an annotation specifying the property as unsupported (see PropertyUnknown
1701 in the Redfish base message registry). The MC may continue to process the client request.

1702 8.5 BEJ decoding

1703 This clause presents implementation considerations for the BEJ decoding process.

1704 Properties in BEJ data may be encoded in any order. Decoders must therefore be prepared to accept
1705 data in whatever order it was encoded in.

1706 8.5.1 Conversion of BEJ data types to JSON

1707 When decoding from BEJ to JSON, the following rules shall be followed. In each of the following,
1708 “property_name” shall be taken to mean the name of the property or annotation as decoded from the
1709 relevant dictionary. For all data types, if the length tuple field is zero, the data shall be decoded as
1710 follows:

1711 “property_name” : null

1712 When multiple properties appear sequentially within a set, they shall be delimited with commas.

1713 8.5.1.1 BEJ Set

1714 A BEJ Set shall be decoded to the following format, with the text inside angle brackets (‘<’, ‘>’) replaced as
1715 indicated:

1716 “property_name” : { <set dependant children decoded individually as a comma-separated list > }

1717 8.5.1.2 BEJ Array

1718 A BEJ Array shall be decoded to the following format, with the text inside angle brackets (‘<’, ‘>’) replaced
1719 as indicated:

1720 “property_name” : [<array dependant children decoded individually as a comma-separated list >]

1721 8.5.1.3 BEJ Integer and BEJ Real

1722 BEJ Integers and BEJ Reals shall be decoded to the following format, with the text inside angle brackets
1723 (‘<’, ‘>’) replaced as indicated:

1724 “property_name” : “<decoded numeric value>”

1725 8.5.1.4 BEJ String

1726 BEJ Strings shall be decoded to the following format, with the text inside angle brackets (‘<’, ‘>’) replaced
1727 as indicated. When converting BEJ strings to JSON format, the null terminator shall be dropped as JSON
1728 string encodings do not include null terminators.

1729 “property_name” : “<decoded string value>”

1730 8.5.1.5 BEJ Boolean

1731 BEJ Booleans shall be decoded to the following format, with the text inside angle brackets (‘<’, ‘>’)
1732 replaced as indicated (note that the “true” and “false” sentinels are not encased in quote marks):

1733 “property_name” : <true or false, depending on the decoded value>

1734 8.5.1.6 BEJ Null

1735 BEJ Null shall be decoded to the following format:

1736 “property_name” : null

1737 8.5.1.7 BEJ Resource Link

1738 A BEJ Resource Link shall be decoded to the following format, with the text inside angle brackets (‘<’, ‘>’)
1739 replaced as indicated.

1740 “property_name” : “<URI for the resource corresponding the Redfish Resource PDR with the
1741 supplied ResourceID>”

1742 MCs shall be aware that either a BEJ Resource Link or a BEJ Resource Link Expansion may be encoded
1743 for a dictionary entry that lists its type as BEJ Resource Link.

1744 8.5.1.8 BEJ Resource Link expansion

1745 A BEJ Resource Link Expansion shall be decoded to the following format, with the text inside angle
1746 brackets (‘<’, ‘>’) replaced as indicated.

1747 <full resource data for the Redfish Resource PDR corresponding to the supplied ResourceID>

1748 NOTE property_name is not included in the decoded JSON output in this case.

1749 If the supplied ResourceID is zero and the parent resource is a collection, the MC shall use the
1750 COLLECTION_MEMBER_TYPE schema dictionary obtained from the collection resource (rather than
1751 trying to use a dictionary from the members) to decode resource data.

1752 MCs shall be aware that either a BEJ Resource Link or a BEJ Resource Link Expansion may be encoded
1753 for a dictionary entry that lists its type as BEJ Resource Link.

1754 8.5.2 Annotations

1755 This clause documents the approach for decoding the two types of Redfish annotations to JSON text.

1756 8.5.2.1 Standalone annotations

1757 Standalone annotations (data from decoded from the annotation dictionary) shall be decoded to the
1758 following format, with the bit inside angle brackets (‘<’, ‘>’) replaced as indicated:

1759 “@annotation_class.annotation_name” : “<decoded annotation value>”

1760 8.5.2.2 BEJ property annotations

1761 BEJ Property Annotations shall be decoded to the following format, with the bit inside angle brackets (‘<’,
1762 ‘>’) replaced as indicated:

1763 "property_name@annotation_class.annotation_name" : "<decoded annotation value from the
1764 annotation's dependent child node>"

1765 8.5.2.3 [MC] Related Properties in message annotations

1766 When a message annotation is sent from the RDE Device to the MC, the related properties field of
1767 message annotations requires special handling in RDE. Specifically, the array element string values are
1768 BEJ locators to individual properties, may be encoded as a colon-delimited string (see clause 8.4.3).
1769 When decoding, the MC shall check the first character of the supplied string. If it is a colon (:), the MC
1770 shall extract the individual sequence numbers for the BEJ locator, and then use them to identify the
1771 property name to send back to the client for the annotation. If the first character of the supplied string is
1772 not a colon, the MC shall return the supplied string unmodified.

1773 8.5.3 Sequence numbers missing from dictionaries

1774 It may transpire that when decoding BEJ data, a decoder finds a sequence number not in its dictionary.
1775 The handling of this case differs between the RDE Device and the MC.

1776 If the RDE Device finds an unrecognized sequence number as part of the payload for a put, patch, or
1777 create operation, the RDE Device shall in accordance with the Redfish standard [DSP0266](#) respond with
1778 an annotation specifying the sequence number as an unsupported property (see PropertyUnknown in the
1779 Redfish base message registry). The RDE Device may continue to decode the remainder of the payload
1780 and perform the requested Operation upon the portion it understands.

1781 If the MC finds an unrecognized sequence number as part of the response payload for a get or action
1782 Operation, or as part of a @Message.ExtendedInfo annotation response for any other Operation, it shall
1783 treat this as a failure on the part of the RDE Device and respond to the client with HTTP status code 500,
1784 Internal Server Error.

1785 8.5.4 Sequence numbers for read-only properties in modification Operations

1786 If the RDE Device is performing a modification operation (create, put, patch, or some actions), and it finds
1787 a sequence number corresponding to a property that is read-only, the RDE Device should in accordance
1788 with the Redfish standard [DSP0266](#) respond with an annotation specifying the sequence number as a
1789 non-updateable property (see PropertyNotWritable in the Redfish base message registry). The RDE
1790 Device may continue to decode and update with the remainder of the payload.

1791 8.6 Example encoding and decoding

1792 The following examples demonstrate the BEJ encoding and decoding processes. For illustrative
1793 purposes, we show the data collected in an XML form that happens to align with the schema; however,
1794 there is no requirement that data be stored in this form. Indeed, it is very unlikely that any RDE Device
1795 would do so.

1796 The examples in this clause use the example dictionary from clause 8.6.1.

1797 8.6.1 Example dictionary

1798 The example dictionary is based on the DummySimple JSON schema presented in Figure 5:

```
1799 {
1800   "$ref": "#/definitions/DummySimple",
1801   "$schema": "http://json-schema.org/draft-04/schema#",
1802   "copyright": "Copyright 2018 DMTF. For
1803     the full DMTF copyright policy, see http://www.dmtf.org/about/policies/copyright",
1804   "definitions": {
1805     "LinkStatus": {
1806       "enum": [
1807         "NoLink",
```

```

1808         "LinkDown",
1809         "LinkUp"
1810     ],
1811     "type": "string"
1812 },
1813 "DummySimple" : {
1814     "additionalProperties": false,
1815     "description": "The DummySimple schema represents a very simple schema used to
1816         demonstrate the BEJ dictionary format.",
1817     "longDescription": "This resource shall not be used except for illustrative
1818         purposes. It does not correspond to any real hardware or software.",
1819     "patternProperties": {
1820         "^[a-zA-Z_][a-zA-Z0-9_]*?@(odata|Redfish|Message|Privileges)\\. [a-zA-Z_][a-zA-
1821 Z0-9_.]+$": {
1822             "description": "This property shall specify a valid odata or Redfish
1823                 property.",
1824             "type": [
1825                 "array",
1826                 "boolean",
1827                 "number",
1828                 "null",
1829                 "object",
1830                 "string"
1831             ]
1832         }
1833     },
1834     "properties": {
1835         "@odata.context": {
1836             "$ref":
1837                 "http://redfish.dmtf.org/schemas/v1/odata.v4_0_1.json#/definitions/context"
1838         },
1839         "@odata.id": {
1840             "$ref":
1841                 "http://redfish.dmtf.org/schemas/v1/odata.v4_0_1.json#/definitions/id"
1842         },
1843         "@odata.type": {
1844             "$ref":
1845                 "http://redfish.dmtf.org/schemas/v1/odata.v4_0_1.json#/definitions/type"
1846         },
1847         "ChildArrayProperty": {
1848             "items": {
1849                 "additionalProperties": false,
1850                 "type": "object",
1851                 "properties": {
1852                     "LinkStatus": {
1853                         "anyOf": [
1854                             {
1855                                 "$ref": "#/definitions/LinkStatus"
1856                             },
1857                             {
1858                                 "type": "null"
1859                             }
1860                         ],
1861                         "readOnly": true
1862                     },
1863                     "AnotherBoolean": {
1864                         "type": "boolean"
1865                     }
1866                 }
1867             },
1868             "type": "array"
1869         }
1870     },
1871     "SampleIntegerProperty": {
1872         "type": "integer"
1873     },
1874     "Id": {
1875         "type": "string",

```



```

1876         "readOnly": true
1877     },
1878     "SampleEnabledProperty": {
1879         "type": "boolean"
1880     }
1881 },
1882 },
1883 "title": "#DummySimple.v1_0_0.DummySimple"
1884 }
    
```

Figure 5 – DummySimple schema

1886 NOTE This is not a published DMTF Redfish schema.

1887 In tabular form, the dictionary for DummySimple appears as shown in Table 41:

Table 41 – DummySimple dictionary (tabular form)

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

1889 Finally, in binary form, the dictionary appears as shown in Figure 6. (Colors in this example match those used in
 1890 Figure 4.)

```

1891 0x00 0x00 0x0B 0x00 0x00 0xF0 0xF0 0xF1
1892 0x12 0x01 0x00 0x00 0x00 0x00 0x00 0x16
1893 0x00 0x04 0x00 0x0C 0x7A 0x00 0x14 0x00
1894 0x00 0x3E 0x00 0x01 0x00 0x13 0x86 0x00
1895 0x56 0x01 0x00 0x00 0x00 0x00 0x00 0x03
1896 0x99 0x00 0x74 0x02 0x00 0x00 0x00 0x00
1897 0x00 0x16 0x9C 0x00 0x34 0x03 0x00 0x00
1898 0x00 0x00 0x00 0x16 0xB2 0x00 0x00 0x00
1899 0x00 0x48 0x00 0x02 0x00 0x00 0x00 0x00
    
```

1900 0x74 0x00 0x00 0x00 0x00 0x00 0x00 0x0F
 1901 0xC8 0x00 0x46 0x01 0x00 0x5C 0x00 0x03
 1902 0x00 0x0B 0xD7 0x00 0x50 0x00 0x00 0x00
 1903 0x00 0x00 0x00 0x09 0xE2 0x00 0x50 0x01
 1904 0x00 0x00 0x00 0x00 0x00 0x07 0xEB 0x00
 1905 0x50 0x02 0x00 0x00 0x00 0x00 0x00 0x07
 1906 0xF2 0x00 0x44 0x75 0x6D 0x6D 0x79 0x53
 1907 0x69 0x6D 0x70 0x6C 0x65 0x00 0x43 0x68
 1908 0x69 0x6C 0x64 0x41 0x72 0x72 0x61 0x79
 1909 0x50 0x72 0x6F 0x70 0x65 0x72 0x74 0x79
 1910 0x00 0x49 0x64 0x00 0x53 0x61 0x6D 0x70
 1911 0x6C 0x65 0x45 0x6E 0x61 0x62 0x6C 0x65
 1912 0x64 0x50 0x72 0x6F 0x70 0x65 0x72 0x74
 1913 0x79 0x00 0x53 0x61 0x6D 0x70 0x6C 0x65
 1914 0x49 0x6E 0x74 0x65 0x67 0x65 0x72 0x50
 1915 0x72 0x6F 0x70 0x65 0x72 0x74 0x79 0x00
 1916 0x41 0x6E 0x6F 0x74 0x68 0x65 0x72 0x42
 1917 0x6F 0x6F 0x6C 0x65 0x61 0x6E 0x00 0x4C
 1918 0x69 0x6E 0x6B 0x53 0x74 0x61 0x74 0x75
 1919 0x73 0x00 0x4C 0x69 0x6E 0x6B 0x44 0x6F
 1920 0x77 0x6E 0x00 0x4C 0x69 0x6E 0x6B 0x55
 1921 0x70 0x00 0x4E 0x6F 0x4C 0x69 0x6E 0x6B
 1922 0x00 0x18 0x43 0x6F 0x70 0x79 0x72 0x69
 1923 0x67 0x68 0x74 0x20 0x28 0x63 0x29 0x20
 1924 0x32 0x30 0x31 0x38 0x20 0x44 0x4D 0x54
 1925 0x46 0x00

1926 **Figure 6 – DummySimple dictionary – binary form**

1927 8.6.2 Example encoding

1928 For this example, we start with the following data (shown here in an XML representation).

1929 **NOTE** The names assigned to array elements are fictitious and inserted for illustrative purposes only. Also, the
 1930 encoding sequence presented here is only one possible approach; any sequence that generates the same
 1931 result is acceptable. Finally, for illustrative purposes we omit here the header bytes contained within the
 1932 bejEncoding type that are not part of the bejTuple PLDM type.

```
1934 <Item name="DummySimple" type="set">
1935   <Item name="ChildArrayProperty" type="array">
1936     <Item name="array element 0">
1937       <Item name="AnotherBoolean" type="boolean" value="true"/>
1938       <Item name="LinkStatus" type="enum" enumtype="String">
1939         <Enumeration value="NoLink"/>
1940     </Item>
```

```

1941     </Item>
1942     <Item name="array element 1">
1943         <Item name="LinkStatus" type="enum" enumtype="String">
1944             <Enumeration value="LinkDown"/>
1945         </Item>
1946     </Item>
1947 </Item>
1948 <Item name="Id" type="string" value="Dummy ID"/>
1949 <Item name="SampleIntegerProperty" type="number" value="12"/>
1950 </Item>

```

The first step of the encoding process is to insert sequence numbers, which can be retrieved from the dictionary. Sequence numbers for array elements correspond to their zero-based index within the array.

```

1951 <Item name="DummySimple" type="set" seqno="major/0">
1952     <Item name="ChildArrayProperty" type="array" seqno="major/0">
1953         <Item name="array element 0" seqno="major/0">
1954             <Item name="AnotherBoolean" type="boolean" value="true" seqno="major/0"/>
1955             <Item name="LinkStatus" type="enum" enumtype="String" seqno="major/1">
1956                 <Enumeration value="NoLink" seqno="major/2"/>
1957             </Item>
1958         </Item>
1959     </Item>
1960     <Item name="array element 1" seqno="major/1">
1961         <Item name="LinkStatus" type="enum" enumtype="String" seqno="major/1">
1962             <Enumeration value="LinkDown" seqno="major/0"/>
1963         </Item>
1964     </Item>
1965 </Item>
1966 </Item>
1967 <Item name="Id" type="string" value="Dummy ID" seqno="major/1"/>
1968 <Item name="SampleIntegerProperty" type="integer" value="12" seqno="major/3"/>
1969 </Item>

```

After the sequence numbers are fully characterized, they can be encoded. We encode the fact that these sequence numbers came from the major dictionary by shifting them left one bit to insert 0b as the low order bit per clause 8.2.1. As the sequence numbers are now assigned, names of properties and enumeration values are no longer needed:

```

1970 <Item type="set" seqno="0">
1971     <Item type="array" seqno="0">
1972         <Item seqno="0">
1973             <Item type="boolean" value="true" seqno="0"/>
1974             <Item type="enum" enumtype="String" seqno="2">
1975                 <Enumeration seqno="4"/>
1976             </Item>
1977         </Item>
1978     <Item seqno="2">
1979         <Item type="enum" enumtype="String" seqno="2">
1980             <Enumeration seqno="0"/>
1981         </Item>
1982     </Item>
1983 </Item>
1984 <Item type="string" value="Dummy ID" seqno="2"/>
1985 <Item type="integer" value="12" seqno="6"/>
1986 </Item>

```

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

```

1992 {0x01 0x00, set, [length placeholder], value={count=3,
1993     {0x01 0x00, array, [length placeholder], value={count=2,
1994         {0x01 0x00, set, [length placeholder], value={count=2,
1995             {0x01 0x00, boolean, [length placeholder], value=true}
1996         {0x01 0x02, enum, [length placeholder], value=2}
1997     }
1998 }

```

```

2000     }}
2001     {0x01 0x02, set, [length placeholder], value={count=1,
2002           {0x01 0x02, enum, [length placeholder], value=0}
2003     }}
2004   }}
2005   {0x01 0x02, string, [length placeholder], value="Dummy ID"}
2006   {0x01 0x06, integer, [length placeholder], value=12}
2007 }}

```

2008 We now encode the formats and the leaf nodes, following Table 9. For sets and arrays, the value
 2009 encoding count prefix is a nonnegative Integer; we can encode that now as well per Table 4. Note the null
 2010 terminator for the string. The encoded sequence numbers for enumeration values do not need a
 2011 dictionary selector inserted as the LSB as the dictionary was already indicated with the sequence number
 2012 for the enumeration itself in the format tuple field.

```

2013
2014 {0x01 0x00, 0x00, [length placeholder], {0x01 0x03,
2015   {0x01 0x00, 0x10, [length placeholder], {0x01 0x02,
2016     {0x01 0x00, 0x00, [length placeholder], {0x01 0x02,
2017       {0x01 0x00, 0x70, [length placeholder], 0xFF}
2018       {0x01 0x02, 0x40, [length placeholder], 0x01 0x02}
2019     }}
2020     {0x01 0x02, 0x00, [length placeholder], {0x01 0x01,
2021       {0x01 0x02, 0x40, [length placeholder], 0x01 0x00}
2022     }}
2023   }}
2024   {0x01 0x02, 0x50, [length placeholder],
2025     0x44 0x75 0x6D 0x6D 0x79 0x20 0x49 0x44 0x00}
2026   {0x01 0x06, 0x30, [length placeholder], 0x0C}
2027 }}

```

2028 All that remains is to fill in the length values. We begin at the leaves:

```

2029
2030 {0x01 0x00, 0x00, [length placeholder], {0x01 0x03,
2031   {0x01 0x00, 0x10, [length placeholder], {0x01 0x02,
2032     {0x01 0x00, 0x00, [length placeholder], {0x01 0x02,
2033       {0x01 0x00, 0x70, 0x01 0x01, 0xFF}
2034       {0x01 0x02, 0x40, 0x01 0x02, 0x01 0x02}
2035     }}
2036     {0x01 0x02, 0x00, [length placeholder], {0x01 0x01,
2037       {0x01 0x02, 0x40, 0x01 0x02, 0x01 0x00}
2038     }}
2039   }}
2040   {0x01 0x02, 0x50, 0x01 0x09,
2041     0x44 0x75 0x6D 0x6D 0x79 0x20 0x49 0x44 0x00}
2042   {0x01 0x06, 0x30, 0x01 0x01, 0x0C}
2043 }}

```

2044 We then work our way from the leaves towards the outermost enclosing tuples. First, the array element
 2045 sets:

```

2046
2047 {0x01 0x00, 0x00, [length placeholder], {0x01 0x03,
2048   {0x01 0x00, 0x10, [length placeholder], {0x01 0x02,
2049     {0x00, 0x00, 0x01 0x0F, {0x01 0x02,
2050       {0x01 0x00, 0x07, 0x01 0x01, 0xFF}
2051       {0x01 0x20, 0x04, 0x01 0x02, 0x01 0x02}
2052     }}
2053     {0x01 0x02, 0x00, 0x01 0x09, {0x01 0x01,
2054       {0x01 0x02, 0x40, 0x01 0x02, 0x01 0x00}

```

```

2055     }}
2056     }}
2057     {0x01 0x02, 0x50, 0x01 0x09,
2058         0x44 0x75 0x6D 0x6D 0x79 0x20 0x49 0x44 0x00}
2059     {0x01 0x06, 0x30, 0x01 0x01, 0x0C}
2060     }}
    
```

Next, the array itself:

```

2061
2062
2063     {0x01 0x00, 0x00, [length placeholder], {0x01 0x03,
2064         {0x01 0x00, 0x10, 0x01 0x24, {0x01 0x02,
2065             {0x01 0x00, 0x00, 0x01 0x0F, {0x01 0x02,
2066                 {0x01 0x00, 0x70, 0x01 0x01, 0xFF}
2067                 {0x01 0x02, 0x40, 0x01 0x02, 0x01 0x02}
2068             }}
2069             {0x01 0x02, 0x00, 0x01 0x09, {0x01 0x01,
2070                 {0x01 0x02, 0x40, 0x01 0x02, 0x01 0x00}
2071             }}
2072         }}
2073     {0x01 0x02, 0x50, 0x01 0x09,
2074         0x44 0x75 0x6D 0x6D 0x79 0x20 0x49 0x44 0x00}
2075     {0x01 0x06, 0x30, 0x01 0x01, 0x0C}
2076     }}
    
```

Finally, the outermost set:

```

2077
2078
2079     {0x01 0x00, 0x00, 0x01 0x3F, {0x01 0x03,
2080         {0x01 0x00, 0x10, 0x01 0x24, {0x01 0x02,
2081             {0x01 0x00, 0x00, 0x01 0x0F, {0x01 0x02,
2082                 {0x01 0x00, 0x70, 0x01 0x01, 0xFF}
2083                 {0x01 0x02, 0x40, 0x01 0x02, 0x01 0x02}
2084             }}
2085             {0x01 0x02, 0x00, 0x01 0x09, {0x01 0x01,
2086                 {0x01 0x02, 0x40, 0x01 0x02, 0x01 0x00}
2087             }}
2088         }}
2089     {0x01 0x02, 0x50, 0x01 0x09,
2090         0x44 0x75 0x6D 0x6D 0x79 0x20 0x49 0x44 0x00}
2091     {0x01 0x06, 0x30, 0x01 0x01, 0x0C}
2092     }}
    
```

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

```

2093
2094
2095     0x01 0x00 0x00 0x01 : 0x3F 0x01 0x03 0x01
2096     0x00 0x10 0x01 0x24 : 0x01 0x02 0x01 0x00
2097     0x00 0x01 0x0F 0x01 : 0x02 0x01 0x00 0x70
2098     0x01 0x01 0xFF 0x01 : 0x02 0x40 0x01 0x02
2099     0x01 0x02 0x01 0x02 : 0x00 0x01 0x09 0x01
2100     0x01 0x01 0x02 0x40 : 0x01 0x02 0x01 0x00
2101     0x01 0x02 0x50 0x01 : 0x09 0x44 0x75 0x6D
2102     0x6D 0x79 0x20 0x49 : 0x44 0x00 0x01 0x06
2103     0x30 0x01 0x01 0x0C
    
```

8.6.3 Example decoding

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

```

2105
2106
2107
2108     0x01 0x00 0x00 0x01 : 0x3F 0x01 0x03 0x01
2109     0x00 0x10 0x01 0x24 : 0x01 0x02 0x01 0x00
    
```

```

2110 0x00 0x01 0x0F 0x01 : 0x02 0x01 0x00 0x70
2111 0x01 0x01 0xFF 0x01 : 0x02 0x40 0x01 0x02
2112 0x01 0x02 0x01 0x02 : 0x00 0x01 0x09 0x01
2113 0x01 0x01 0x02 0x40 : 0x01 0x02 0x01 0x00
2114 0x01 0x02 0x50 0x01 : 0x09 0x44 0x75 0x6D
2115 0x6D 0x79 0x20 0x49 : 0x44 0x00 0x01 0x06
2116 0x30 0x01 0x01 0x0C

```

2117 The first step of the decoding process is to map the byte data to {SFLV} tuples, using the length bytes and
 2118 set/array counts to identify tuple boundaries:

```

2119
2120 {S=0x01 0x00, F=0x00, L=0x01 0x3F, V={0x01 0x03,
2121   {S=0x01 0x00, F=0x10, L=0x01 0x24, V={0x01 0x02,
2122     {S=0x01 0x00, F=0x00, L=0x01 0x0F, V={0x01 0x02,
2123       {S=0x01 0x00, F=0x70, L=0x01 0x01, V=0xFF}
2124         {S=0x01 0x02, F=0x40, L=0x01 0x02, V=0x01 0x02}
2125           }}
2126     {S=0x01 0x02, F=0x00, L=0x01 0x09, V={0x01 0x01,
2127       {S=0x01 0x02, F=0x40, L=0x01 0x02, V=0x01 0x00}
2128         }}
2129     }}
2130 {S=0x01 0x02, F=0x50, L=0x01 0x09,
2131   V=0x44 0x75 0x6D 0x6D 0x79 0x20 0x49 0x44 0x00}
2132 {0x01 S=0x06, F=0x30, L=0x01 0x01, V=0x0C}
2133 }}

```

2134 After the tuple boundaries are understood, the length and count data are no longer needed:

```

2135
2136 {S=0x01 0x00, F=0x00, V={
2137   {S=0x01 0x00, F=0x10, V={
2138     {S=0x01 0x00, F=0x00, V={
2139       {S=0x01 0x00, F=0x70, V=0xFF}
2140       {S=0x01 0x02, F=0x40, V=0x01 0x02}
2141     }}
2142     {S=0x01 0x02, F=0x00, V={
2143       {S=0x01 0x02, F=0x40, V=0x01 0x00}
2144     }}
2145   }}
2146 {S=0x01 0x02, F=0x50, V=0x44 0x75 0x6D 0x6D 0x79 0x20 0x49 0x44 0x00}
2147 {S=0x01 0x06, F=0x30, V=0x0C}
2148 }}

```

2149 The next step is to decode format tuple bytes using Table 9. This will tell us how to decode the value
 2150 data:

```

2151
2152 {S=0x01 0x00, set, V={
2153   {S=0x01 0x00, array, V={
2154     {S=0x01 0x00, set, V={
2155       {S=0x01 0x00, boolean, V=0xFF}
2156       {S=0x01 0x02, enum, V=0x01 0x02}
2157     }}
2158     {S=0x01 0x02, set, V={
2159       {S=0x01 0x02, enum, V=0x01 0x00}
2160     }}
2161   }}
2162 {S=0x01 0x02, string, V=0x44 0x75 0x6D 0x6D 0x79 0x20 0x49 0x44 0x00}
2163 {S=0x01 0x06, integer, V=0x0C}
2164 }}

```

2165 We now decode value data:

2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179

```
{S=0x01 0x00, set, {
  {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}
}}
```

2180 Next we decode the sequence numbers to identify which dictionary they select:

2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194

```
{S=major/0, set, {
  {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}
}}
```

2195 Next we use the selected dictionary to replace decoded sequence numbers with the strings they represent:

2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210

```
{"DummySimple", set, {
  {"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}
}}
```

2211 We can now write out the decoded BEJ data in JSON format if desired (an MC will need to do this to forward an RDE Device's response to a client, but an RDE Device may not need this step):

2212
2213
2214
2215
2216
2217
2218
2219
2220
2221

```
{
  "DummySimple" : {
    "ChildArrayProperty" : [
      {
        "AnotherBoolean" : true,
        "LinkStatus" : "NoLink"
      },
      {
```

```

2222         "LinkStatus" : "LinkDown"
2223     }
2224 },
2225     "Id" : "Dummy ID",
2226     "SampleIntegerProperty" : 12
2227 }
2228 }

```

2229 8.7 BEJ locators

2230 A BEJ locator represents a particular location within a resource at which some operation is to take place.
 2231 The locator itself consists of a list of sequence numbers for the series of nodes representing the traversal
 2232 from the root of the schema tree down to the point of interest. The list of schema nodes is concatenated
 2233 together to form the locator. A locator with no sequence numbers targets the root of the schema.

2234 **NOTE** The sequence numbers are absolute as they are relative to the schema, not to the subset of the schema for
 2235 which the RDE Device supports data. This enables a locator to be unambiguous.

2236 As an example, consider a locator, encoded for the example dictionary of clause 8.6.1:

```
2237     0x01 0x08 0x01 0x00 0x01 0x00 0x01 0x06 0x01 0x02
```

2238 Decoding this locator, begins with decoding the length in bytes of the locator. In this case, the first two
 2239 bytes specify that the remainder of the locator is 8 bytes long. The next step is to decode the bejTupleS-
 2240 formatted sequence numbers. The low-order bit of each sequence number references the schema to
 2241 which it refers; in this case, the pattern 0b indicates the major schema. Decoding produces the following
 2242 list:

```
2243     0, 0, 3, 1
```

2244 Now, referring to the dictionary enables identification of the target location. Remember that all indices are
 2245 zero-based:

- 2246 • The first zero points to DummySimple
- 2247 • The second zero points to the first child of DummySimple, or ChildArrayProperty
- 2248 • The three points to the fourth element in the ChildArrayProperty array, an anonymous instance
 2249 of the array type (array instances are not reflected in the dictionary, but are implicitly the
 2250 immediate children of any array)
- 2251 • The one points to the second child inside the ChildArray element type, or LinkStatus

2252 9 Operational behaviors

2253 This clause describes the operational behavior for initialization, Operations/Tasks, and Events.

2254 9.1 Initialization (MC perspective)

2255 The following clauses present initialization of RDE Devices with MCs.

2256 9.1.1 Sample initialization ladder diagram

2257 Figure 7 presents the ladder diagram for an example initialization sequence.

2258 Once the MC detects the RDE Device, it begins the discovery process by invoking the
 2259 NegotiateRedfishParameters command to determine the concurrency and feature support for the RDE

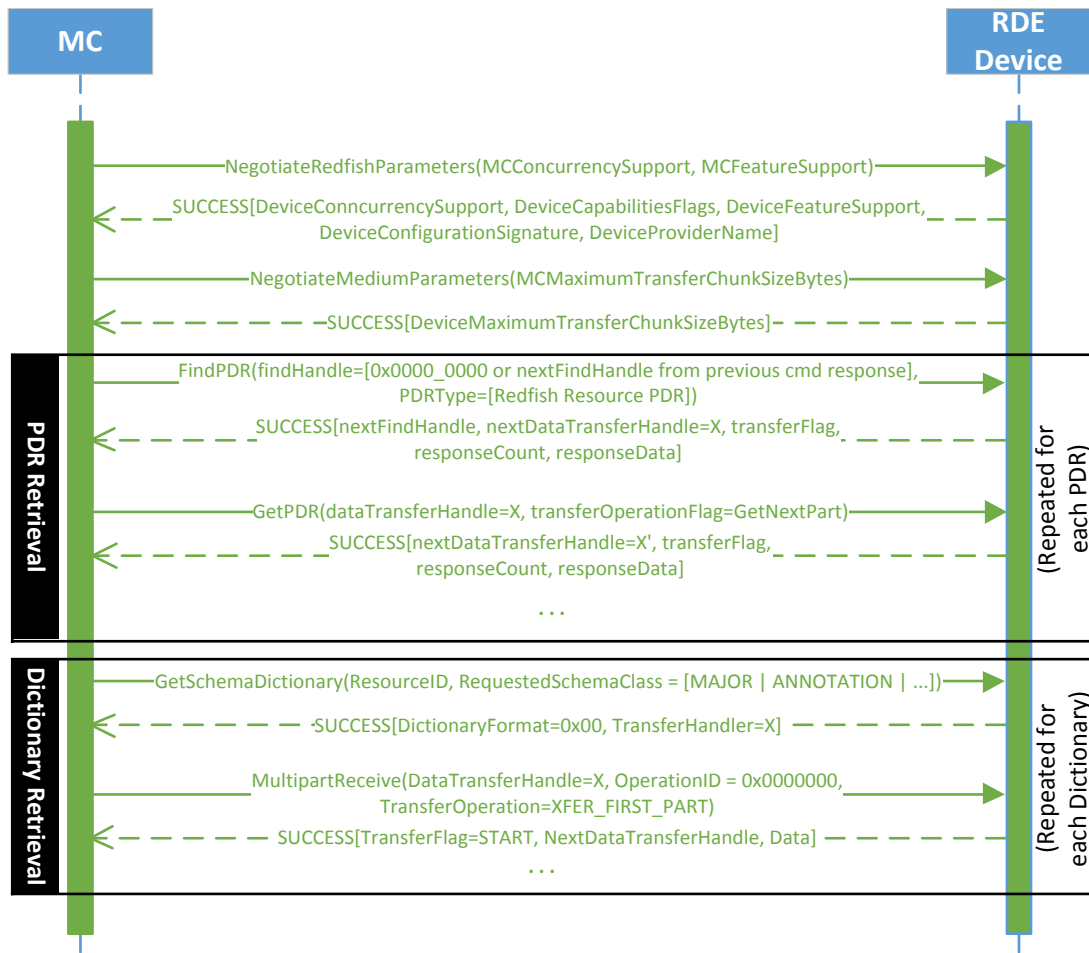
2260 Device. It then uses the NegotiateMediumParameters command to determine the maximum message
2261 size that the MC and the RDE Device can both support. This finishes the RDE discovery process.

2262 After discovery comes the RDE registration process. It consists of two parts, PDR retrieval and dictionary
2263 retrieval. To retrieve the RDE PDRs, the MC utilizes the PLDM for Platform Monitoring and Control
2264 FindPDR command to locate PDRs that are specific to RDE⁴. For each such PDR located, the MC then
2265 retrieves it via one or more message sequences in the PLDM for Platform Monitoring and Control
2266 GetPDR command.

2267 After all the PDRs are retrieved, the next step is to retrieve dictionaries. For each Redfish Resource PDR
2268 that the MC retrieved, it retrieves the relevant dictionaries via a standardized process in which it first
2269 executes the GetSchemaDictionary command to obtain a transfer handle for the dictionary. It then uses
2270 the transfer handle with the MultipartReceive command to retrieve the corresponding dictionary.

2271 Multiple initialization variants are possible; for example, it is conceivable that retrieval of some or all
2272 dictionaries could be postponed until such time as the MC needs to translate BEJ and/or JSON code for
2273 the relevant schema. Further, the MC may be able to determine that of the dictionaries it has already
2274 retrieved is adequate to support a PDR and thus skip retrieving that dictionary anew. Finally, if the
2275 DeviceConfigurationSignature from the NegotiateRedfishParameters command matches the one for data
2276 that the MC has already cached for the RDE Device, it may elide 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.



2277

2278

Figure 7 – Example Initialization ladder diagram

2279 **9.1.2 Initialization workflow diagram**

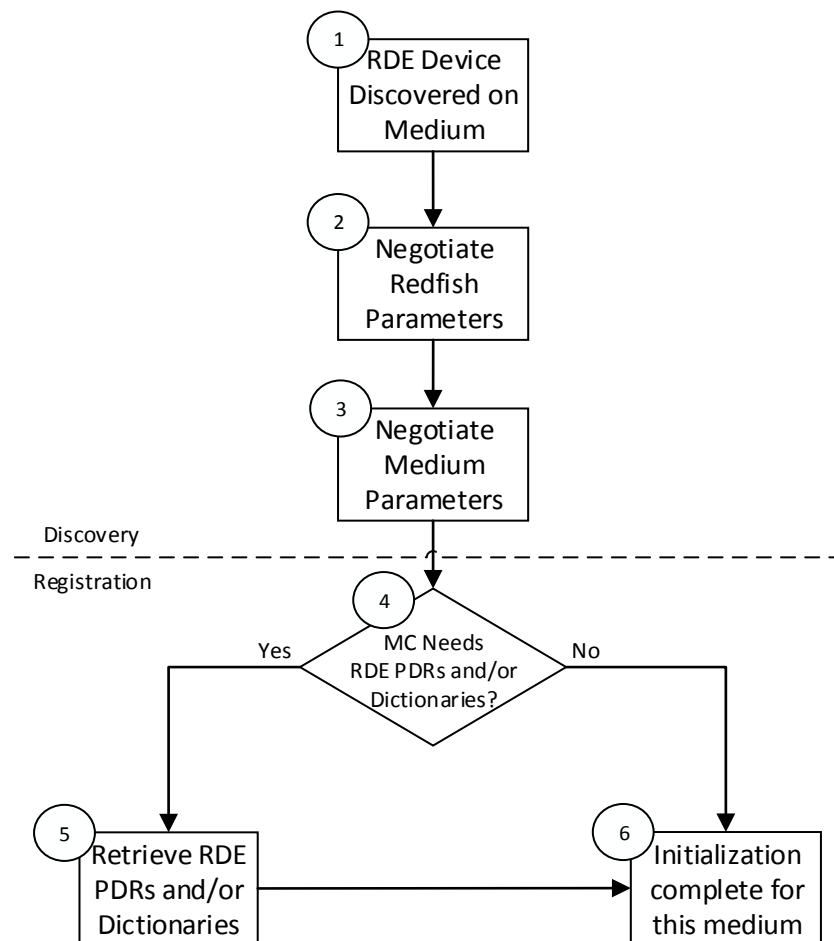
2280 Table 42 details the information presented visually in Figure 8.

2281

Table 42 – 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

Step	Description	Condition	Next Step
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



2282

2283

Figure 8 – Typical RDE Device discovery and registration

2284 **9.2 Operation/Task lifecycle**

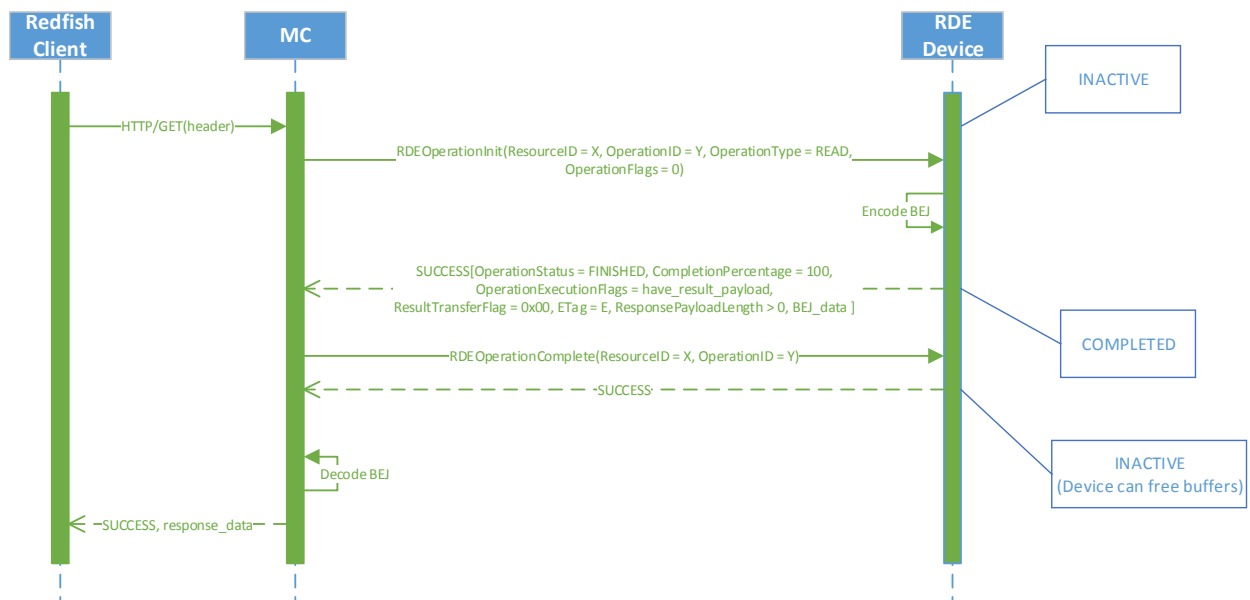
2285 The following clauses present the Task lifecycle from two perspectives, first from an Operation-centric
 2286 viewpoint and then from the RDE Device perspective. MC and RDE Device implementations of RDE shall
 2287 comply with the sequences presented here.

2288 **9.2.1 Example Operation command sequence diagrams**

2289 This clause presents request/response messaging sequences for common Operations.

2290 **9.2.1.1 Simple read Operation ladder diagram**

2291 Figure 9 presents the ladder diagram for a simple read Operation. The Operation begins when the
 2292 Redfish client sends a GET request over an HTTP connection to the MC. The MC decodes the URI
 2293 targeted by the GET operation to pin it down to a specific resource and PDR and sends the
 2294 RDEOperationInit command to the RDE Device that owns the PDR, with OperationType set to READ.
 2295 The RDE Device now has everything it needs for the Operation, so it performs a BEJ encoding of the
 2296 schema data for the requested resource and sends it as an inlined payload back to the MC. Sending
 2297 inline is possible in this case because the read data is small enough to not cause the response message
 2298 to exceed the maximum transfer size that was previously negotiated in the NegotiateMediumParameters
 2299 command. The MC in turn has all of the results for the Operation, so it sends RDEOperationComplete to
 2300 finalize the Operation. The RDE Device can now throw away the BEJ encoded read result, and responds
 2301 to the MC with success. Finally, the MC uses the dictionary it previously retrieved from the RDE Device to
 2302 decode the BEJ payload for the read command into JSON data and the MC sends the JSON data back to
 2303 the client.



2304
2305

2306 **Figure 9 – Simple read Operation ladder diagram**

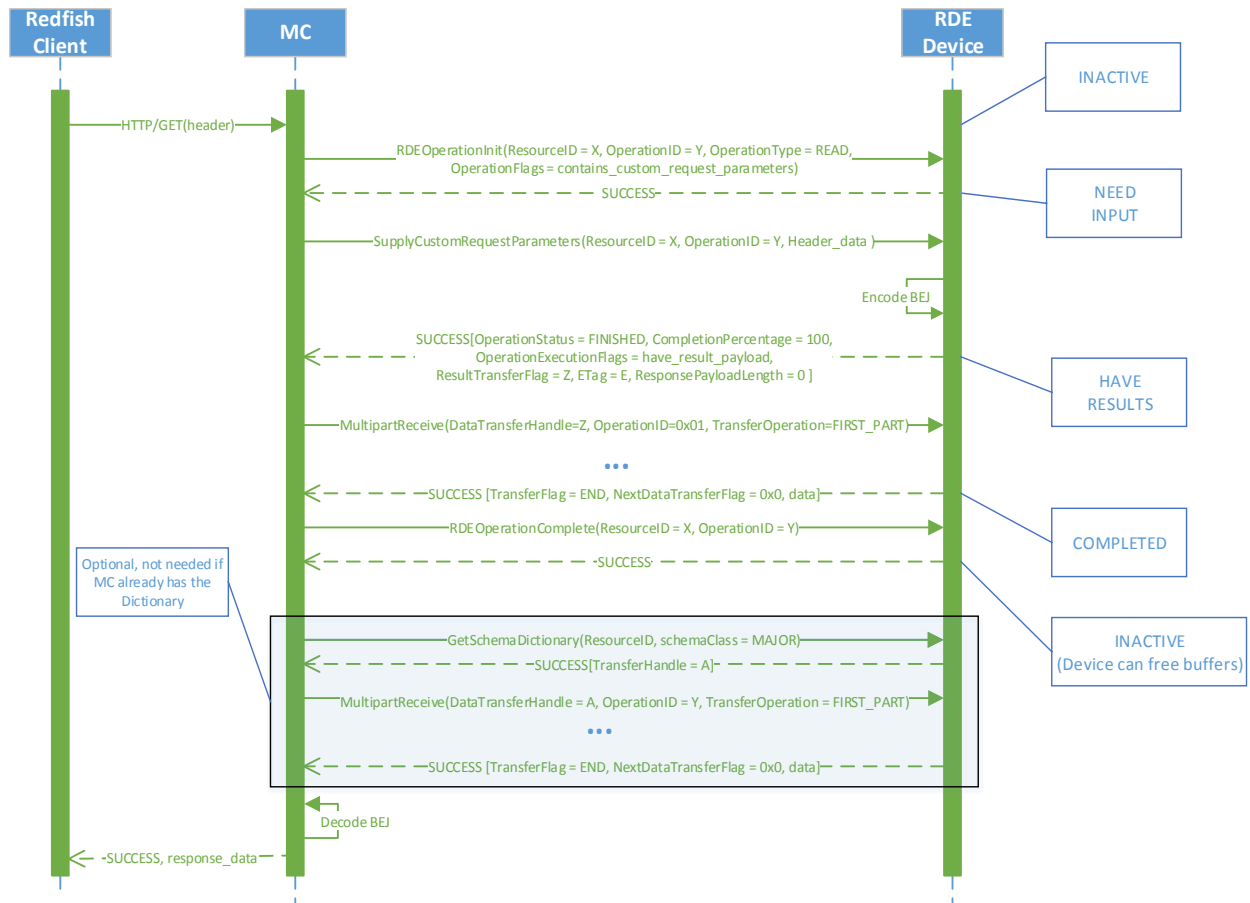
2307 **9.2.1.2 Complex read Operation diagram**

2308 Figure 10 presents the ladder diagram for a more complex read Operation. As with the simple read case,
 2309 the Operation begins when the Redfish client sends a GET request over an HTTP connection to the MC.

2310 The MC again decodes the URI targeted by the GET operation to pin it down to a specific resource and
2311 PDR and sends the RDEOperationInit command to the RDE Device that owns the PDR, with
2312 OperationType set to READ. In this case, however, the OperationFlags that the MC sent with the
2313 RDEOperationInit command indicate that there are supplemental parameters to be sent to the RDE
2314 Device, so the RDE Device must wait for these before beginning work on the Operation. The MC sends
2315 these supplemental parameters to the RDE Device via the SupplyCustomRequestParameters command.

2316 At this point, the RDE Device has everything it needs for the Operation, so just as before, the RDE
2317 Device performs a BEJ encoding of the schema data for the requested resource. As opposed to the
2318 previous example, in this case the BEJ-encoded payload is too large to fit within the response message,
2319 so the RDE Device instead supplied a transfer handle that the MC can use to retrieve the BEJ payload
2320 separately. The MC, seeing this, performs a series of MultipartReceive commands to retrieve the payload.
2321 Once it is all transferred, the MC has everything it needs. If it needs a dictionary to decode the BEJ
2322 payload, it may retrieve one via the GetSchemaDictionary command followed by one or more
2323 MultipartReceive commands to retrieve the binary dictionary data. (Normally, the MC would have
2324 retrieved the dictionary during initialization; however, if the MC has limited storage space to cache
2325 dictionaries, it may have been forced to evict it.) Whether it needed to retrieve a dictionary or it already
2326 had one, the MC now sends the RDEOperationComplete command to finalize the Operation and allow
2327 the RDE Device to throw away the BEJ encoded read result. Finally, the MC uses the dictionary to
2328 decode the BEJ payload for the read command into JSON data and then the MC sends the JSON data
2329 back to the client.

2330



2331
2332

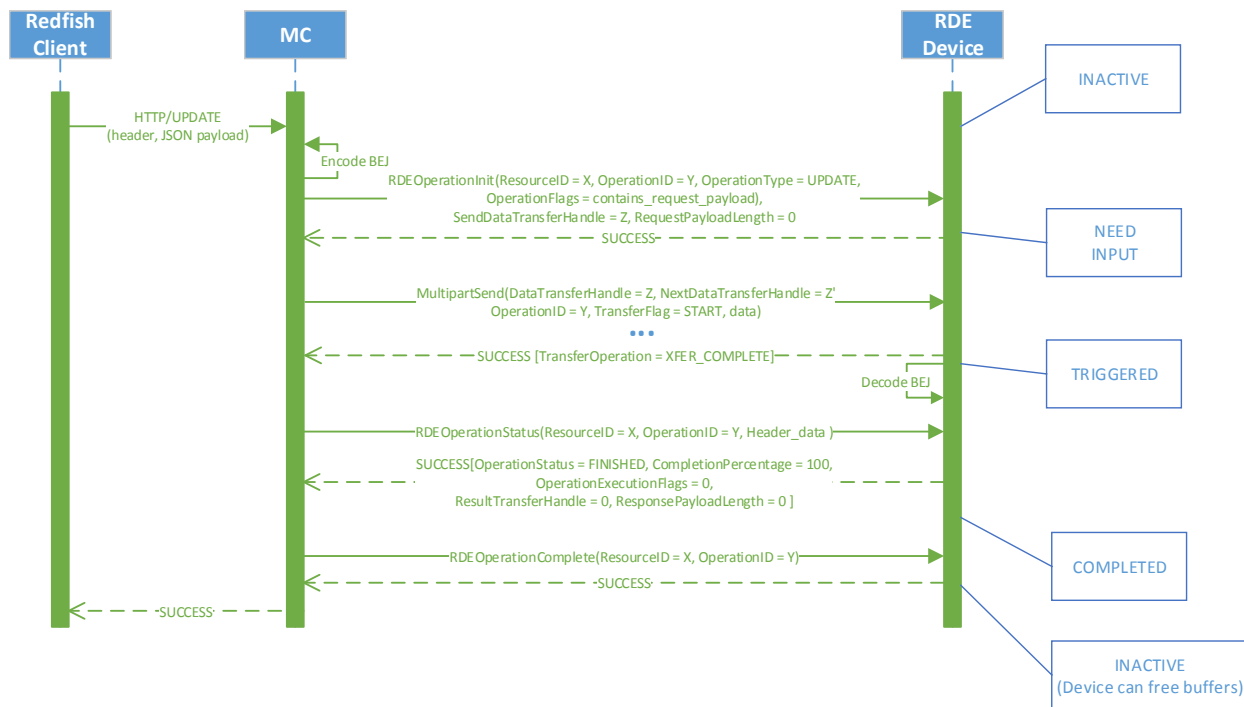
2333 **Figure 10 – Complex Read Operation ladder diagram**

2334 **9.2.1.3 Write (update) Operation ladder diagram**

2335 Figure 11 presents the ladder diagram for a write Operation. As with the read cases, the Operation begins
 2336 when the Redfish client sends a request over an HTTP connection to the MC, in this case, an UPDATE.
 2337 Once again, the MC decodes the URI targeted by the UPDATE Operation to pin it down to a specific
 2338 resource and PDR. Before it can send the RDEOperationInit command to the RDE Device that owns the
 2339 PDR, however, the MC must perform a BEJ encoding of the JSON payload it received from the Redfish
 2340 client. If the BEJ encoded payload were small enough to fit within the maximum transfer chunk, the MC
 2341 could inline it with the RDEOperationInit command; however, in this example, that is not the case. The
 2342 MC therefore sends RDEOperationInit with the OperationType set to UPDATE and a nonzero transfer
 2343 handle. Seeing this, the RDE Device knows to expect a larger payload via MultipartSend.

2344 The MC uses the MultipartSend command to transfer the encoded payload to the RDE Device in one or
 2345 more chunks. The contains_request_parameters Operation flag is not set, so the RDE Device will not
 2346 expect supplemental parameters as part of this Operation. Having everything it needs to execute, the
 2347 RDE Device moves to the TRIGGERED state. The MC now sends the RDEOperationStatus command to
 2348 the RDE Device to have it execute the Operation. (In practice, the RDE Device is allowed to begin
 2349 executing the Operation as soon as it has received the request payload, so it may choose not to wait for
 2350 the RDEOperationStatus command to do so.) The RDE Device executes the Operation and sends the

2351 results to the MC as the response to the RDEOperationStatus command. As before, the MC finalizes the
 2352 Operation via RDEOperationComplete and then sends the results back to the client.



2353
 2354

2355 **Figure 11 – Write Operation ladder diagram**

2356 **9.2.1.4 Write (update) with Long-running Task Operation Ladder Diagram**

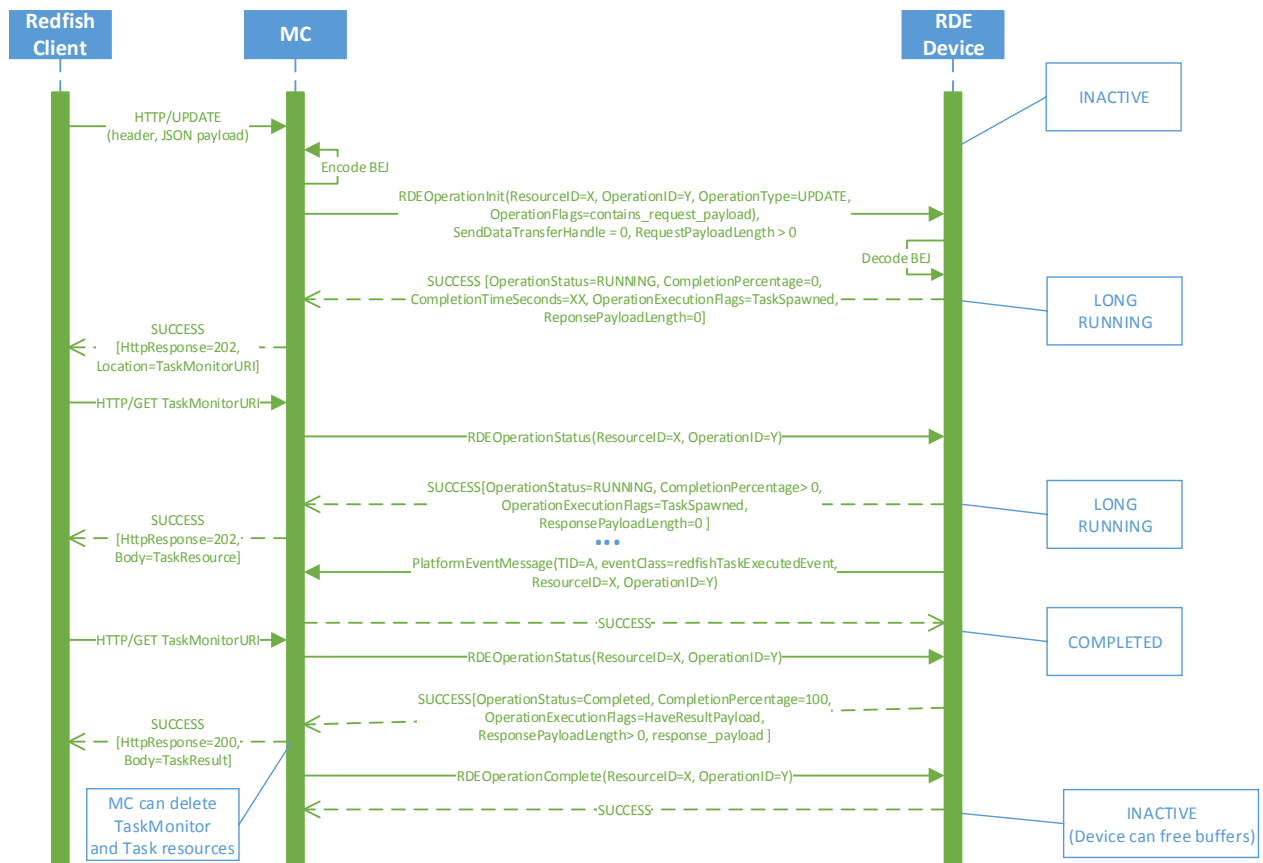
2357 Figure 12 presents the ladder diagram for a write Operation that spawns a long-running Task. As with the
 2358 previous case, the Operation begins when the Redfish client sends an UPDATE request over an HTTP
 2359 connection to the MC, and the MC decodes the URI targeted by the UPDATE Operation to pin it down to
 2360 a specific resource and PDR. Before it can send the RDEOperationInit command to the RDE Device that
 2361 owns the PDR, however, the MC must perform a BEJ encoding of the JSON payload it received from the
 2362 Redfish client. Unlike the previous example, the BEJ encoded payload here is small enough to fit in the
 2363 maximum transfer chunk, so the MC inlines it into the RDEOperationInit request command. Again, the
 2364 contains_request_parameters Operation flag is not set, so the RDE Device will not expect supplemental
 2365 parameters as part of this Operation.

2366 When the RDE Device receives the RDEOperationInit request command, it has everything it needs to
 2367 begin work on the Operation. In this case, the RDE Device determines that performing the write will take
 2368 longer than PT1, so the RDE Device spawns a long-running Task to process the write asynchronously
 2369 and sends TaskSpawned in the OperationExecutionFlags to inform the MC.

2370 When it discovers that the RDE Device spawned a long-running Task, the MC adds a member to the
 2371 Task collection it maintains and synthesizes a TaskMonitor URI to send back to the client in a location
 2372 response header. At this point, the client can issue an HTTP GET to retrieve a status update on the Task;
 2373 when it does so, the MC sends RDEOperationStatus to the RDE Device to get the status update and
 2374 sends it back to the client as the result of the GET operation.

2375 At some point, the asynchronous Task finishes executing. When this happens, the RDE Device issues a
 2376 PlatformEventMessage to send a TaskCompletion event to the MC. (This presupposes that the RDE
 2377 Device and the MC both support asynchronous eventing. Were this not the case, the RDE Device would

2378 still generate the TaskCompletion event, but would wait for the MC to invoke the
 2379 PollForPlatformEventMessage command to report the event.) Regardless of which way the MC gets the
 2380 event, it then sends the RDEOperationStatus command one last time in order to retrieve the final results
 2381 from the Operation. The next time the client performs a GET on the TaskMonitor, the MC can send back
 2382 the final results of the Operation. Finally, the MC finalizes the Operation via RDEOperationComplete at
 2383 which point the MC can delete the Task collection member and the TaskMonitor URI and the RDE Device
 2384 can free up any buffers associated with the Operation and/or Task.
 2385



2386
2387

2388 **Figure 12 – Write Operation with long-running Task ladder diagram**

2389 **9.2.2 Operation/Task overview workflow diagrams (Operation perspective)**

2390 This clause describes the operating behavior for MCs and RDE Devices over the lifecycle of Operations
 2391 from an Operation-centric perspective. The workflow diagrams are split between simpler, short-lived
 2392 Operations and those that spawn a Task to be processed asynchronously. These workflow diagrams are
 2393 intended to capture the standard flow for the execution of most Operations, but do not cover every
 2394 possible error condition. For full precision, refer to clause 9.2.3.

2395 **9.2.2.1 Operation overview workflow diagram**

2396 Table 43 details the information presented visually in Figure 13.

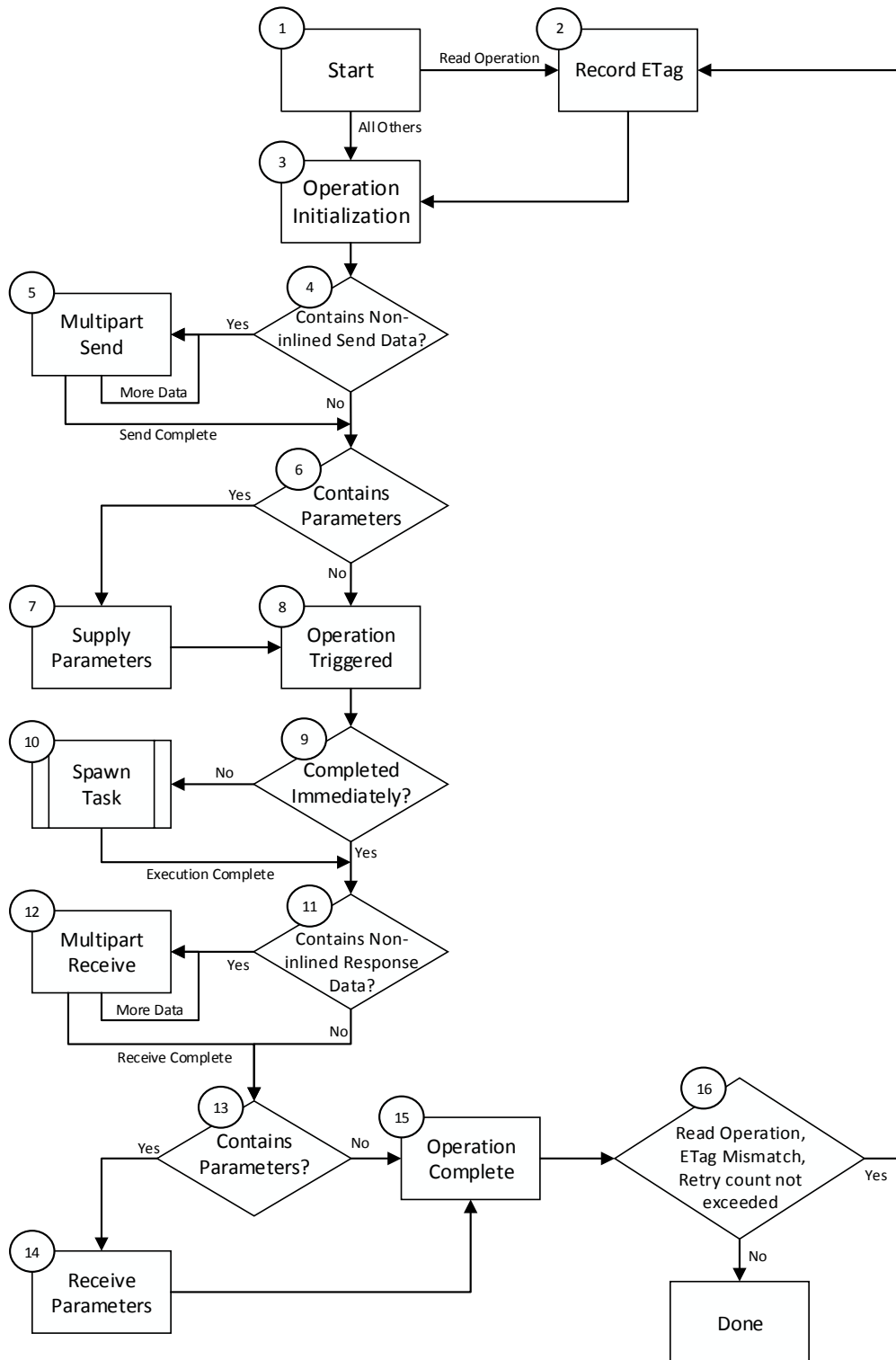
2397

Table 43 – 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 MultipartSend 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. NOTE The transfer of a noninlined request payload and supplemental request parameters may be performed in either order. For simplicity, the flow shown assumes that a payload would be transferred before supplemental request parameters; however, the opposite assumption could be made by swapping the positions of blocks 4/5 with blocks 6/7 in the figure.	If the Operation contains supplemental request parameters	7
		Otherwise	8
7 – SEND_PARAMS	The MC uses the SupplyCustomRequestParameters command to submit the	Unconditional	8

Step	Description	Condition	Next Step
	supplemental request parameters to the RDE Device		
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 RDEOperationStatus if the last data was sent via MultipartSend) within PT1 time. If it can complete the Operation within that timeframe, it does not need to spawn a Task to run the Operation asynchronously.	If the RDE Device is able to complete the Operation “quickly”	11
		Otherwise	10
10 – LONG_RUN	If the RDE Device was not able to complete the Operation quickly enough it spawns a Task to execute asynchronously. See Figure 14 for details of the Task sublifecyle.	Once the Task finishes executing	11
11 – RCV_PAYLOAD_CHK	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 MultipartSend 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 MultipartReceive 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 –	The MC checks to see if the	If the Operation contains response	14

Step	Description	Condition	Next Step
RCV_PARAMS_CHK	Operation result contains supplemental response parameters	parameters	
		Otherwise	15
14 – RCV_PARAMS	<p>The MC uses the RetrieveCustomResponseParameters command to obtain the supplemental response parameters.</p> <p>NOTE The transfer of a noninlined response payload and supplemental response parameters may be performed in either order. For simplicity, the flow shown assumes that a response payload would be transferred before supplemental response parameters; however, the opposite assumption could be made by swapping the positions of blocks 11/12 with blocks 13/14 in the figure.</p>	Unconditional	15
15 – COMPLETE	The MC sends the RDEOperationComplete command to finalize the Operation	n/a	n/a
16 – CMP_DIGEST	If the Operation was a read and the MC collected an ETag in step 2, the MC compares the response ETag with the one it collected in step 2 to check for a consistency violation. If it finds one, it may retry the operation or give up. The MC may skip the consistency check (treat it as successful without checking) if the RDE Device advertised that it has the capability to read a resource atomically in its response to the NegotiateRedfishParameters command (see clause 11.1).	Read operation and mismatched ETags and retry count not exceeded	2
		Not a read, no ETag collected, the ETags match, or retry count exceeded	n/a: Done



2398

2399

Figure 13 – RDE Operation lifecycle overview (holistic perspective)

2400 9.2.2.2 Task overview workflow diagram

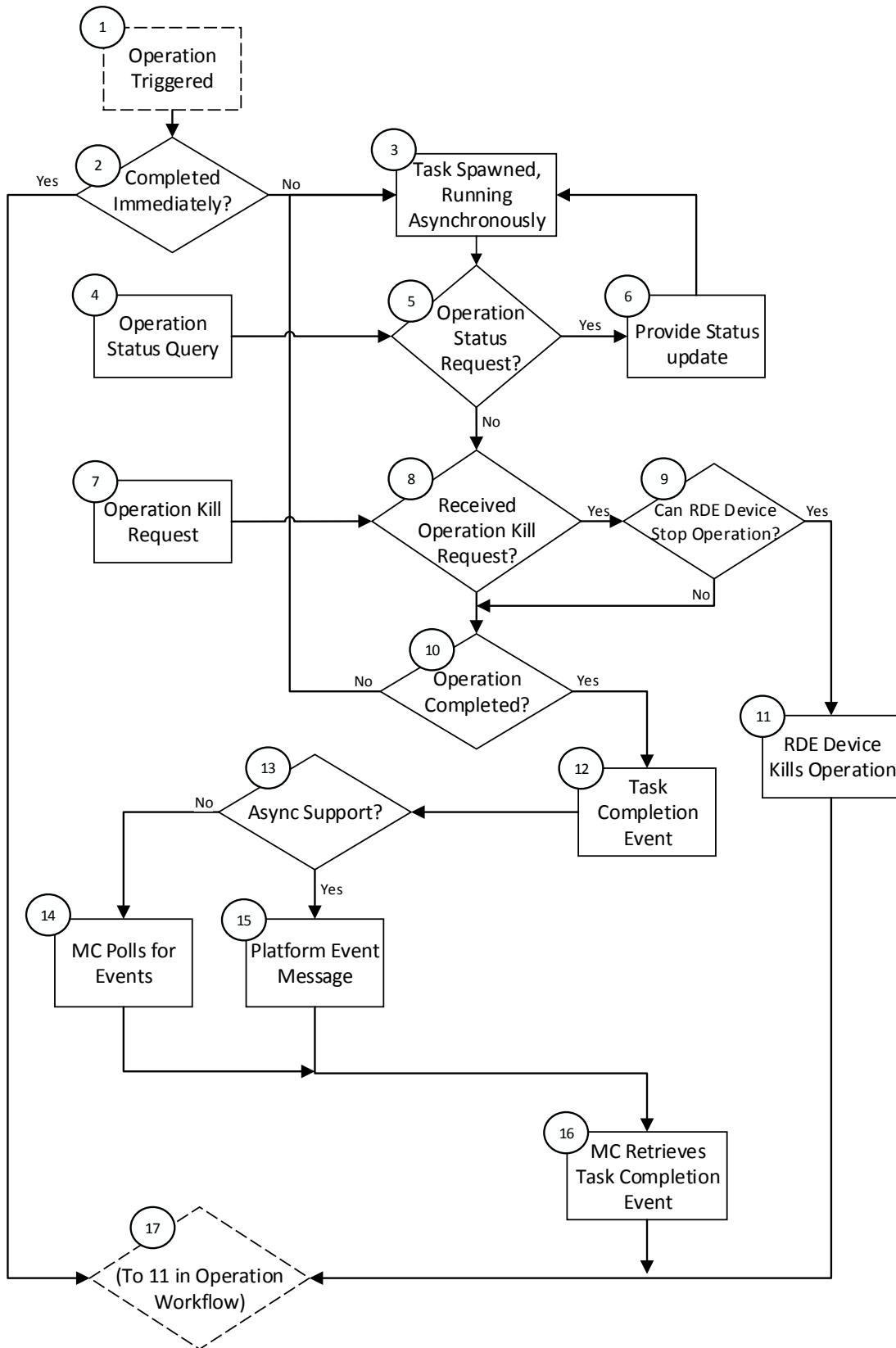
2401 Table 44 details the information presented visually in Figure 14.

2402 **Table 44 – 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 43.)	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 43.)	See Step 11 in Table 45	See Step 11 in Table 45

2403



2404

2405

Figure 14 – RDE Task lifecycle overview (holistic perspective)

2406 9.2.3 RDE Operation state machine (RDE Device perspective)

2407 The following clauses describe the operating behavior for the lifecycle of Operations and Tasks from an
2408 RDE Device-centric perspective. Table 45 details the information presented visually in Figure 15. The
2409 states presented in this state machine are not the total state for the RDE Device, but rather the state for
2410 the Operation. The total state for the RDE Device would involve separate instances of the Task/Operation
2411 state machine replicated once for each of the concurrent Operations that the RDE Device and the MC
2412 negotiated to support at registration time.

2413 9.2.3.1 State definitions

2414 The following states shall be implemented by the RDE Device for each Operation it is supporting.

- 2415 • INACTIVE
 - 2416 ○ INACTIVE is the default Operation state in which the RDE Device shall start after
 - 2417 initialization. In this state, the RDE Device is not processing an Operation as it has not
 - 2418 received an RDEOperationInit command from the MC
- 2419 • NEED_INPUT
 - 2420 ○ After receiving the RDEOperationInit command, the RDE Device moves to this state if it
 - 2421 is expecting additional Operation-specific parameters or a payload that was not inlined in
 - 2422 the RDEOperationInit command
- 2423 • TRIGGERED
 - 2424 ○ Once the RDE Device receives everything it needs to execute an Operation, it begins
 - 2425 executing it immediately. If the triggering command – the command that supplied the last
 - 2426 bit of data needed to execute the Operation – was RDEOperationInit or
 - 2427 SupplyCustomRequestParameters, the response message to the triggering command
 - 2428 reflects the initial results for the Operation. However, if the triggering command was a
 - 2429 MultipartSend, initial results are deferred until the MC invokes the RDEOperationStatus
 - 2430 command. This state captures the case where the Operation was triggered by a
 - 2431 MultipartSend and the MC has not yet sent an RDEOperationStatus command to get
 - 2432 initial results. In this state, the RDE Device may execute the Operation; alternatively, it
 - 2433 may wait to receive RDEOperationStatus to begin execution.
- 2434 • TASK_RUNNING
 - 2435 ○ If the RDE Device cannot complete the Operation within the timeframe needed for the
 - 2436 response to the command that triggered it, the RDE Device spawns a Task in which to
 - 2437 execute the Operation asynchronously
- 2438 • HAVE_RESULTS
 - 2439 ○ When execution of the Operation produces a response parameters or a response
 - 2440 payload that does not fit in the response message for the command that triggered the
 - 2441 Operation (or detected its completion, if a Task was spawned or if there was a payload
 - 2442 but no custom request parameters), the RDE Device remains in this state until the MC
 - 2443 has collected all of these results
- 2444 • COMPLETED
 - 2445 ○ The RDE Device has completed processing of the Operation and awaits
 - 2446 acknowledgment from the MC that it has received any Operation response data. This
 - 2447 acknowledgment is done by the MC issuing the RDEOperationComplete command.
 - 2448 When the RDE Device receives this command, it may discard any internal records or
 - 2449 state it has maintained for the Operation
- 2450 • ABANDONED
 - 2451 ○ If MC fails to progress the Operation through this state machine, the RDE Device may
 - 2452 abort the Operation and mark it as abandoned
- 2453 • FAILED
 - 2454 ○ The MC has explicitly killed the Operation or an error prevented execution of the
 - 2455 Operation

2456 **9.2.3.2 Operation lifecycle state machine**

2457 Figure 15 illustrates the state transitions the RDE Device shall implement. Each bubble represents a
 2458 particular state as defined in the previous clause. Upon initialization, system reboot, or an RDE Device
 2459 reset the RDE Device shall enter the INACTIVE state.

2460 **Table 45 – 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, request contains any other error	Various, depending on the specific error encountered	INACTIVE
	RDEOperationStatus	OPERATION_INACTIVE	INACTIVE
	RDEOperationInit; - valid request - Operation Flags indicate request non-inlined payload or parameters to be sent from MC to RDE Device	Success	NEED_INPUT
	RDEOperationInit;	Success	TASK_RUNNING

Current State	Trigger	Response	Next State
	<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 		
	<p>RDEOperationInit;</p> <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
	<p>RDEOperationInit;</p> <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 - 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 	Success	COMPLETED

Current State	Trigger	Response	Next State
	- no response parameters		
	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 MultipartSend/SupplyCustomRequestParameterscommand	None	ABANDONED
	RDEOperationKill; - neither run_to_completion nor discard_record flag set	Success	FAILED
	RDEOperationKill; - run_to_completion flag not set - discard_record flag set	Success	INACTIVE
	RDEOperationKill; - 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)	FAILED
	RDEOperationStatus	OPERATION_NEED_INPUT	NEED_INPUT
	MultipartSend; - data inlined or Operation flags indicate no payload data	ERROR_UNEXPECTED	FAILED
	MultipartSend; - transfer error	Error specific to type of transfer failure encountered	NEED_INPUT (MC may retry send or use RDEOperationKill to abort Operation)
	MultipartSend; - more data to be sent from the MC to the RDE Device after this chunk	Success	NEED_INPUT
	MultipartSend; - no more data to be sent from the MC to the RDE Device after this chunk - RDEOperationInit request	Success	NEED_INPUT

Current State	Trigger	Response	Next State
	flags indicated supplemental parameters needed - params not yet sent		
	MultipartSend; - no more data to be sent after this chunk - RDEOperationInit request flags indicated supplemental parameters not needed or parameters already sent	Success	TRIGGERED
	MultipartSend; - data already transferred	ERROR_UNEXPECTED	FAILED
	SupplyCustomRequestParameters; - Operation flags indicated supplemental parameters not needed or payload data remaining to be sent	ERROR_UNEXPECTED	FAILED
	SupplyCustomRequestParameters; - no payload data remaining to be sent - ETagOperation is ETAG_IF_MATCH and no ETag matches or ETagOperation is ETAG_IF_NONE_MATCH and an ETAG matches	ERROR_ETAG_MATCH	FAILED
	SupplyCustomRequestParameters; - request contains unsupported 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	Success	HAVE_RESULTS

Current State	Trigger	Response	Next State
	<ul style="list-style-type: none"> - response flags indicate response parameters or a non-inlined response payload to be retrieved from RDE Device 		
	SupplyCustomRequestParameters; <ul style="list-style-type: none"> - 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
	MultipartReceive, RDEOperationComplete	ERROR_UNEXPECTED	FAILED
	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 RDE Device has another slot to execute an Operation	The new Operation is tracked in a separate copy of the state machine; this Operation remains in TRIGGERED
	T _{abandon} timeout waiting for RDEOperationStatus command	None	ABANDONED
	RDEOperationStatus; error occurs in processing of Operation	Error specific to type of failure encountered	FAILED
	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 set - discard_record flag not set 	Success	INACTIVE
	RDEOperationKill <ul style="list-style-type: none"> - Operation executing - Operation can be killed 	ERROR_UNEXPECTED (can't run to completion without further input from MC to move it to TASK_RUNNING, so the request	FAILED

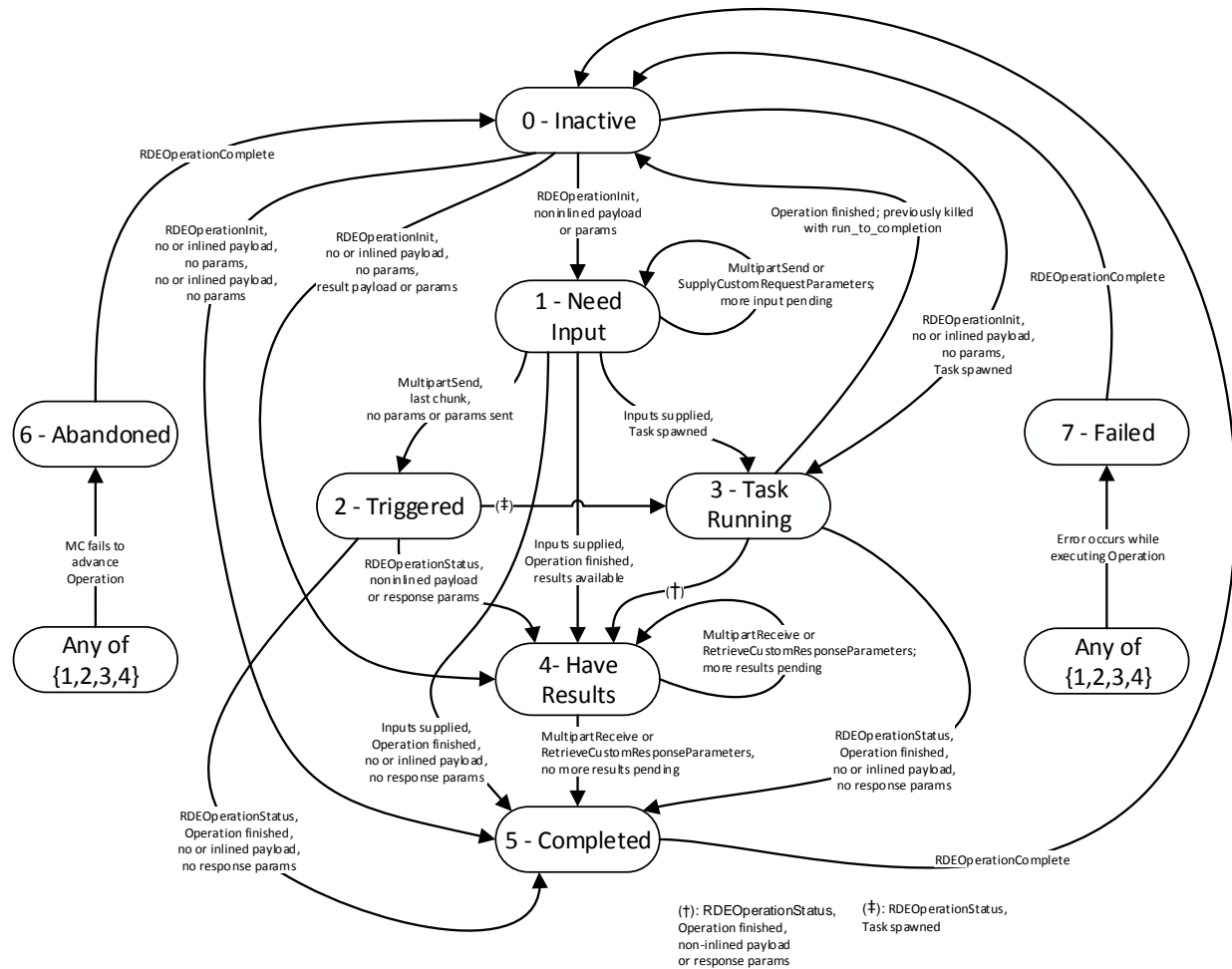
Current State	Trigger	Response	Next State
	<ul style="list-style-type: none"> - both run_to_completion and discard_record flags set 	is contradictory)	
	RDEOperationKill <ul style="list-style-type: none"> - Operation executing - Operation cannot be killed or Operation execution finished - any combination 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
	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
	MultipartSend, MultipartReceive, SupplyCustomRequestParameters, RetrieveCustomResponseParameters, RDEOperationComplete	ERROR_UNEXPECTED	FAILED
	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 NEED_INPUT
	Error occurs in processing of	None	FAILED

Current State	Trigger	Response	Next State
	Operation		
	RDEOperationKill; <ul style="list-style-type: none"> - Operation can be aborted - neither run_to_completion nor discard_record flag set 	Success	FAILED
	RDEOperationKill <ul style="list-style-type: none"> - Operation executing - Operation can be killed - run_to_completion flag set - discard_record flag not 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

Current State	Trigger	Response	Next State
	RDEOperationStatus; <ul style="list-style-type: none"> - execution finished - no payload or payload fits in response message such that total response message size is within negotiated maximum transfer chunk - no response parameters 	OPERATION_COMPLETED	COMPLETED
	MultipartSend, MultipartReceive, RDEOperationComplete	ERROR_UNEXPECTED	FAILED
	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 NEED_INPUT
	RDEOperationKill; <ul style="list-style-type: none"> - any combination of run_to_completion and discard_record flags set 	ERROR_OPERATION_UNKILLABLE	HAVE_RESULTS
	RDEOperationStatus	OPERATION_HAVE_RESULTS	HAVE_RESULTS
	MultipartReceive; <ul style="list-style-type: none"> - MC aborts transfer 	Do not send data; Success; Prepare to restart transfer with next MultipartReceive command	HAVE_RESULTS
	MultipartReceive; <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)
	MultipartReceive; <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
	MultipartReceive; <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
MultipartReceive; <ul style="list-style-type: none"> - no more data to transfer from the RDE Device to 	Send data; Success	COMPLETED	

Current State	Trigger	Response	Next State
	the MC after this chunk - no response parameters present		
	T_{abandon} timeout waiting for MultipartReceive and/or RetrieveCustomResponseParameters commands (depending on type of results still to be retrieved)	None	ABANDONED
	ReceiveCustomResponseParameters - RDE Device was not ready when RDEOperationInit command was sent and wished to specify a deferral timeframe	Deferral Timeframe; Success	FAILED
	ReceiveCustomResponseParameters - response payload data not yet transferred	Success	HAVE_RESULTS
	ReceiveCustomResponseParameters - response payload data partially transferred	ERROR_UNEXPECTED	HAVE_RESULTS
	ReceiveCustomResponseParameters - 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 NEED_INPUT
	RDEOperationKill; - any 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 - 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 NEED_INPUT
	RDEOperationKill; - any 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
7 - 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 NEED_INPUT
	RDEOperationKill - any 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



2461

2462

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

2463

9.3 Event lifecycle

2464

Table 46 describes the operating behavior for MCs and RDE Devices over the lifecycle of Events

2465

depicted visually in Figure 16. This sequence applies to both Task completion Events and schema-based

2466

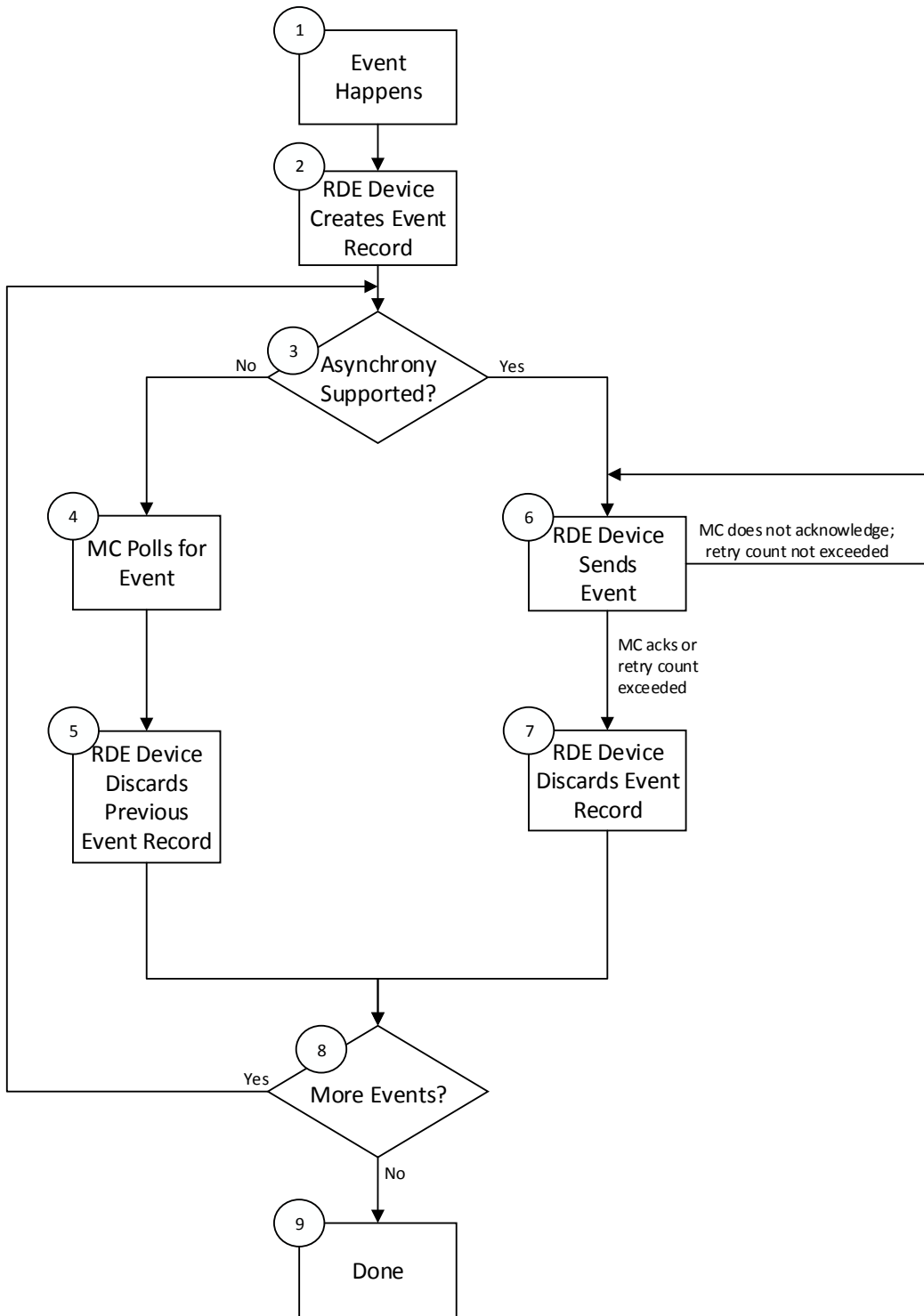
Events. MC and RDE Device implementations of RDE shall comply with the sequences presented here.

2467

Table 46 – 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

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



2468

2469

Figure 16 – Redfish event lifecycle overview

2470 **10 PLDM commands for Redfish Device Enablement**

2471 This clause provides the list of command codes that are used by MCs and RDE Devices that implement
 2472 PLDM Redfish Device Enablement as defined in this specification. The command codes for the PLDM
 2473 messages are given in Table 47. RDE Devices and MCs shall implement all commands where the entry
 2474 in the “Command Requirement for RDE Device” or “Command Requirement for MC”, respectively, is
 2475 listed as Mandatory. RDE Devices and MCs may optionally implement any commands where the entry in
 2476 the “Command Requirement for RDE Device” or “Command Requirement for MC”, respectively, is listed
 2477 as Optional.

2478 **Table 47 – 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
Reserved	0x06-0x0F				
RDE Operation and Task Commands					
RDEOperationInit	0x10	Mandatory	Mandatory	MC	See 12.1
SupplyCustomRequestParameters	0x11	Mandatory	Mandatory	MC	See 12.2
RetrieveCustomResponseParameters	0x12	Mandatory	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					
MultipartSend	0x30	Conditional ₁	Conditional ₁	MC	See 13.1
MultipartReceive	0x31	Mandatory	Mandatory	MC	See 13.2
Reserved	0x32-0x3F				
Reserved For Future Use					
Reserved	0x40-0xFF				

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

2479 Notes:

- 2480 1) MultipartSend is required if the RDE Device intends to support write Operations
- 2481 2) SetEventReceiver is mandatory if the RDE Device intends to support asynchronous messaging for
- 2482 Events via PlatformEventMessage
- 2483 3) RDE Devices and MCs must support either PlatformEventMessage or
- 2484 PollForPlatformEventMessage in order to enable Event support

2485 **11 PLDM for Redfish Device Enablement – Discovery and schema**

2486 **commands**

2487 This clause describes the commands that are used by RDE Devices and MCs that implement the

2488 discovery and schema management commands defined in this specification. The command codes for the

2489 PLDM messages are given in Table 47.

2490 **11.1 NegotiateRedfishParameters command format**

2491 This command enables the MC to negotiate general Redfish parameters with an RDE Device. The MC

2492 shall send this command to the RDE Device prior to any other RDE command. An RDE Device that

2493 supports multiple mediums shall provide the same response to this command independent of the medium

2494 on which this command was issued.

2495 When the RDE Device receives a request with data formatted per the Request Data section below, it shall

2496 respond with data formatted per the Response Data section. For a non-SUCCESS CompletionCode, only

2497 the CompletionCode field of the Response Data shall be returned.

2498

Table 48 – 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>[15:8] - reserved</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>[7:2] - reserved</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>

2499 **11.2 NegotiateMediumParameters command format**

2500 This command enables the MC to negotiate medium-specific parameters with an RDE Device. The MC
 2501 should invoke this command on each communication medium (e.g., RBT, SMBus, PCIe VDM) on which it
 2502 intends to interface with the RDE Device. The MC shall send this command over the transport for a
 2503 particular medium to negotiate parameters for that medium. When the RDE Device receives a request
 2504 with data formatted per the Request Data section below, it shall respond with data formatted per the
 2505 Response Data section. For a non-SUCCESS CompletionCode, only the CompletionCode field of the
 2506 Response Data shall be returned.

2507

Table 49 – 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>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>
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>NOTE for MCTP-based mediums, this is relative to the message size, not the packet size.</p>

2508 11.3 GetSchemaDictionary command format

2509 This command enables the MC to retrieve a dictionary (full or truncated; see clause 7.2.3) associated with
 2510 a Redfish Resource PDR. After invoking the GetSchemaDictionary command, the MC shall, upon receipt
 2511 of a successful completion code and a valid read transfer handle, invoke one or more MultipartReceive
 2512 commands (clause 13.2) to transfer data for the dictionary from the RDE Device. The MC shall only have
 2513 one dictionary retrieval in process from a given RDE Device at any time. In the event that the MC begins
 2514 a dictionary retrieval when a previous retrieval has not yet completed (i.e., more chunks of dictionary data
 2515 remain to be retrieved), the previous retrieval is implicitly aborted and the RDE Device may discard any
 2516 data associated with the transfer.

2517 When the RDE Device receives a request with data formatted per the Request Data section below, it shall
 2518 respond with data formatted per the Response Data section if it supports the command. For a non-
 2519 SUCCESS CompletionCode, only the CompletionCode field of the Response Data shall be returned.

2520

Table 50 – 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 MultipartReceive commands (see clause 13.2). In conjunction with a non-failed CompletionCode, the RDE Device shall return a valid transfer handle.</p>

2521 **11.4 GetSchemaURI command format**

2522 This command enables the MC to retrieve the formal URI for one of the RDE Device's schemas.

2523 When the RDE Device receives a request with data formatted per the Request Data section below, it shall
 2524 respond with data formatted per the Response Data section if it supports the command. For a non-
 2525 SUCCESS CompletionCode, only the CompletionCode field of the Response Data shall be returned.

2526 **Table 51 – GetSchemaURI command format**

Type	Request data
uint32	<p>ResourceID</p> <p>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.</p>
schemaClass	<p>RequestedSchemaClass</p> <p>The class of schema being requested</p>
uint8	<p>OEMExtensionNumber</p> <p>Shall be zero for a standard DMTF-published schema, or the one-based OEM extension to a standard schema</p>

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

2527 11.5 GetResourceETag command format

2528 This command enables the MC to retrieve a hashed summary of the data contained immediately within a
 2529 resource, including all OEM extensions to it, or of all data within an RDE Device. The retrieved ETag shall
 2530 reflect the underlying data as specified in the Redfish specification ([DSP0266](#)).

2531 When the RDE Device receives a request with data formatted per the Request Data section below, it shall
 2532 respond with data formatted per the Response Data section if it supports the command. For a non-
 2533 SUCCESS CompletionCode, only the CompletionCode field of the Response Data shall be returned.

2534 **Table 52 – GetResourceETag command format**

Type	Request data
uint32	<p>ResourceID</p> <p>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</p>
Type	Response data
enum8	<p>CompletionCode</p> <p>value: { PLDM_BASE_CODES, ERROR_NO_SUCH_RESOURCE }</p>
varstring	<p>ETag</p> <p>The ETag string data; the string text format shall be UTF-8. This field shall be omitted if the CompletionCode is not SUCCESS.</p>

2535 **12 PLDM for Redfish Device Enablement – RDE Operation and Task**
 2536 **commands**

2537 This clause describes the Task commands that are used by RDE Devices and MCs that implement
 2538 Redfish Device Enablement as defined in this specification. The command numbers for the PLDM
 2539 messages are given in Table 47.

2540 **12.1 RDEOperationInit command format**

2541 This command enables the MC to initiate a Redfish Operation with an RDE Device on behalf of a client.
 2542 After invoking the RDEOperationInit command, the MC may, upon receipt of a successful completion
 2543 code, invoke one or more MultipartSend commands (clause 13.1) to transfer payload data of type
 2544 bejEncoding to the RDE Device. The MC shall only use MultipartSend to transfer the payload data if that
 2545 data cannot fit in the request message of the RDEOperationInit command. After any payload has been
 2546 transferred, the MC may invoke the SupplyCustomRequestParameters command if additional parameters
 2547 are required. See clause 9 for more details on the Operation lifecycle.

2548 After the RDE Device receives the RDEOperationInit command, if flags are not set to indicate that it
 2549 should expect either payload data or custom request parameters, the RDE Device is triggered and shall
 2550 begin execution of the Operation. Similarly, if the flags are set to expect a payload but not parameters,
 2551 and the payload is contained inline in the request message, the RDE Device is implicitly triggered and
 2552 shall begin execution of the Operation.

2553 If triggered, the RDE Device shall respond with results if it is able to complete the Operation within the
 2554 time period required for a response to this message. If there is a response payload that fits within the
 2555 ResponsePayload field while maintaining a message size compatible with the negotiated maximum chunk
 2556 size (see NegotiateMediumParameters, clause 11.2), the RDE Device shall include it within this
 2557 response. Only if including a response payload would cause the message to exceed the negotiated chunk
 2558 size may the RDE Device flag it for transfer via MultipartReceive.

2559 When the RDE Device receives a request with data formatted per the Request Data section below, it shall
 2560 respond with data formatted per the Response Data section. Even with a non-SUCCESS
 2561 CompletionCode, all fields of the Response Data shall be returned.

2562 **Table 53 – RDEOperationInit command format**

Type	Request data
uint32	ResourceID The resourceID of a resource in the Redfish Resource PDR for the data that is the target of this operation
rdeOpID	OperationID Identification number for this Operation; must match the one used for all commands relating to this Operation
enum8	OperationType The type of Redfish Operation being performed. values: { OPERATION_HEAD = 0; OPERATION_READ = 1; OPERATION_CREATE = 2; OPERATION_DELETE = 3; OPERATION_UPDATE = 4; OPERATION_REPLACE = 5; OPERATION_ACTION = 6 }

Type	Request data
bitfield8	<p>OperationFlags</p> <p>Flags associated with this Operation:</p> <p>[7:3] - reserved for future use</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 MultipartSend 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 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 if the locator_valid bit in the OperationFlags field above is set to 0b.</p>
uint32	<p>RequestPayloadLength</p> <p>Length in bytes of the request payload in this message. This value shall be zero 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>

Type	Request data
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>
Type	Response data (continued)
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 MultipartReceive 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>
bitfield8	<p>PermissionFlags</p> <p>Indicates the access level (types of Operations; see Table 31) 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>To process PermissionFlags, the MC shall behave as described in clause 7.2.4.2.8.</p> <p>This field shall be ignored by the MC and set to 0b for all bits unless the Operation is failed with completion code ERROR_NOT_ALLOWED.</p>

Type	Request data
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 (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 failed or not yet finished.</p> <p>To process an ETag, the MC shall behave as described in clause 7.2.4.2.4.</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>

2563 12.2 SupplyCustomRequestParameters command format

2564 This command enables the MC to send custom HTTP/HTTPS X- headers and other uncommon request
 2565 parameters to an RDE Device to be applied to an Operation if the client's HTTP operation contains any
 2566 such parameters. The MC must not use this command to submit any headers for which a standard
 2567 handling is defined in either this specification or [DSP0266](#). If the client's HTTP operation does not contain
 2568 the parameters conveyed in this command, the MC shall not send this command as part of its processing
 2569 of the Operation.

2570 The MC shall only invoke this command in the event that at least one custom header or uncommon
 2571 request parameter needs to be transferred to the RDE Device. When sent, the
 2572 **SupplyCustomRequestParameters** command shall be invoked after the MC sends the
 2573 RDEOperationInit command.

2574 After the RDE Device receives the SupplyCustomRequestParameters command, if flags from the original
 2575 RDEOperationInit command (see clause 12.1) were not set to indicate that it should expect payload data
 2576 or if the RDE Device has already received payload data, the RDE Device shall consider itself triggered
 2577 and begin execution of the Operation.

2578 If triggered, the RDE Device shall respond with results if it is able to complete the Operation within the
 2579 time period required for a response to this message. If there is a response payload that fits within the
 2580 ResponsePayload field while maintaining a message size compatible with the negotiated maximum chunk
 2581 size (see clause 11.2), the RDE Device shall include it within this response. Only if including a response
 2582 payload would cause the message to exceed the negotiated chunk size may the RDE Device flag it for
 2583 transfer via MultipartReceive.

2584 The size of the request message is limited to the negotiated maximum chunk size (see clause 11.2). If the
 2585 client supplied sufficiently many custom request headers and/or ETags that the request message would
 2586 exceed this negotiated size, the MC shall abort the request and perform the following steps:

- 2587 1) Use the RDEOperationKill (see clause 12.6) and then RDEOperationComplete (see clause
 2588 12.4) commands to abort and finalize the Operation if it had already been initiated via
 2589 RDEOperationInit (see clause 12.1).
- 2590 2) Return to the client HTTP/HTTPS error code 431, Request Header Fields Too Large.
- 2591 3) Cease processing of the client request.

2592 When the RDE Device receives a request with data formatted per the Request Data section below, it shall
 2593 respond with data formatted per the Response Data section. Even with a non-SUCCESS
 2594 CompletionCode, all fields of the Response Data shall be returned.

2595 **Table 54 – 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. When supporting this command, an RDE Device shall encode pages expanded into with the bejResourceLinkExpansion format specification
uint16	CollectionSkip 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. To process a CollectionSkip value, the RDE Device shall respond as described in clause 7.2.4.3.1
uint16	CollectionTop 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. To process a CollectionTop value, the RDE Device shall respond as described in clause 7.2.4.3.2
uint16	PaginationOffset 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. To process a PaginationOffset value, the RDE Device shall respond as described in clause 14.2.8
enum8	ETagOperation To process an ETagOperation, the RDE Device shall respond as described in clauses 7.2.4.2.1 and 7.2.4.2.2. values: { ETAG_IGNORE = 0; ETAG_IF_MATCH = 1; ETAG_IF_NONE_MATCH = 2 }

Type	Request data
uint8	ETagCount Number of ETags supplied in this message; should be zero if ETagOperation above is ETAG_IGNORE and nonzero otherwise
varstring	ETag [0] String data for first ETag, if ETagCount > 0. This string shall be UTF-8 format. To process an ETag, the MC shall behave as described in clause 7.2.4.2.4.
...	Additional ETags
uint8	HeaderCount The number of custom headers being supplied in this operation. To process custom headers, the RDE Device shall respond as described in clause 7.2.4.2.3
varstring	HeaderName [0] The name of the header, including the X- prefix
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
...	...
Type	Response data
enum8	CompletionCode value: { PLDM_BASE_CODES, ERROR_OPERATION_ABANDONED, ERROR_OPERATION_FAILED, ERROR_UNSUPPORTED, ERROR_NOT_ALLOWED, ERROR_WRONG_LOCATION_TYPE, ERROR_UNEXPECTED, ERROR_UNRECOGNIZED_CUSTOM_HEADER, ERROR_ETAG_MATCH, ERROR_NO_SUCH_RESOURCE } Response codes ERROR_NOT_ALLOWED, ERROR_WRONG_LOCATION_TYPE, and ERROR_UNSUPPORTED shall be used to indicate that the Operation has been triggered and an error was encountered in executing it. These responses represent an operational failure, not a command failure.
enum8	OperationStatus values: { OPERATION_INACTIVE = 0; OPERATION_NEEDS_INPUT = 1; OPERATION_TRIGGERED = 2; OPERATION_RUNNING = 3; OPERATION_HAVE_RESULTS = 4; OPERATION_COMPLETED = 5, OPERATION_FAILED = 6, OPERATION_ABANDONED = 7 }
uint8	CompletionPercentage 0..100: percentage complete; 101-253: reserved for future use; 254: not supported or otherwise unable to estimate (but a valid Operation) 255: invalid Operation This value shall be zero if the Operation has not yet been triggered or if the Operation has failed.
uint32	CompletionTimeSeconds An estimate of the number of seconds remaining before the Operation is completed, or 0xFFFF FFFF if such an estimate cannot be provided. This value shall be 0xFFFF FFFF if the Operation has not yet been triggered or if the Operation has failed.

Type	Request data
Type	Response data (continued)
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 MultipartReceive 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>
bitfield8	<p>PermissionFlags</p> <p>Indicates the access level (types of Operations; see Table 31) 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] - execute access (for actions); 1b = access allowed</p> <p>[3] - delete access; 1b = access allowed</p> <p>[2] - create access; 1b = access allowed</p> <p>[1] - write access; 1b = access allowed</p> <p>[0] - read access; 1b = access allowed</p> <p>To process PermissionFlags, the MC shall behave as described in clause 7.2.4.2.8.</p> <p>This field shall be ignored by the MC and set to 0b for all bits unless the Operation is failed with completion code ERROR_NOT_ALLOWED.</p>
uint32	<p>ResponsePayloadLength</p> <p>Length in bytes of the response payload in this message. This value shall be zero under any of the following conditions:</p> <ul style="list-style-type: none"> • The Operation has not yet been triggered • The Operation status is not completed or failed, as indicated by the OperationStatus parameter above. For a failed Operation, a @Message.ExtendedInfo annotation may be supplied in the response payload. • There is no response payload as indicated by Bit 2 of the OperationExecutionFlags parameter above • The entire payload cannot fit within this message, subject to the maximum transfer chunk size as determined at registration time via the NegotiateMediumParameters command

Type	Request data
Type	Response data (continued)
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 as described in clause 7.2.4.2.4.</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>

2596 12.3 RetrieveCustomResponseParameters command format

2597 This command enables the MC to retrieve custom HTTP/HTTPS headers or other uncommon response
 2598 parameters from an RDE Device to be forwarded to the client that initiated a Redfish operation. The MC
 2599 shall only invoke this command when the **HaveCustomResponseParameters** flag in the response
 2600 message for a triggered RDE command indicates that it is needed.

2601 The RDE Device shall not supply more response headers than would allow the response message to fit in
 2602 the negotiated maximum transfer chunk size (see clause 11.2).

2603 When the RDE Device receives a request with data formatted per the Request Data section below, it shall
 2604 respond with data formatted per the Response Data section. For a non-SUCCESS CompletionCode, only
 2605 the CompletionCode field of the Response Data shall be returned.

2606 Table 55 – RetrieveCustomResponseParameters command format

Type	Request data
uint32	<p>ResourceID</p> <p>The resourceID of a resource in the Redfish Resource PDR for the instance from which custom headers should be reported</p>
rdeOpID	<p>OperationID</p> <p>Identification number for this Operation; must match the one used for all commands relating to this Operation</p>

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. To process a DeferralTimeframe, the MC shall behave as described 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. To process a NewResourceID, the MC shall behave as described in clause 7.2.4.2.6.
Type	Response data (continued)
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
...	...

2607 **12.4 RDEOperationComplete command format**

2608 This command enables the MC to inform an RDE Device that it considers an Operation to be complete,
2609 including failed and abandoned Operations. The RDE Device in turn may discard any internal records for
2610 the Operation.

2611 When the RDE Device receives a request with data formatted per the Request Data section below, it shall
2612 respond with data formatted per the Response Data section.

2613 **Table 56 – 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 }

2614 12.5 RDEOperationStatus command format

2615 This command enables the MC to query an RDE Device for the status of an Operation. It is additionally
2616 used to collect the initial response when an RDE Operation is triggered by a MultipartSend command or
2617 after a Task finishes asynchronous execution.

2618 When providing result data for an Operation that has finished executing, if there is a response payload
2619 that fits within the ResponsePayload field while maintaining a message size compatible with the
2620 negotiated maximum chunk size (see NegotiateMediumParameters, clause 11.2), the RDE Device shall
2621 include it within this response. Only if including a response payload would cause the message to exceed
2622 the negotiated chunk size may the RDE Device flag it for transfer via MultipartReceive.

2623 When the RDE Device receives a request with data formatted per the Request Data section below, it shall
2624 respond with data formatted per the Response Data section. Even with a non-SUCCESS
2625 CompletionCode, all fields of the Response Data shall be returned.

2626 **Table 57 – 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	CompletionCode value: { PLDM_BASE_CODES, ERROR_NOT_ALLOWED, ERROR_WRONG_LOCATION_TYPE, ERROR_UNSUPPORTED, , ERROR_NO_SUCH_RESOURCE } Response codes ERROR_NOT_ALLOWED, ERROR_WRONG_LOCATION_TYPE, and ERROR_UNSUPPORTED shall be used to indicate that the Operation has been triggered and an error was encountered in executing it. These responses represent an operational failure, not a command failure. The RDE Device shall not respond with any of the following codes as these statuses shall be reported in the OperationStatus field below: ERROR_NO_SUCH_RESOURCE, ERROR_OPERATION_ABANDONED, ERROR_OPERATION_FAILED, ERROR_UNEXPECTED.
enum8	OperationStatus values: { OPERATION_INACTIVE = 0; OPERATION_NEEDS_INPUT = 1; OPERATION_TRIGGERED = 2; OPERATION_RUNNING = 3; OPERATION_HAVE_RESULTS = 4; OPERATION_COMPLETED = 5, OPERATION_FAILED = 6, OPERATION_ABANDONED = 7 }
uint8	CompletionPercentage 0..100: percentage complete; 101-253: reserved for future use; 254: not supported or otherwise unable to estimate (but a valid Operation) 255: invalid Operation This value shall be zero if the Operation has not yet been triggered or if the Operation has failed.

Type	Response data (continued)
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 MultipartReceive 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 MultipartSend or MultipartReceive command.</p>
bitfield8	<p>PermissionFlags</p> <p>Indicates the access level (types of Operations; see Table 31) 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] - execute access (for actions); 1b = access allowed</p> <p>[3] - delete access; 1b = access allowed</p> <p>[2] - create access; 1b = access allowed</p> <p>[1] - write access; 1b = access allowed</p> <p>[0] - read access; 1b = access allowed</p> <p>To process PermissionFlags, the MC shall behave as described in clause 7.2.4.2.8.</p> <p>This field shall be ignored by the MC and set to 0b for all bits unless the Operation is failed with completion code ERROR_NOT_ALLOWED..</p>

Type	Response data (continued)
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>To process an ETag, the MC shall behave as described in clause 7.2.4.2.4.</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>

2627 12.6 RDEOperationKill command format

2628 This command enables the MC to request that an RDE Device terminate an Operation. The RDE Device
 2629 shall kill the Operation if the Operation can be killed; however, the MC must be aware that not all
 2630 Operations can be terminated.

2631 When the RDE Device receives a request with data formatted per the Request Data section below, it shall
 2632 respond with data formatted per the Response Data section if it supports the command. Even with a non-
 2633 SUCCESS CompletionCode, all fields of the Response Data shall be returned.

2634

Table 58 – RDEOperationKill command format

Type	Request data
uint32	<p>ResourceID</p> <p>The resourceID of a resource in the Redfish Resource PDR to which the Task’s operation was targeted</p>
rdeOpID	<p>OperationID</p> <p>Identification number for this Operation; must match the one used for all commands relating to this Operation</p>
bitfield8	<p>KillFlags</p> <p>Flags for killing the Operation:</p> <p>[7:2] - reserved for future use</p> <p>[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.</p> <p>[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.</p>
Type	Response data
enum8	<p>CompletionCode</p> <p>value: { PLDM_BASE_CODES, ERROR_OPERATION_ABANDONED, ERROR_OPERATION_FAILED, ERROR_OPERATION_UNKILLABLE, ERROR_NO_SUCH_RESOURCE }</p>

2635 **12.7 RDEOperationEnumerate command format**

2636 This command enables the MC to request that an RDE Device enumerate all Operations that are
 2637 currently active (not in state INACTIVE in the Operation lifecycle state machine of clause 9.2.3.2). It is
 2638 expected that the MC will typically use this command during its initialization to discover any Operations
 2639 that spawned Tasks that were active through a shutdown.

2640 NOTE When instantiating Operations, the RDE Device shall not create a new Operation if including the total data
 2641 for all Operations would cause the response message for this command to exceed the negotiated maximum
 2642 transfer chunk size (see clause 11.2) for any of the mediums on which the MC has communicated with the
 2643 RDE Device.

2644 If the RDE Device accepts operations from protocols other than Redfish, it should make them visible as
 2645 RDE Operations while they are active by enumerating them in response to this command.

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 if it supports the command. For a non-
 2648 SUCCESS CompletionCode, only the CompletionCode field of the Response Data shall be returned.

2649

Table 59 – 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 }

2650 13 PLDM for Redfish Device Enablement – Utility commands

2651 13.1 MultipartSend command format

2652 This command enables the MC to send a large volume of data to an RDE Device. In the event of a data
 2653 checksum error, the MC may reissue the first MultipartSend command with the initial data transfer handle;
 2654 the RDE Device shall recognize this to mean that the transfer failed and respond as if this were the first
 2655 transfer attempt. If the MC chooses not to restart the transfer, or in any other error occurs, the MC should
 2656 abandon the transfer. In the latter case, if the transfer is part of an Operation, the MC shall explicitly abort

2657 and then finalize the Operation via the RDEOperationKill and RDEOperationComplete commands (see
 2658 clauses 12.6 and 12.4).

2659 Similarly, in the event of transient transfer errors for individual chunks of the data, the MC may retry those
 2660 chunks by reissuing the MultipartSend command corresponding to those chunks provided it has not yet
 2661 issued a MultipartSend command for a subsequent chunk. When the RDE Device receives a request with
 2662 data formatted per the Request Data section below, it shall respond with data formatted per the
 2663 Response Data section. For a non-SUCCESS CompletionCode, only the CompletionCode field of the
 2664 Response Data shall be returned.

2665 **Table 60 – MultipartSend 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>
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>

2666 13.2 MultipartReceive command format

2667 This command enables the MC to receive a large volume of data from an RDE Device. In the event of a
 2668 data checksum error, the MC may reissue the first MultipartReceive command with the initial data transfer
 2669 handle; the RDE Device shall recognize this to mean that the transfer failed and respond as if this were
 2670 the first transfer attempt. If the MC chooses not to restart the transfer, or in any other error occurs, the MC
 2671 should abandon the transfer. In the latter case, if the transfer is part of an Operation, the MC shall
 2672 explicitly abort and finalize the Operation via the RDEOperationKill and then RDEOperationComplete
 2673 commands (see clauses 12.6 and 12.4).

2674 Similarly, in the event of transient transfer errors for individual chunks of the data, the MC may retry those
 2675 chunks by reissuing the MultipartReceive command corresponding to those chunks provided it has not
 2676 yet issued a MultipartReceive command for a subsequent chunk.

2677 When the RDE Device receives a request with data formatted per the Request Data section below, it shall
 2678 respond with data formatted per the Response Data section if it supports the command. For a non-
 2679 SUCCESS CompletionCode, only the CompletionCode field of the Response Data shall be returned.

Table 61 – MultipartReceive 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 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>

2681 14 Additional Information

2682 14.1 Multipart transfers

2683 The various commands defined in clauses 10 and 12 support bulk transfers via the MultipartSend and
 2684 MultipartReceive commands defined in clause 13. The MultipartSend and MultipartReceive commands
 2685 use flags and data transfer handles to perform multipart transfers. A data transfer handle uniquely
 2686 identifies the next part of the transfer. The data transfer handle values are implementation specific. For
 2687 example, an implementation can use memory addresses or sequence numbers as data transfer handles.

2688 14.1.1 Flag usage for MultipartSend

2689 The following list shows some requirements for using TransferOperationFlag, TransferFlag, and
 2690 DataTransferHandle in MultipartSend data transfers:

- 2691 • To prepare a large data send for use in an RDE command, a DataTransferHandle shall be sent
 2692 by the MC in the request message of the RDEOperationInit command.
- 2693 • To reflect a data transfer (re)initiated with a MultipartSend command, the TransferOperation
 2694 shall be set to XFER_FIRST_PART in the response message.
- 2695 • For transferring a part after the first part of data, the TransferOperation shall be set to
 2696 XFER_NEXT_PART and the DataTransferHandle shall be set to the NextDataTransferHandle
 2697 that was obtained in the request for the previous MultipartSend command for this data transfer.
- 2698 • The TransferFlag specified in the request for a MultipartSend command has the following
 2699 meanings:
 - 2700 – START, which is the first part of the data transfer
 - 2701 – MIDDLE, which is neither the first nor the last part of the data transfer
 - 2702 – END, which is the last part of the data transfer
 - 2703 – START_AND_END, which is the first and the last part of the data transfer. In this case, the
 2704 transfer consists of a single chunk

- 2705
- 2706
- 2707
- For a MultipartSend, 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.

2708 14.1.2 Flag usage for MultipartReceive

2709 The following list shows some requirements for using TransferOperationFlag, TransferFlag, and
2710 DataTransferHandle in MultipartReceive data transfers:

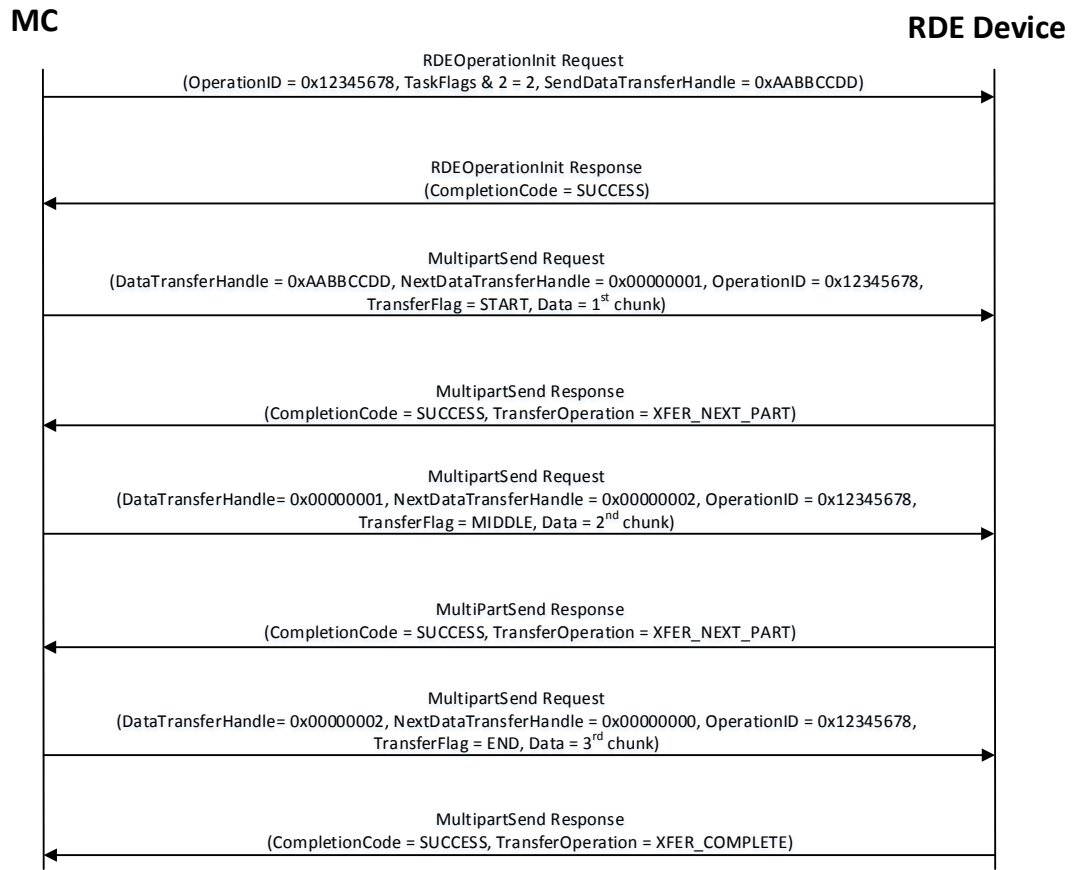
- 2711
- 2712
- 2713
- 2714
- To prepare a large data transfer receive for use in an RDE command, a DataTransferHandle shall be sent by the RDE Device in the response message to the RDEOperationInit, SupplyCustomRequestParameters, or RDEOperationStatus command after an Operation has finished execution and results are ready for pick-up.
- 2715
- 2716
- To initiate a data transfer with either a MultipartReceive command, the TransferOperation shall be set to XFER_FIRST_PART in the request message.
- 2717
- 2718
- 2719
- 2720
- For transferring a part after the first part of data, the TransferOperation shall be set to XFER_NEXT_PART and the DataTransferHandle shall be set to the NextDataTransferHandle that was obtained in the response to the previous MultipartReceive command for this data transfer.
- 2721
- 2722
- The TransferFlag specified in the response of a MultipartReceive command has the following meanings:
- 2723
- START, which is the first part of the data transfer
- 2724
- MIDDLE, which is neither the first nor the last part of the data transfer
- 2725
- END, which is the last part of the data transfer
- 2726
- START_AND_END, which is the first and the last part of the data transfer
- 2727
- For a MultipartReceive, the requester shall consider a data transfer complete when the
2728 TransferFlag in the response is set to End or StartAndEnd.

2729 14.1.3 Multipart transfer examples

2730 The following examples show how the multipart transfers can be performed using the generic mechanism
2731 defined in the commands.

2732 In the first example, the MC sends data to the RDE Device as part of a Redfish Update operation.
2733 Following the RDEOperationInit command sequence, the MC effects the transfer via a series of
2734 MultipartSend commands. Figure 17 shows the flow of the data transfer.

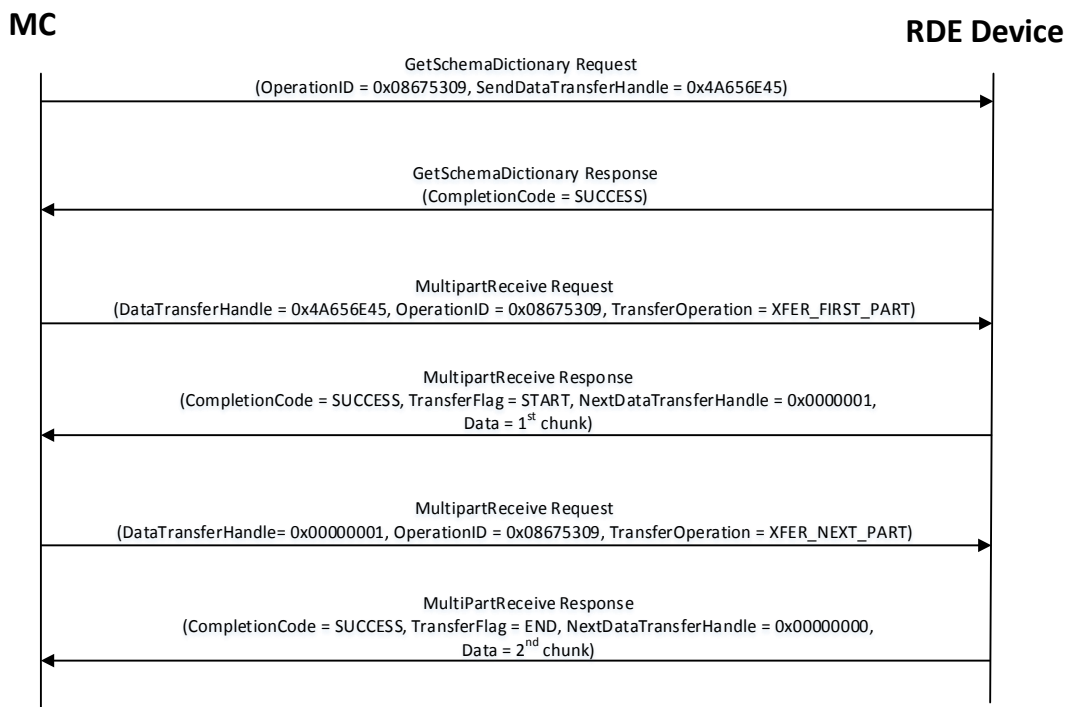
2735 In the second example, the MC retrieves the dictionary for a schema. The request is initiated via the
2736 GetSchemaDictionary command and then effected via one or more MultipartReceive commands. Figure
2737 18 shows the flow of the data transfer.



2738

2739

Figure 17 – MultipartSend example



2740

2741

Figure 18 – MultipartReceive example

2742 **14.2 Implementation notes**

2743 Several implementation notes apply to manufacturers of RDE Devices or of management controllers.

2744 **14.2.1 Schema updates**

2745 If one or more schemas for an RDE Device are updated, the RDE Device may communicate this to the
 2746 MC by triggering an event for the affected PDRs. When the MC detects a PDR update, it shall reread the
 2747 affected PDRs.

2748 **14.2.2 Storage of dictionaries**

2749 It is not necessary for the MC to maintain all dictionaries in memory at any given time. It may flush
 2750 dictionaries at will since they can be retrieved on demand from the RDE Devices via the
 2751 GetSchemaDictionary command (clause 11.2). However, if the MC has to retrieve a dictionary “on
 2752 demand” to support a Redfish query, this will likely incur a performance delay in responding to the client.
 2753 For MCs with highly limited memory that cannot retain all the dictionaries they need to support, care must
 2754 thus be exercised in the runtime selection of dictionaries to evict. Such caching considerations are
 2755 outside the scope of this specification.

2756 **14.2.3 Dictionaries for related schemas**

2757 MCs must not assume that sibling instances of Redfish Resource PDRs in a hierarchy (such as collection
 2758 members) use the same version of a schema. They could, for example, correspond to individual elements
 2759 from an array of hardware (such as a disk array) built by separate manufacturers and supporting different
 2760 versions of a major schema or with different OEM extensions to it. However, at such time as the MC has
 2761 verified that two siblings do in fact use the same schemas, there is no reason to store multiple copies of
 2762 the dictionary corresponding to that schema. Of course, sibling instances of resources stored within the

2763 same PDR share all dictionaries; it is only with instances of resources from separate PDRs that this
2764 applies.

2765 Similarly, it is expected to be fairly commonplace that the system managed by an MC could have multiple
2766 RDE Devices of the same class, such as multiple network adapters or multiple RAID array controllers. In
2767 such cases, however, there is no guarantee that each such RDE Device will support the same version of
2768 any given Redfish schema.

2769 To handle such cases, MCs have two choices. The most straightforward approach is to simply maintain
2770 each dictionary associated with the RDE Device it came from. This of course has space implications. A
2771 more practical approach is to store one copy of the dictionary for each version of the schema and then
2772 keep track of which version of the dictionary to use with which RDE Device. Because RDE Devices may
2773 support only subsets of the properties in resources, care must be taken when employing this approach to
2774 ensure that all supported properties are covered in the dictionaries selected. This may be done by
2775 merging dictionaries at runtime, though details of how to merge dictionaries are out of scope for this
2776 specification. In particular, OEM sections of dictionaries are not generally able to be merged as the
2777 sequence numbers for the names of the different OEM extensions themselves are likely to overlap.

2778 However, a yet better approach is available. In Redfish schemas, so long as only the minor and release
2779 version numbers change, schemas are required to be fully backward compatible with earlier revisions.
2780 Individual properties and enumeration values may be added but never removed. The MC can therefore
2781 leverage this to retain only the newest instance of dictionary for each major version supported by RDE
2782 Devices. Again, the fact that RDE Devices may support only subsets of the properties in a resource
2783 means that care must be taken to ensure dictionary support for all the properties used across all RDE
2784 Devices that implement any given schema.

2785 14.2.4 [MC] HTTP/HTTPS POST Operations

2786 As specified in [DSP0266](#), a Redfish POST Operation can represent either a Create Operation or an
2787 Action. To distinguish between these cases, the MC may examine the URI target supplied with the
2788 operation. If it points to a collection, the MC may assume that the Operation is a Create; if it points to an
2789 action, the MC may assume the Operation is an Action. Alternatively, the MC may presuppose that the
2790 POST is a Create Operation and if it receives an ERROR_WRONG_LOCATION_TYPE error code from
2791 the RDE Device, retry the Operation as an Action. This second approach reduces the amount of URI
2792 inspection the MC has to perform in order to proxy the Operation at the cost of a small delay in
2793 completion time for the Action case. (The supposition that POSTs correspond to Create Operations could
2794 of course be reversed, but it is expected that Actions will be much rarer than Create Operations.)
2795 Implementers should be aware that such delays could cause a client-side timeout.

2796 Another clue that could be used to differentiate between POSTs intended as create operations vs POSTs
2797 intended as actions would be trial encodings of supplied payload data. If there is no payload data, then
2798 the request is either in error or an action. In this case, the payload should be encoded with the dictionary
2799 for the major schema associated with target resource. On the other hand, if the payload is intended for a
2800 create operation, the correct dictionary to use would be the collection member dictionary, which may be
2801 retrieved via the GetSchemaDictionary command (clause 11.2), specifying
2802 COLLECTION_MEMBER_TYPE as the dictionary to retrieve.

2803 14.2.4.1 Support for Actions

2804 When a Redfish client issues a Redfish Operation for an Action, the URI target for the Action will be a
2805 POST of the form /redfish/v1/{path to root of RDE Device component}/{path to RDE Device owned
2806 resource}/Actions/schema_name.action_name. To process this, the MC may translate {path to root of
2807 RDE Device component} and {path to RDE Device owned resource} normally to identify the PDR against
2808 which the Operation should be executed. (If the URI is not in this format, this is another indication that the
2809 POST operation is probably a CREATE.) After it has performed this step, the MC can then check its PDR

2810 hierarchy to find the Redfish Action PDR containing an action named schema_name.action_name. If it
2811 doesn't find one, the MC shall respond with HTTP status code 404, Not Found and stop processing the
2812 Operation.

2813 After the correct Action is located, the MC can translate any request parameters supplied with the Action.
2814 To do so, it should look within the dictionary at the point beginning with the named action, and then
2815 navigate into the Parameters set under the action. From there, standard encoding rules apply. When
2816 supplying a locator for the Action to the RDE Device as part of the RDEOperationInit command, the MC
2817 shall not include the Parameters set as one of the sequence numbers comprising the locator; rather, it
2818 shall stop with the sequence number for the property corresponding to the Action's name.

2819 After the Action is complete, it may contain result parameters. If present, definitions for these will be found
2820 in the dictionary in a ReturnType set parallel to the Parameters set that contained any request
2821 parameters. If an Action does not contain explicit result parameters, the ReturnType set will generally not
2822 be present in the dictionary. The structure of the ReturnType set mirrors exactly that of the Parameters
2823 set.

2824 14.2.5 Consistency checking of read Operations

2825 Because the collection of data contained within a schema cannot generally be read atomically by RDE
2826 Devices, issues of consistency arise. In particular, if the RDE Device reads some of the data, performs an
2827 update, and then reads more data, there is no guarantee that data read in the separate "chunks" will be
2828 mutually consistent. While the level of risk that this could pose for a client consumer of the data may vary,
2829 the threat will not. The problem is exacerbated when reads must be performed across multiple resources
2830 in order to satisfy a client request: The window of opportunity for a write to slip in between distinct
2831 resource reads is much larger than the window between reads of individual pieces of data in a single
2832 resource.

2833 To resolve the threat of inconsistency, MCs should utilize a technique known as consistency checking.
2834 Before issuing a read, the MC should retrieve the ETag for the schema to be read, using the
2835 GetResourceETag command (clause 11.5). For a read that spans multiple resources, the global ETag
2836 should be read instead, by supplying 0xFFFFFFFF for the ResourceID in the command. The MC should
2837 then proceed with all of the reads and then check the ETag again. If the ETag matches what was initially
2838 read, the MC may conclude that the read was consistent and return it to the client. Otherwise, the MC
2839 should retry. It is expected that consistency failures will be very rare; however, if after a three attempts,
2840 the MC cannot obtain a consistent read, it should report error 500, Internal Server Error to the client.

2841 NOTE For reads that only span a single resource, if the RDE Device asserts the **atomic_resource_read** bit in the
2842 **DeviceCapabilitiesFlags** response message to the NegotiateRedfishParameters command (clause 11.1),
2843 the MC may skip consistency checking.

2844 14.2.6 [MC] Placement of RDE Device resources in the outward-facing Redfish 2845 URI hierarchy

2846 In the Redfish Resource PDRs and Redfish Entity Association PDRs that an RDE Device presents, there
2847 will normally be one or a limited number that reflect EXTERNAL (0x0000) as their ContainingResourceID.
2848 These resources need to be integrated into the outward-facing Redfish URI hierarchy. Resources that do
2849 not reflect EXTERNAL as their ContainingResourceID do not need to be placed by the MC; it is the RDE
2850 Device's responsibility to make sure that they are accessible via some chain of Redfish Resource and
2851 Redfish Entity Association PDRs (including PDRs chained via @link properties) that ultimately link to
2852 EXTERNAL.

2853 When retrieving these PDRs for RDE Device components, the MC should read the
2854 ProposedContainingResourceName from the PDR. While following this recommendation is not
2855 mandatory, the MC should use it to inform a placement decision. If the MC does not follow the placement
2856 recommendation, it should read the MajorSchemaName field to identify the type of RDE Device they
2857 correspond to. Within the canon of standard Redfish schemas, there are comparatively few that reside at

2858 the top level, and each has a well-defined place it should appear within the hierarchy. The MC should
2859 thus make a simple map of which top-level schema types map to which places in the hierarchy and use
2860 this to place RDE Devices. In making these placement decisions, the MC should take information about
2861 the hardware platform topology into account so as to best reflect the overall Redfish system.

2862 It may happen that the MC encounters a schema it does not recognize. This can occur, for example, if a
2863 new schema type is standardized after the MC firmware is built. The handling of such cases is up to the
2864 MC. One possibility would be to place the schema in the OEM section under the most appropriate
2865 subobject. For an unknown DMTF standard schema, this should be the OEM/DMTF object. (To tell that a
2866 schema is DMTF standard, the MC may retrieve the published URI via GetSchemaURI command of
2867 clause 11.4, download the schema, and inspect the schema, namespace, or other content.)

2868 Naturally, wherever the MC places the RDE Device component, it shall add a link to the RDE Device
2869 component in the JSON retrieved by a client from the enclosing location.

2870 **14.2.7 LogEntry and LogEntryCollection resources**

2871 RDE Devices that support the LogEntry and LogEntryCollection resources must be aware that large
2872 volumes of LogEntries can overwhelm the 16 bit ResourceID space available for identifying Redfish
2873 Resource PDRs. To handle this case, it is recommended that RDE Devices provide a PDR for the
2874 LogEntryCollection but do NOT provide PDRs for the individual LogEntry instances. Instead, RDE
2875 Devices that support these schemas should also support the link expansion query parameter (see \$levels
2876 in [DSP0266](#) and the LinkExpand parameter from SupplyCustomRequestParameters in clause 12.2). This
2877 means that they should fill out the related resource links in the “Members” section of the response with
2878 bejResourceLinkExpansion data in which the encoded ResourceID is set to zero to ensure that the MC
2879 gets the COLLECTION_MEMBER_TYPE dictionary from the LogEntryCollection.

2880 **14.2.8 On-demand pagination**

2881 In Redfish, certain read operations may produce a very large amount of data. For example, reading a
2882 collection with many members will produce output with size proportional to the number of members.
2883 Rather than overload clients with a huge transfer of data, Redfish Devices may paginate it into chunks
2884 and provide one page at a time with an @odata.nextlink annotation giving a URI from which to retrieve
2885 the next piece.

2886 RDE supports the same pagination approach. It is entirely at the RDE Device’s discretion whether to
2887 paginate and where to draw pagination boundaries. When the RDE Device wishes to paginate, it shall
2888 insert an @odata.nextlink annotation, using a deferred binding pagination reference (see
2889 \$LINK.PDR<resource-ID>.PAGE<pagination-offset>% in clause 8.3), filling in the next page number for
2890 the data being returned. When the MC decodes this deferred binding, it shall create a temporary URI for
2891 the pagination and expose this pagination URI in the decoded JSON response it sends back to the client.
2892 Naturally, the encoded pagination URI must be decodable to extract the page number. Finally, when the
2893 client attempts a read from the pagination URI, the MC shall extract out the page number and send it to
2894 the RDE Device via the PaginationOffset field in the request message for the
2895 SupplyCustomRequestParameters command (clause 12.2).

2896 **14.2.9 Considerations for Redfish clients**

2897 No changes to behavior are required of Redfish clients in order to interact with BEJ-based RDE Devices;
2898 the details of providing them to the client are completely transparent from the client perspective. In fact, a
2899 fundamental design goal of this specification is that it should be impossible for a client to tell whether a
2900 Redfish message was ultimately serviced by an RDE Device that operates in JSON over HTTP/HTTPS or
2901 BEJ over PLDM.

**ANNEX A
(informative)****Change log**

Version	Date	Description
1.0.0	2019-06-25	

2902
2903
2904
2905

2906