Document Identifier: DSP0218

Date: 2020-01-21

Version: 1.0.1

Platform Level Data Model (PLDM) for Redfish Device Enablement

Supersedes: 1.0.0

Document Class: Normative

Document Status: Published

Document Language: en-US

CONTENTS

Figures

Tables

Foreword

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

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

Acknowledgments

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

Editor:

- Bill Scherer – Hewlett Packard Enterprise

Contributors:

- Richelle Ahlvers – Broadcom Inc.
- Jeff Autor – Hewlett Packard Enterprise
- Patrick Caporale – Lenovo
- Mike Garrett – Hewlett Packard Enterprise
- Jeff Hilland – Hewlett Packard Enterprise
- Yuval Itkin – Mellanox Technologies
- Ira Kalman – Intel
- Eliel Louzoun – Intel
- Balaji Natrajan – Microchip Technology Inc.
- Edward Newman – Hewlett Packard Enterprise
- Zvika Perry Peleg – Cavium
- Scott Phuong, Cisco Systems, Inc.
- Jeffrey Plank – Microchip Technology Inc.
- Joey Rainville – Hewlett Packard Enterprise
- Patrick Schoeller – Hewlett Packard Enterprise
- Hemal Shah – Broadcom Inc.
- Bob Stevens – Dell
- Bill Vetter – Lenovo

Introduction

The *Platform Level Data Model (PLDM) for Redfish Device Enablement Specification* defines messages and data structures used for enabling PLDM-capable 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 may be supported under PLDM. 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.

Document conventions

Clause naming conventions

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

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

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

Typographical conventions

This document uses the following typographical conventions:

- Document titles are marked in *italics*.

Platform Level Data Model (PLDM) for Redfish Device Enablement

1 Scope

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

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

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

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

- DMTF DSP0266, *Redfish Scalable Platforms Management API Specification Redfish Scalable Platforms Management API Specification*, defines the main Redfish protocols.
- DMTF DSP0240, *Platform Level Data Model (PLDM) Base Specification*, provides definitions of common terminology, conventions, and notations used across the different PLDM specifications as well as the general operation of the PLDM messaging protocol and message format.
- DMTF DSP0245, *Platform Level Data Model (PLDM) IDs and Codes Specification*, defines the values that are used to represent different type codes defined for PLDM messages.
- DMTF DSP0248, *Platform Level Data Model (PLDM) for Platform Monitoring and Control Specification,* defines the event and Redfish PDR data structures referenced in this specification.

2 Normative references

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

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

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

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

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

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

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

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

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

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

ECMA International Standard ECMA-404, The JSON Data Interchange Syntax, http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf

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

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

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

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

IETF RFC7231, R. Fielding et al., Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content, https://tools.ietf.org/html/rfc7231

IETF RFC 7232, R. Fielding et al., Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests, http://www.ietf.org/rfc/rfc7232.txt

IETF RFC 7234, R. Fielding et al., Hypertext Transfer Protocol (HTTP/1.1): Caching, https://tools.ietf.org/rfc/rfc7234.txt

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

ISO/IEC Directives, Part 2, *Rules for the structure and drafting of International Standards,* http://isotc.iso.org/livelink/livelink.exe?func=ll&objId=4230456&objAction=browse&sort=subtype

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

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

3 Terms and definitions

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

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

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

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

Refer to DSP0240 for terms and definitions that are used across the PLDM specifications, DSP0248 for terms and definitions used specifically for PLDM Monitoring and Control, and to DSP0266 for terms and definitions specific to Redfish. For the purposes of this document, the following additional terms and definitions apply.

3.1
Action

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

3.2
Annotation

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

3.3

Client

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

3.4

Collection

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

3.5

Device Component

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

3.6

Dictionary

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

3.7

Discovery

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

3.8

Major Schema

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

3.9

Member

Any of the independent resources contained within a collection

3.10

Metadata

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

3.11

OData

The Open Data protocol, a source of annotations in Redfish, as defined by OASIS.

3.12

OEM Extension

Any manufacturer-specific addition to major schema

3.13

Property

An individual datum contained within a Resource

3.14

RDE Device

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

3.15

RDE Provider

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

3.16

RDE Operation

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

3.17

Redfish Operation

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

3.18

Redfish Provider

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

3.19

Registration

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

3.20

Resource

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

3.21

Schema

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

3.22

Task

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

3.23

Triggering Command

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

3.24

Truncated

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

4 Symbols and abbreviated terms

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

4.1

BEJ

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

4.2

JSON

JavaScript Object Notation

4.3

RDE

Redfish Device Enablement

5 Conventions

Refer to DSP0240 for conventions, notations, and data types that are used across the PLDM specifications.

5.1 Reserved and unassigned values

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

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

5.2 Byte ordering

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

5.3 PLDM for Redfish Device Enablement data types

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

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

5.3.1 varstring PLDM data type

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

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

5.3.2 schemaClass PLDM data type

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

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

Table 3 – schemaClass enumeration

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

5.3.3 nnint PLDM data type

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

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

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)

5.3.4 bejEncoding PLDM data type

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

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

5.3.5 bejTuple PLDM data type

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

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

5.3.6 bejTupleS PLDM data type

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

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.

5.3.7 bejTupleF PLDM data type

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

Table 8 – bejTupleF encoding for BEJ

Type	Description
bitfield8	Format code; the high nibble represents the data type and the low nibble represents a series of flag bits [7:4] - principal data type; see Table 9 below for values [3] - reserved flag. 1b indicates the flag is set [2] - nullable_property flag ***. 1b indicates the flag is set [1] - read_only_property flag **. 1b indicates the flag is set [0] - deferred_binding flag *. 1b indicates the flag is set

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

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

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

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

5.3.8 bejTupleL PLDM data type

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

Table 10 – bejTupleL encoding for BEJ

Type	Description
nnint	Length in bytes of value tuple field

5.3.9 bejTupleV PLDM data type

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

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.

5.3.10 bejNull PLDM data type

The length tuple value for bejNull data shall be zero.

Table 12 – bejNull value encoding for BEJ

Type	Description
(none)	No fields

5.3.11 bejInteger PLDM data type

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

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)

5.3.12 bejEnum PLDM data type

Table 14 – bejEnum value encoding for BEJ

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

5.3.13 bejString PLDM data type

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

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

5.3.14 bejReal PLDM data type

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

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

Table 16 – bejReal value encoding for BEJ

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

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

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

Table 17 – bejReal value encoding example

Type	Bytes	Description
nnint	0x01 0x01	Length of *whole* (1 byte)

Type	Bytes	Description
bejInteger	0x01	*whole* (1)
nnint	0x01 0x03	leading zero count for *fract* (3)
nnint	0x01 0x05	*fract* (5)
nnint	0x01 0x01	Length of *exp* (1)
bejInteger	0x0A	*Exp* (10)

5.3.15 bejBoolean PLDM data type

The bejBoolean PLDM data type captures boolean data.

Table 18 – bejBoolean value encoding for BEJ

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

5.3.16 bejBytestring PLDM data type

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

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)

5.3.17 bejSet PLDM data type

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

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)

5.3.18 bejArray PLDM data type

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

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)

5.3.19 bejChoice data PLDM type

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

Table 22 – bejChoice value encoding for BEJ

Type	Description
bejTuple	Selected option

5.3.20 bejPropertyAnnotation PLDM data type

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

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

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

“Status@Redfish.RequiredOnCreate” : false

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

5.3.21 bejResourceLink PLDM data type

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.

Table 25 – bejResourceLink value encoding for BEJ

Type	Description
nnint	ResourceID of Redfish Resource PDR for linked schema

5.3.22 bejResourceLinkExpansion PLDM data type

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.

Table 26 – bejResourceLinkExpansion value encoding for BEJ

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

5.3.23 bejLocator PLDM data type

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.

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]

5.3.24 rdeOpID PLDM data type

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

Table 28 – rdeOpID data structure

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

6 PLDM for Redfish Device Enablement version

The version of this Platform Level Data Model (PLDM) for Redfish Device Enablement Specification shall be 1.0.1 (major version number 1, minor version number 0, update version number 1, and no alpha version).

In response to the GetPLDMVersion command described in DSP0240, the reported version for Type 6 (PLDM for Redfish Device Enablement, this specification) shall be encoded as 0xF1F0F100.

7 PLDM for Redfish Device Enablement Overview

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

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

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

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

- Binary Encoded JSON (BEJ)
    - Simplified compact binary format for communicating Redfish JSON data payloads
    - Captures essential schema information into a compact binary dictionary so that it does not need to be transferred as part of message payloads
    - Defined locators allow for selection of a specific object or property inside the schema's data hierarchy to perform an operation
    - Encoders and decoders account for the unordered nature of BEJ and JSON properties

- RDE Device Registration for Redfish
    - A mechanism to determine the schemas the RDE Device supports, including OEM custom extensions
    - A mechanism to determine parameters for limitations on the types of communication the RDE Device can perform, the number of outstanding operations it can support, and other management parameters

- Messaging Support for Redfish Operations via BEJ
    - Read, Update, Post, Create, Delete Operations
    - Asynchrony support for Operations that spawn long-running Tasks
    - Notification Events for completion of long-running Tasks and for other RDE Device-specific happenings[1]
    - Advanced operations such as pagination and ETag support

7.1 Redfish Provider architecture overview

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

7.1.1 Roles

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

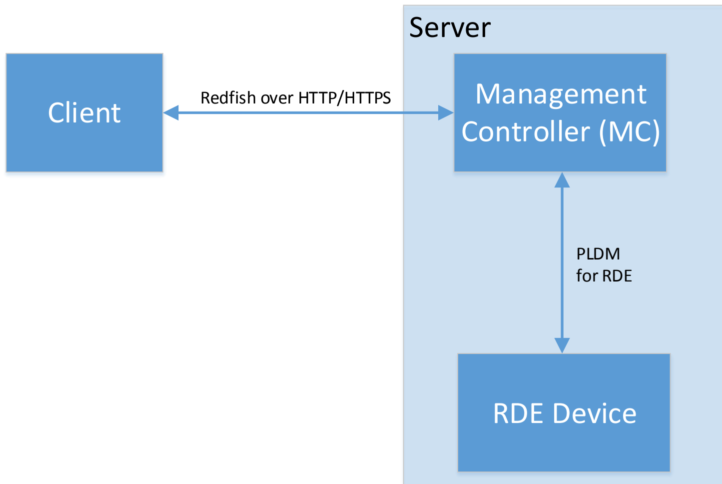


Figure 1 – RDE Roles

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

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

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

7.2 Redfish Device Enablement concepts

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

7.2.1 RDE Device discovery and registration

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

7.2.1.1 RDE Device discovery

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

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

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

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

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

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

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

7.2.1.2 RDE Device registration

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

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

In PLDM for Redfish Device Enablement, each[2] Redfish resource is uniquely identified by a Resource Identifier that maps from the identifier to a collection of schemas that define the data for it. The identifiers in turn are collected together into Redfish Resource PDRs; resources that share a common set of schemas and are linked to from a common parent (such as sibling collections members) are enumerated within the same PDR. Data for secondary schemas such as annotations or the message registry is linked together with the major schema in the PDR structure. The resources link together to form a management topology of one or more trees called device components; each resource corresponds to a node in one (or more) of these trees.

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

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

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

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

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

7.2.2 Data instances of Redfish schemas: Resources

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

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

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

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

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

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

1) The resource is contained within the RDE Device. If it is not, it does not need to be part of the RDE Device's management topology model.

2) The resource is subordinate to another resource contained within the RDE Device. If it is not, the resource can be linked directly to the resource outside the RDE Device by setting its parent field to EXTERNAL.

3) The resource needs to be linked to another resource outside the RDE Device.

7.2.2.1 Alignment of resources

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

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

7.2.2.2 Example linking of PDRs within RDE Devices

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

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

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

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

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

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

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

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

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

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

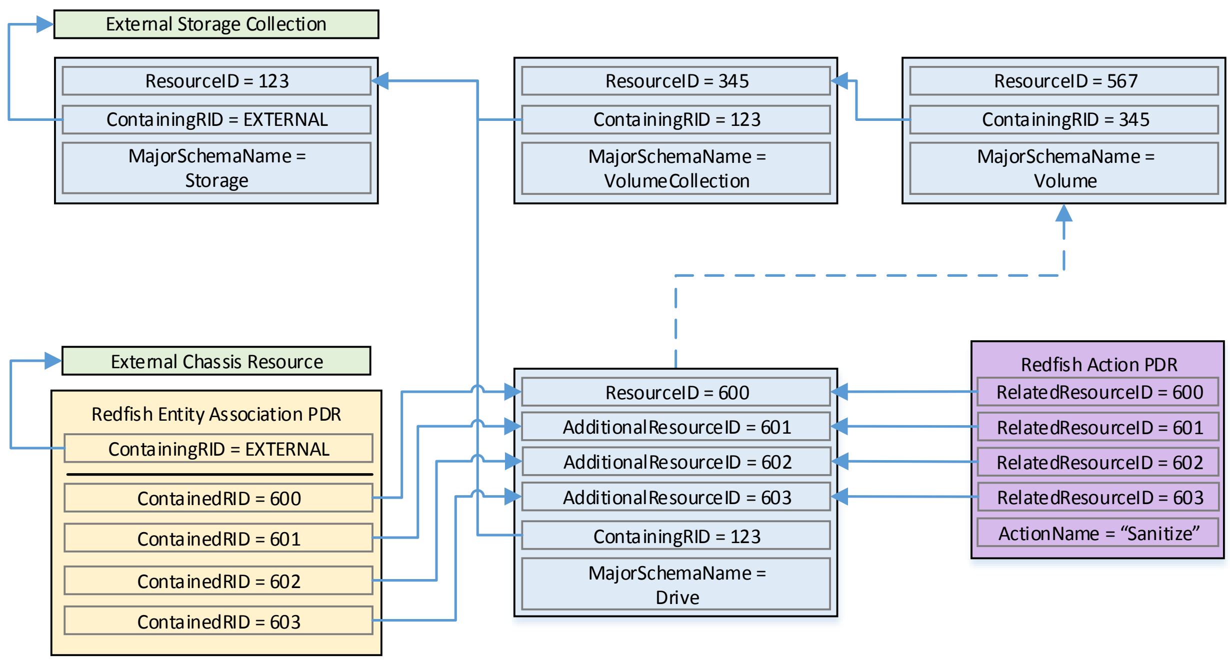


Figure 2 – Example linking of Redfish Resource and Redfish Entity Association PDRs

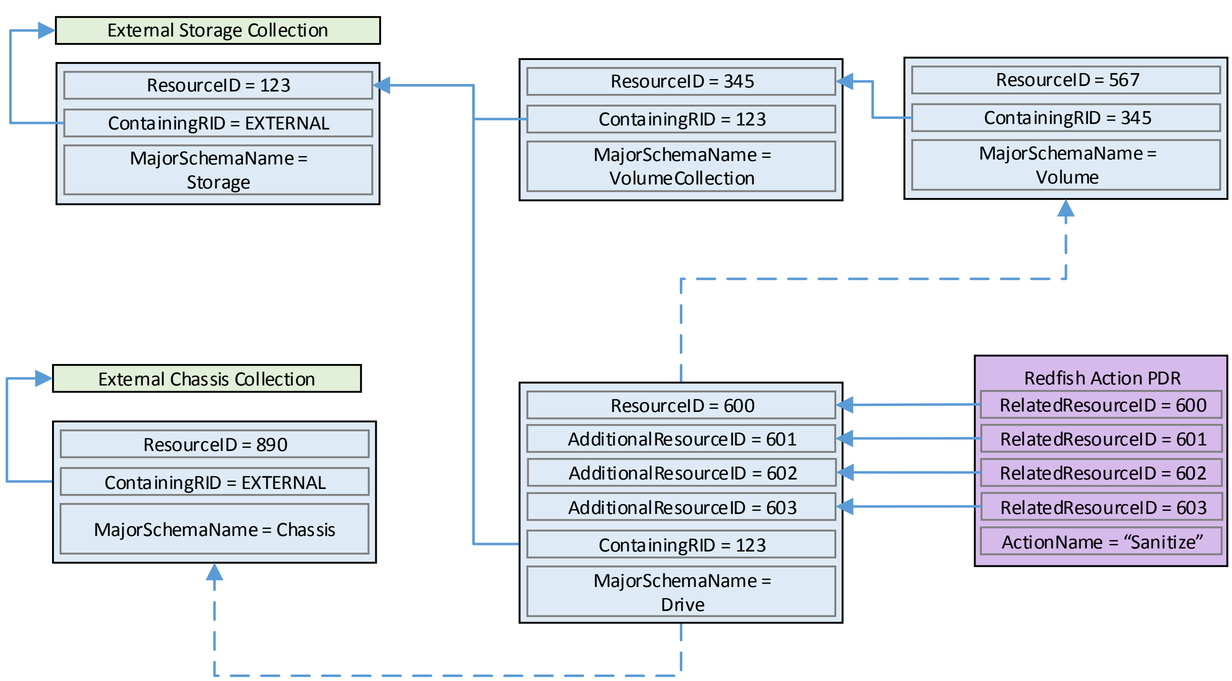


Figure 3 – Schema linking without Redfish entity association PDRs

7.2.3 Dictionaries

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

Each Redfish resource PDR can reference up to four classes of dictionaries for the schemas it can use[3]:

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

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

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

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

7.2.3.1 Canonizing a schema into a dictionary

In Redfish schemas, the order of properties is indeterminate and properties are identified by name identifiers that are of unbounded length. While this is beneficial from a human readability perspective, from a strict information-theoretical point of view, using long strings for this purpose is grossly inefficient: a numeric value of Log_2(nChildren) bits ought to be sufficient. To make this work in practice, we impose a canonical ordering that assigns each property or enumeration value a numeric sequence number. Sequence numbers shall be assigned according to the following rules:

1) The children properties (properties immediately contained within other properties such as sets or arrays) shall collectively receive an independent set of sequence numbers ranging from zero to N – 1, where N is the number of children. Sequence numbers for properties that do not share a common parent are not related in any way.

2) For the initial revision of a Redfish schema (usually v1.0), sequence numbers shall be assigned according to a strict alphabetical ordering of the property names from the schema.

3) In order to preserve backwards compatibility with earlier versions of schemas, for subsequent revisions of Redfish schemas, the sequence numbers for child properties added in that revision shall be assigned sequence numbers N to N + A – 1, where N is the number of sequence numbers assigned in the previous revision and A is the number of properties added in the present revision. (In other words, we append to the existing set and use sequence numbers beginning with the next one available.) The new sequence numbers shall be assigned according to a strict alphabetical ordering of their names from the schema.

[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.

4) In the event that a property is deleted from a schema, its sequence number shall not be reused; the sequence number for the deleted property shall forever remain allocated to that property.

5) As with properties, the values of an enumeration shall collectively receive an independent set of sequence numbers ranging from zero to N – 1, where N is the number of enumeration values. Sequence numbers for enumeration values not belonging to the same enumeration are not related in any way.

6) For the initial version of a Redfish schema, sequence numbers for enumeration values shall be assigned according to a strict alphabetical ordering of the enumeration values from the schema.

7) In order to preserve backwards compatibility with earlier versions of schemas, for subsequent revisions of Redfish schemas, the sequence numbers for enumeration values added in that revision shall be assigned sequence numbers N to N + A – 1, where N is the number of sequence numbers assigned in the previous revision and A is the number of enumeration values added in the present revision. The new sequence numbers shall be assigned according to a strict alphabetical ordering of their value strings from the schema.

8) In the event that an enumeration value is deleted from a schema, its sequence number shall not be reused; the sequence number for the deleted enumeration value shall forever remain allocated to that enumeration value.

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

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

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

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

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

7.2.3.2 Dictionary binary format

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

Table 29 – Redfish dictionary binary format

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

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

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.

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

	Byte offset			
DWORD	**+0**	**+1**	**+2**	**+3**
00	VersionTag 0x00	DictionaryFlags	EntryCount$_1$	EntryCount$_2$
01	SchemaVersion$_1$	SchemaVersion$_2$	SchemaVersion$_3$	SchemaVersion$_4$
02	DictionarySize$_1$	DictionarySize$_2$	DictionarySize$_3$	DictionarySize$_4$
03	Format[0]	SequenceNumber[0]$_2$	SequenceNumber[0]$_1$	ChildPointerOffset[0]$_2$
04	ChildPointerOffset[0]$_1$	ChildCount[0]$_2$	ChildCount[0]$_1$	NameLength[0]
05	NameOffset[0]$_2$	NameOffset[0]$_1$	…	…
06	…	…	…	…

	Byte offset			
DWORD	**+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$_{terminator}$			

Figure 4 – Dictionary binary format

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

7.2.3.2.1 Hierarchical organization of entries

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

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

7.2.3.3 Properties that support multiple formats

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

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
…	…	…	…	…

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

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

7.2.3.4 Annotation dictionary format

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

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

7.2.4 Redfish Operation support

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

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

7.2.4.1 Primary Operations

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

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

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

7.2.4.1.1 HTTP/HTTPS and Redfish

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

7.2.4.1.1.1 Redfish Operation requests

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

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

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

7.2.4.1.1.2 Redfish Operation responses

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

7.2.4.1.1.3 Generic handling of Redfish Operations

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

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

7.2.4.2 Redfish operation headers

Several HTTP/HTTPS transport layer headers modify Redfish operations when translated in the context of RDE Operations. These are summarized in Table 32. Implementation notes for how the MC and RDE Device shall support some of these modifiers – when attached to Redfish operations – may be found in the indicated subsections. For headers not listed here, the implementation is outside the scope of this specification; implementers shall refer to DSP0266 and standard HTTP/HTTPS documentation for more information on processing these headers.

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.

7.2.4.2.1 If-Match request header

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

The parameter for this header is an ETag.

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

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

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

7.2.4.2.2 If-None-Match request header

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

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

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

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

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

7.2.4.2.3 Custom HTTP headers

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

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

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

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

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

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

7.2.4.2.4 ETag response header

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

The parameter for this header is an ETag.

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

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

7.2.4.2.5 Link response header

The MC shall provide one or more Link headers in response to every Redfish HTTP/HTTPS GET and HEAD operation as described in DSP0266.

The parameter for this header is a URI.

This header has three forms as described in DSP0266; all three shall be supported by MCs. The handling for these three forms is detailed in the next three clauses.

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

7.2.4.2.5.1 Schema form

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

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

An example of a schema form link header is as follows; readers are referred to DSP0266 for more detail:

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

7.2.4.2.5.2 Annotation form

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

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

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

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

7.2.4.2.5.3 Passthrough form

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

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

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

7.2.4.2.6 Location response header

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

The parameter for this header is a URI.

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

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

7.2.4.2.7 Cache-Control response header

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

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

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

7.2.4.2.8 Allow response header

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

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

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

7.2.4.2.9 Retry-After response header

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

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

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

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

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

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

7.2.4.3 Redfish Operation request query options

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

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)

7.2.4.3.1 $skip query option

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

The parameter for this query option is an integer representing the number of members of a resource collection to skip over. See DSP0266 for more details on the usage of $skip.

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

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

7.2.4.3.2 $top query option

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

The parameter for this query option is an integer representing the number of members of a resource collection to return. See DSP0266 for more details on the usage of $top. If the parameter for $top exceeds the remaining number of members in a resource collection, the number returned shall be truncated to those remaining.

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

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

7.2.4.3.3 $expand query option

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

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

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

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

7.2.4.3.4 $levels query option qualifier

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

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

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

NegotiateRedfishParameters command (clause 11.1), or if the expansion type is not “All Links” (see clause 7.2.4.3.3), the MC shall not select passing it to the RDE Device.

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

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

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

Implementers should refer to DSP0266 for more details and caveats to be applied when expanding links with $levels > 1.

7.2.4.3.5 $select query option qualifier

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

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

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

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

7.2.4.4 HTTP/HTTPS status codes

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

7.2.4.5 Multihosting and Operations

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

7.2.5 PLDM RDE Events

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

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

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

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

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

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

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

7.2.5.1 [MC] Event subscriptions

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

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

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

7.2.6 Task support

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

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

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

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

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

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

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

7.3 Type code

Refer to DSP0245 for a list of PLDM Type Codes in use. This specification uses the PLDM Type Code 000110b as defined in DSP0245.

7.4 Transport protocol type supported

PLDM can support bindings over multiple interfaces; refer to DSP0245 for the complete list. All transport protocol types can be supported for the commands defined in Table 47.

7.5 Error completion codes

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

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
0x93	ETAG_CALCULATION_ONGOING	Calculating the ETag in response to the GetResourceETag command is taking too long to provide an immediate response	N/A – MC retries with the same command later

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

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

7.6 Timing specification

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

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.

8 Binary Encoded JSON (BEJ)

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

8.1 BEJ design principles

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

The following design principles guided the development of BEJ:

1) It must be possible to support full expressive range of JSON.

2) The encoding should be binary and compact, with as much of the encoding as possible dedicated to the JSON data elements. The amount of space afforded to metadata that conveys elements such as type format and hierarchy information should be carefully limited.

2) There is no need to support multiple encoding techniques for one type of data; there is therefore no need to distinguish which encoding technique is in use.

3) Schema information – such as the names of data items – does not need to be encoded into BEJ because the recipient can use a prior knowledge of the data organization to determine semantic information about the encoded data. In contrast to JSON, which is unordered, BEJ must adopt an explicit ordering for its data to support this goal.

4) The need for contextual awareness should be minimized in the encoding and decoding process. Supporting context requires extra lookup tables (read: more memory) and delays processing

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

8.2 SFLV tuples

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

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

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

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

8.2.1 Sequence number

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

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

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

8.2.2 Format

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

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

8.2.3 Length

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

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

8.2.4 Value

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

The following JSON data types are supported in BEJ:

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

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

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

8.3 Deferred binding of data

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

All substitution macros are bracketed with the percent sign (%) character. While it would in theory be possible for the MC to check every string it decodes for the presence of this escape character, in practice that would be an inefficient waste of MC processing time. Instead, the RDE Device shall flag any string containing substitution macros with the deferred binding bit to inform the MC of their presence; the MC shall only perform macro substitution if the deferred binding bit is set. The MC shall support the deferred bindings listed in Table 39.

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

8.4 BEJ encoding

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

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

8.4.1 Conversion of JSON data types to BEJ

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

8.4.1.1 JSON objects

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

8.4.1.2 JSON arrays

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

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

8.4.1.3 JSON numbers

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

8.4.1.4 JSON strings

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

8.4.1.5 JSON Boolean

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

8.4.1.6 JSON null

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

8.4.2 Resource links

Most schemas contain links to other schemas within their properties, formatted as @odata.id annotations. When encoding these links in BEJ, the URI shall be encoded as a bejString. Deferred binding substitutions may be employed as needed within the bejString to complete the reference.

8.4.3 Annotations

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

Annotations in Redfish have two forms:

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

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

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

Table 40 – Message annotation related property BEJ locator encoding

Description
Delimiter Shall be ‘:’

Description
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

8.4.4 Choice encoding for properties that support multiple data types

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

8.4.5 Properties with invalid values

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

8.4.6 Properties missing from dictionaries

When encoding JSON data, an encoder may find that the name of a property does not correspond to a string found in the dictionary. If the encoder is the RDE Device, this should never happen as the RDE

Device is responsible for the dictionary. This situation therefore represents a non-compliant RDE implementation.

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

8.5 BEJ decoding

This clause presents implementation considerations for the BEJ decoding process.

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

8.5.1 Conversion of BEJ data types to JSON

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

“property_name” : null

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

8.5.1.1 BEJ Set

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

“property_name” : { ‹set dependent children decoded individually as a comma-separated list› }

8.5.1.2 BEJ Array

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

“property_name” : [‹array dependent children decoded individually as a comma-separated list›]

8.5.1.3 BEJ Integer and BEJ Real

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

“property_name” : “‹decoded numeric value›”

8.5.1.4 BEJ String

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

“property_name” : “‹decoded string value›”

8.5.1.5 BEJ Boolean

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

“property_name” : ‹`true` or `false`, depending on the decoded value›

8.5.1.6 BEJ Null

BEJ Null shall be decoded to the following format:

“property_name” : `null`

8.5.1.7 BEJ Resource Link

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

“property_name” : “‹URI for the resource corresponding the Redfish Resource PDR with the supplied ResourceID›”

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

8.5.1.8 BEJ Resource Link expansion

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

‹full resource data for the Redfish Resource PDR corresponding to the supplied ResourceID›

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

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

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

8.5.2 Annotations

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

8.5.2.1 Standalone annotations

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

“@annotation_class.annotation_name” : “‹decoded annotation value›”

8.5.2.2 BEJ property annotations

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

“property_name@annotation_class.annotation_name” : “‹decoded annotation value from the annotation’s dependent child node›”

8.5.2.3 [MC] Related Properties in message annotations

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

8.5.3 Sequence numbers missing from dictionaries

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

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

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

8.5.4 Sequence numbers for read-only properties in modification Operations

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

8.6 Example encoding and decoding

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

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

8.6.1 Example dictionary

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

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

```
                "LinkDown",
                "LinkUp"
            ],
            "type": "string"
        },
        "DummySimple" : {
            "additionalProperties": false,
            "description": "The DummySimple schema represents a very simple schema used to
                            demonstrate the BEJ dictionary format.",
            "longDescription": "This resource shall not be used except for illustrative
                            purposes. It does not correspond to any real hardware or software.",
            "patternProperties": {
                "^([a-zA-Z_][a-zA-Z0-9_]*)?@(odata|Redfish|Message|Privileges)\\.[a-zA-Z_][a-zA-
Z0-9_.]+$": {
                    "description": "This property shall specify a valid odata or Redfish
                                    property.",
                    "type": [
                        "array",
                        "boolean",
                        "number",
                        "null",
                        "object",
                        "string"
                    ]
                }
            },
            "properties": {
                "@odata.context": {
                    "$ref":
                    "http://redfish.dmtf.org/schemas/v1/odata.v4_0_1.json#/definitions/context"
                },
                "@odata.id": {
                    "$ref":
                          "http://redfish.dmtf.org/schemas/v1/odata.v4_0_1.json#/definitions/id"
                },
                "@odata.type": {
                    "$ref":
                        "http://redfish.dmtf.org/schemas/v1/odata.v4_0_1.json#/definitions/type"
                },
                "ChildArrayProperty": {
                    "items": {
                        "additionalProperties": false,
                        "type": "object",
                        "properties": {
                            "LinkStatus": {
                                "anyOf": [
                                    {
                                        "$ref": "#/definitions/LinkStatus"
                                    },
                                    {
                                        "type": "null"
                                    }
                                ],
                                "readOnly": true
                            },
                            "AnotherBoolean": {
                                "type": "boolean"
                            }
                        }
                    },
                    "type": "array"
                }
            },
            "SampleIntegerProperty": {
                "type": "integer"
            },
            "Id": {
                "type": "string",
```

```
                "readOnly": true
            },
            "SampleEnabledProperty": {
                "type": "boolean"
            }
        }
    },
    "title": "#DummySimple.v1_0_0.DummySimple"
}
```

Figure 5 – DummySimple schema

NOTE This is not a published DMTF Redfish schema.

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

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

```
0x00 0x00 0x0B 0x00  0x00 0xF0 0xF0 0xF1
0x12 0x01 0x00 0x00  0x00 0x00 0x00 0x16
0x00 0x04 0x00 0x0C  0x7A 0x00 0x14 0x00
0x00 0x3E 0x00 0x01  0x00 0x13 0x86 0x00
0x56 0x01 0x00 0x00  0x00 0x00 0x00 0x03
0x99 0x00 0x74 0x02  0x00 0x00 0x00 0x00
0x00 0x16 0x9C 0x00  0x34 0x03 0x00 0x00
0x00 0x00 0x00 0x16  0xB2 0x00 0x00 0x00
0x00 0x48 0x00 0x02  0x00 0x00 0x00 0x00
0x74 0x00 0x00 0x00  0x00 0x00 0x00 0x0F
0xC8 0x00 0x46 0x01  0x00 0x5C 0x00 0x03
0x00 0x0B 0xD7 0x00  0x50 0x00 0x00 0x00
0x00 0x00 0x00 0x09  0xE2 0x00 0x50 0x01
0x00 0x00 0x00 0x00  0x00 0x07 0xEB 0x00
0x50 0x02 0x00 0x00  0x00 0x00 0x00 0x07
0xF2 0x00 0x44 0x75  0x6D 0x6D 0x79 0x53
0x69 0x6D 0x70 0x6C  0x65 0x00 0x43 0x68
0x69 0x6C 0x64 0x41  0x72 0x72 0x61 0x79
0x50 0x72 0x6F 0x70  0x65 0x72 0x74 0x79
0x00 0x49 0x64 0x00  0x53 0x61 0x6D 0x70
0x6C 0x65 0x45 0x6E  0x61 0x62 0x6C 0x65
0x64 0x50 0x72 0x6F  0x70 0x65 0x72 0x74
0x79 0x00 0x53 0x61  0x6D 0x70 0x6C 0x65
0x49 0x6E 0x74 0x65  0x67 0x65 0x72 0x50
0x72 0x6F 0x70 0x65  0x72 0x74 0x79 0x00
0x41 0x6E 0x6F 0x74  0x68 0x65 0x72 0x42
0x6F 0x6F 0x6C 0x65  0x61 0x6E 0x00 0x4C
0x69 0x6E 0x6B 0x53  0x74 0x61 0x74 0x75
0x73 0x00 0x4C 0x69  0x6E 0x6B 0x44 0x6F
0x77 0x6E 0x00 0x4C  0x69 0x6E 0x6B 0x55
0x70 0x00 0x4E 0x6F  0x4C 0x69 0x6E 0x6B
0x00 0x18 0x43 0x6F  0x70 0x79 0x72 0x69
0x67 0x68 0x74 0x20  0x28 0x63 0x29 0x20
0x32 0x30 0x31 0x38  0x20 0x44 0x4D 0x54
0x46 0x00
```

Figure 6 – DummySimple dictionary – binary form

8.6.2 Example encoding

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

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

```
<Item name=”DummySimple” type=”set”>
   <Item name=”ChildArrayProperty” type=”array”>
      <Item name=”array element 0”>
         <Item name=”AnotherBoolean” type=”boolean” value=”true”/>
         <Item name=”LinkStatus” type=”enum” enumtype=”String”>
            <Enumeration value=”NoLink”/>
         </Item>
      </Item>
      <Item name=”array element 1”>
         <Item name=”LinkStatus” type=”enum” enumtype=”String”>
```

```
                <Enumeration value="LinkDown"/>
              </Item>
           </Item>
        </Item>
        <Item name="Id" type="string" value="Dummy ID"/>
        <Item name="SampleIntegerProperty" type="number" value="12"/>
</Item>
```

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.

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

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:

```
<Item type="set" seqno="0">
   <Item type="array" seqno="0">
      <Item seqno="0">
         <Item type="boolean" value="true" seqno="0"/>
         <Item type="enum" enumtype="String" seqno="2">
            <Enumeration seqno="4"/>
         </Item>
      </Item>
      <Item seqno="2">
         <Item type="enum" enumtype="String" seqno="2">
            <Enumeration seqno="0"/>
          </Item>
       </Item>
   </Item>
   <Item type="string" value="Dummy ID" seqno="2"/>
   <Item type="integer" value="12" seqno="6"/>
</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.

```
{0x01 0x00, set, [length placeholder], value={count=3,
   {0x01 0x00, array, [length placeholder], value={count=2,
      {0x01 0x00, set, [length placeholder], value={count=2,
         {0x01 0x00, boolean, [length placeholder], value=true}
         {0x01 0x02, enum, [length placeholder], value=2}
      }}
      {0x01 0x02, set, [length placeholder], value={count=1,
         {0x01 0x02, enum, [length placeholder], value=0}
```

```
      }}
   }}
   {0x01 0x02, string, [length placeholder], value="Dummy ID"}
   {0x01 0x06, integer, [length placeholder], value=12}
}}
```

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

```
{0x01 0x00, 0x00, [length placeholder], {0x01 0x03,
   {0x01 0x00, 0x10, [length placeholder], {0x01 0x02,
      {0x01 0x00, 0x00, [length placeholder], {0x01 0x02,
         {0x01 0x00, 0x70, [length placeholder], 0xFF}
         {0x01 0x02, 0x40, [length placeholder], 0x01 0x02}
      }}
      {0x01 0x02, 0x00, [length placeholder], {0x01 0x01,
         {0x01 0x02, 0x40, [length placeholder], 0x01 0x00}
      }}
   }}
   {0x01 0x02, 0x50, [length placeholder],
      0x44 0x75 0x6D 0x6D 0x79 0x20 0x49 0x44 0x00}
   {0x01 0x06, 0x30, [length placeholder], 0x0C}
}}
```

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

```
{0x01 0x00, 0x00, [length placeholder], {0x01 0x03,
   {0x01 0x00, 0x10, [length placeholder], {0x01 0x02,
      {0x01 0x00, 0x00, [length placeholder], {0x01 0x02,
         {0x01 0x00, 0x70, 0x01 0x01, 0xFF}
         {0x01 0x02, 0x40, 0x01 0x02, 0x01 0x02}
      }}
      {0x01 0x02, 0x00, [length placeholder], {0x01 0x01,
         {0x01 0x02, 0x40, 0x01 0x02, 0x01 0x00}
      }}
   }}
   {0x01 0x02, 0x50, 0x01 0x09,
      0x44 0x75 0x6D 0x6D 0x79 0x20 0x49 0x44 0x00}
   {0x01 0x06, 0x30, 0x01 0x01, 0x0C}
}}
```

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

```
{0x01 0x00, 0x00, [length placeholder], {0x01 0x03,
   {0x01 0x00, 0x10, [length placeholder], {0x01 0x02,
      {0x00, 0x00, 0x01 0x0F, {0x01 0x02,
         {0x01 0x00, 0x07, 0x01 0x01, 0xFF}
         {0x01 0x20, 0x04, 0x01 0x02, 0x01 0x02}
      }}
      {0x01 0x02, 0x00, 0x01 0x09, {0x01 0x01,
         {0x01 0x02, 0x40, 0x01 0x02, 0x01 0x00}
      }}
   }}
   {0x01 0x02, 0x50, 0x01 0x09,
```

```
      0x44 0x75 0x6D 0x6D 0x79 0x20 0x49 0x44 0x00}
   {0x01 0x06, 0x30, 0x01 0x01, 0x0C}
}}
```

Next, the array itself:

```
{0x01 0x00, 0x00, [length placeholder], {0x01 0x03,
   {0x01 0x00, 0x10, 0x01 0x24, {0x01 0x02,
      {0x01 0x00, 0x00, 0x01 0x0F, {0x01 0x02,
         {0x01 0x00, 0x70, 0x01 0x01, 0xFF}
         {0x01 0x02, 0x40, 0x01 0x02, 0x01 0x02}
      }}
      {0x01 0x02, 0x00, 0x01 0x09, {0x01 0x01,
         {0x01 0x02, 0x40, 0x01 0x02, 0x01 0x00}
      }}
   }}
   {0x01 0x02, 0x50, 0x01 0x09,
      0x44 0x75 0x6D 0x6D 0x79 0x20 0x49 0x44 0x00}
   {0x01 0x06, 0x30, 0x01 0x01, 0x0C}
}}
```

Finally, the outermost set:

```
{0x01 0x00, 0x00, 0x01 0x3F, {0x01 0x03,
   {0x01 0x00, 0x10, 0x01 0x24, {0x01 0x02,
      {0x01 0x00, 0x00, 0x01 0x0F, {0x01 0x02,
         {0x01 0x00, 0x70, 0x01 0x01, 0xFF}
         {0x01 0x02, 0x40, 0x01 0x02, 0x01 0x02}
      }}
      {0x01 0x02, 0x00, 0x01 0x09, {0x01 0x01,
         {0x01 0x02, 0x40, 0x01 0x02, 0x01 0x00}
      }}
   }}
   {0x01 0x02, 0x50, 0x01 0x09,
      0x44 0x75 0x6D 0x6D 0x79 0x20 0x49 0x44 0x00}
   {0x01 0x06, 0x30, 0x01 0x01, 0x0C}
}}
```

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

```
0x01 0x00 0x00 0x01 : 0x3F 0x01 0x03 0x01
0x00 0x10 0x01 0x24 : 0x01 0x02 0x01 0x00
0x00 0x01 0x0F 0x01 : 0x02 0x01 0x00 0x70
0x01 0x01 0xFF 0x01 : 0x02 0x40 0x01 0x02
0x01 0x02 0x01 0x02 : 0x00 0x01 0x09 0x01
0x01 0x01 0x02 0x40 : 0x01 0x02 0x01 0x00
0x01 0x02 0x50 0x01 : 0x09 0x44 0x75 0x6D
0x6D 0x79 0x20 0x49 : 0x44 0x00 0x01 0x06
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:

```
0x01 0x00 0x00 0x01 : 0x3F 0x01 0x03 0x01
0x00 0x10 0x01 0x24 : 0x01 0x02 0x01 0x00
0x00 0x01 0x0F 0x01 : 0x02 0x01 0x00 0x70
0x01 0x01 0xFF 0x01 : 0x02 0x40 0x01 0x02
0x01 0x02 0x01 0x02 : 0x00 0x01 0x09 0x01
```

```
0x01 0x01 0x02 0x40 : 0x01 0x02 0x01 0x00
0x01 0x02 0x50 0x01 : 0x09 0x44 0x75 0x6D
0x6D 0x79 0x20 0x49 : 0x44 0x00 0x01 0x06
0x30 0x01 0x01 0x0C
```

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

```
{S=0x01 0x00, F=0x00, L=0x01 0x3F, V={0x01 0x03,
   {S=0x01 0x00, F=0x10, L=0x01 0x24, V={0x01 0x02,
      {S=0x01 0x00, F=0x00, L=0x01 0x0F, V={0x01 0x02,
         {S=0x01 0x00, F=0x70, L=0x01 0x01, V=0xFF}
         {S=0x01 0x02, F=0x40, L=0x01 0x02, V=0x01 0x02}
      }}
      {S=0x01 0x02, F=0x00, L=0x01 0x09, V={0x01 0x01,
         {S=0x01 0x02, F=0x40, L=0x01 0x02, V=0x01 0x00}
      }}
   }}
   {S=0x01 0x02, F=0x50, L=0x01 0x09,
      V=0x44 0x75 0x6D 0x6D 0x79 0x20 0x49 0x44 0x00}
   {0x01 S=0x06, F=0x30, L=0x01 0x01, V=0x0C}
}}
```

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

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

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

```
{S=0x01 0x00, set, V={
   {S=0x01 0x00, array, V={
      {S=0x01 0x00, set, V={
         {S=0x01 0x00, boolean, V=0xFF}
         {S=0x01 0x02, enum, V=0x01 0x02}
      }}
      {S=0x01 0x02, set, V={
         {S=0x01 0x02, enum, V=0x01 0x00}
      }}
   }}
   {S=0x01 0x02, string, V=0x44 0x75 0x6D 0x6D 0x79 0x20 0x49 0x44 0x00}
   {S=0x01 0x06, integer, V=0x0C}
}}
```

We now decode value data:

```
{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}
}}
```

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

```
{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}
}}
```

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

```
{“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}
}}
```

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):

```
{
   “DummySimple” : {
      “ChildArrayProperty” : [
         {
            “AnotherBoolean” : true,
            “LinkStatus” : “NoLink”
         },
         {
            “LinkStatus” : “LinkDown”
         }
```

```
        ],
        "Id" : "Dummy ID",
        "SampleIntegerProperty" : 12
    }
}
```

8.7 BEJ locators

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

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

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

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

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

0, 0, 3, 1

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

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

9 Operational behaviors

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

9.1 Initialization (MC perspective)

The following clauses present initialization of RDE Devices with MCs.

9.1.1 Sample initialization ladder diagram

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

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

After discovery comes the RDE registration process. It consists of two parts, PDR retrieval and dictionary retrieval. To retrieve the RDE PDRs, the MC utilizes the PLDM for Platform Monitoring and Control FindPDR command to locate PDRs that are specific to RDE[4]. For each such PDR located, the MC then retrieves it via one or more message sequences in the PLDM for Platform Monitoring and Control GetPDR command.

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

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

Figure 7 – Example Initialization ladder diagram

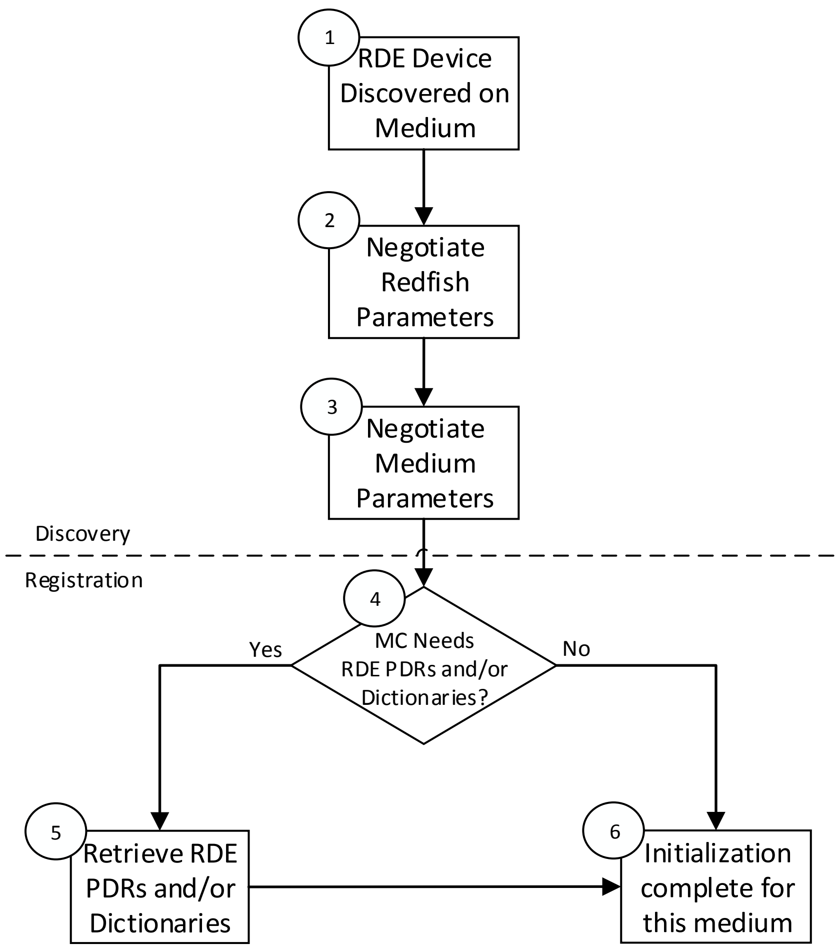
9.1.2 Initialization workflow diagram

Table 42 details the information presented visually in Figure 8.

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

2281 

2282 **Figure 8 – Typical RDE Device discovery and registration**

9.2 Operation/Task lifecycle

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

9.2.1 Example Operation command sequence diagrams

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

9.2.1.1 Simple read Operation ladder diagram

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

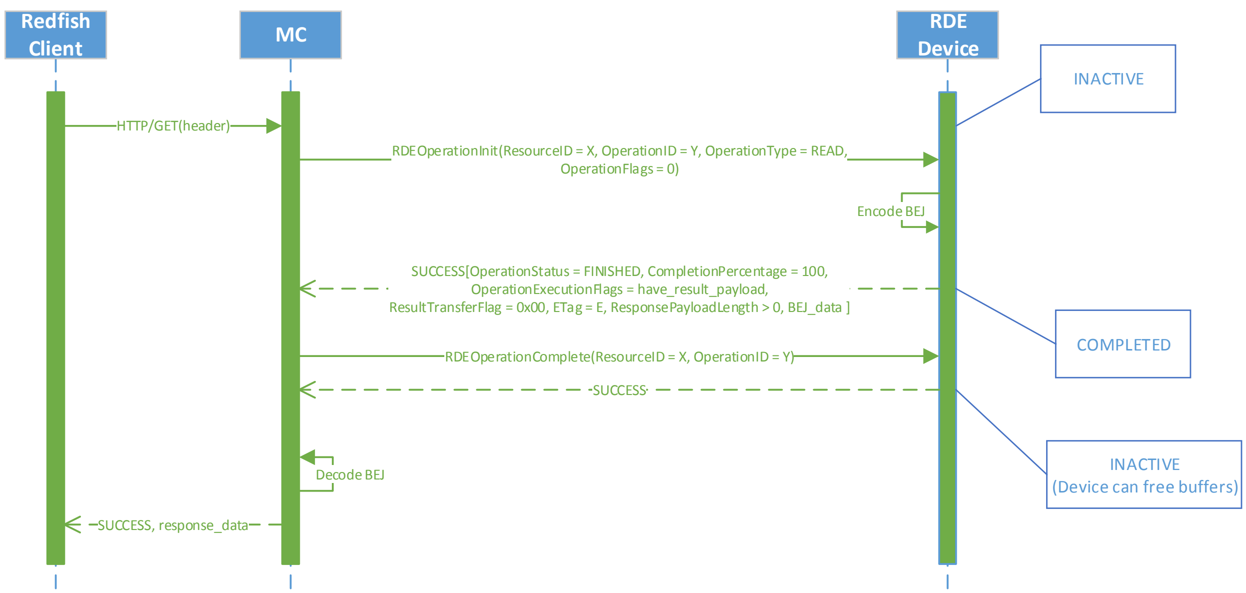
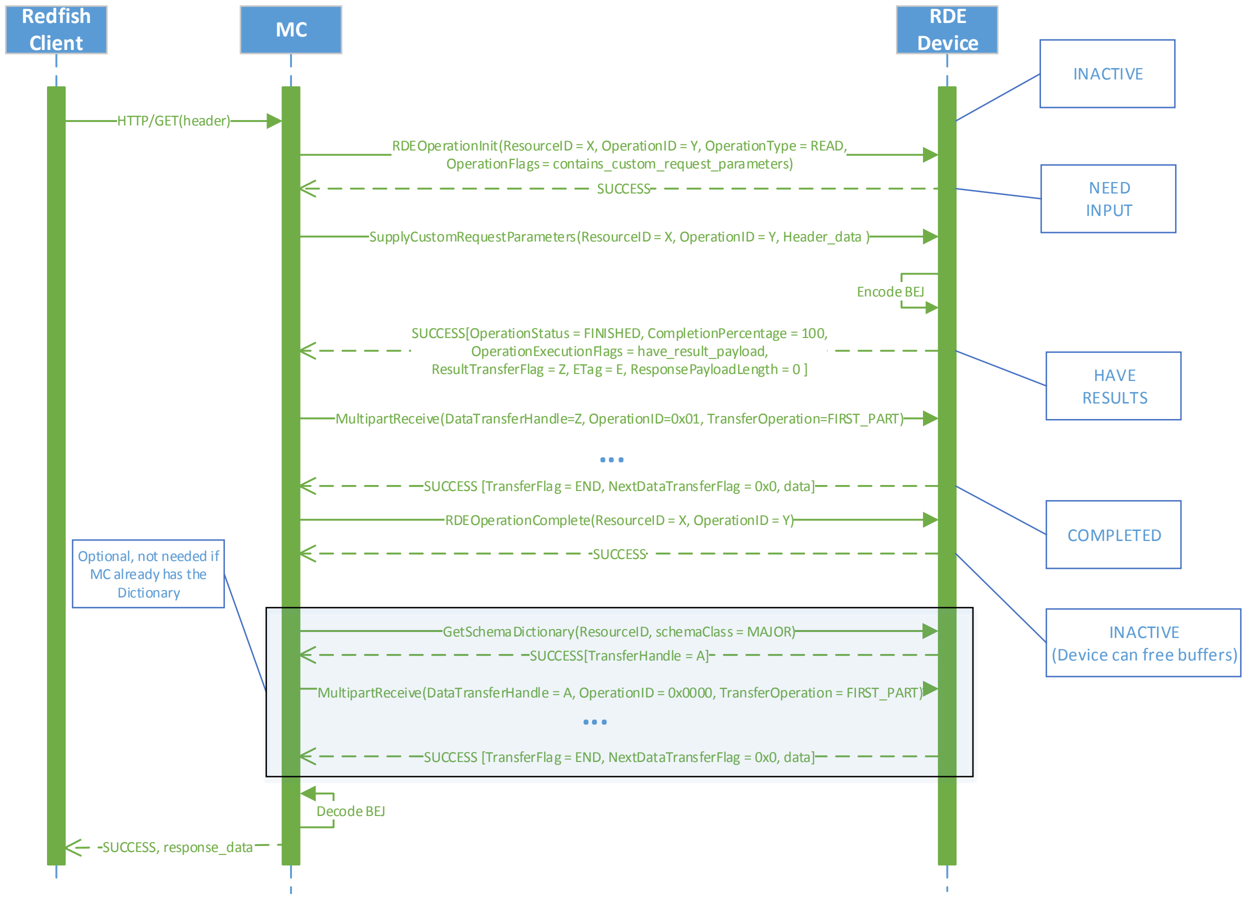


Figure 9 – Simple read Operation ladder diagram

9.2.1.2 Complex read Operation diagram

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

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



2329
2330 **Figure 10 – Complex Read Operation ladder diagram**

9.2.1.3 Write (update) Operation ladder diagram

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

The MC uses the MultipartSend command to transfer the encoded payload to the RDE Device in one or more chunks. The contains_request_parameters Operation flag is not set, so the RDE Device will not expect supplemental parameters as part of this Operation. Having everything it needs to execute, the RDE Device moves to the TRIGGERED state. The MC now sends the RDEOperationStatus command to the RDE Device to have it execute the Operation. (In practice, the RDE Device is allowed to begin executing the Operation as soon as it has received the request payload, so it may choose not to wait for the RDEOperationStatus command to do so.) The RDE Device executes the Operation and sends the results to the MC as the response to the RDEOperationStatus command. As before, the MC finalizes the Operation via RDEOperationComplete and then sends the results back to the client.

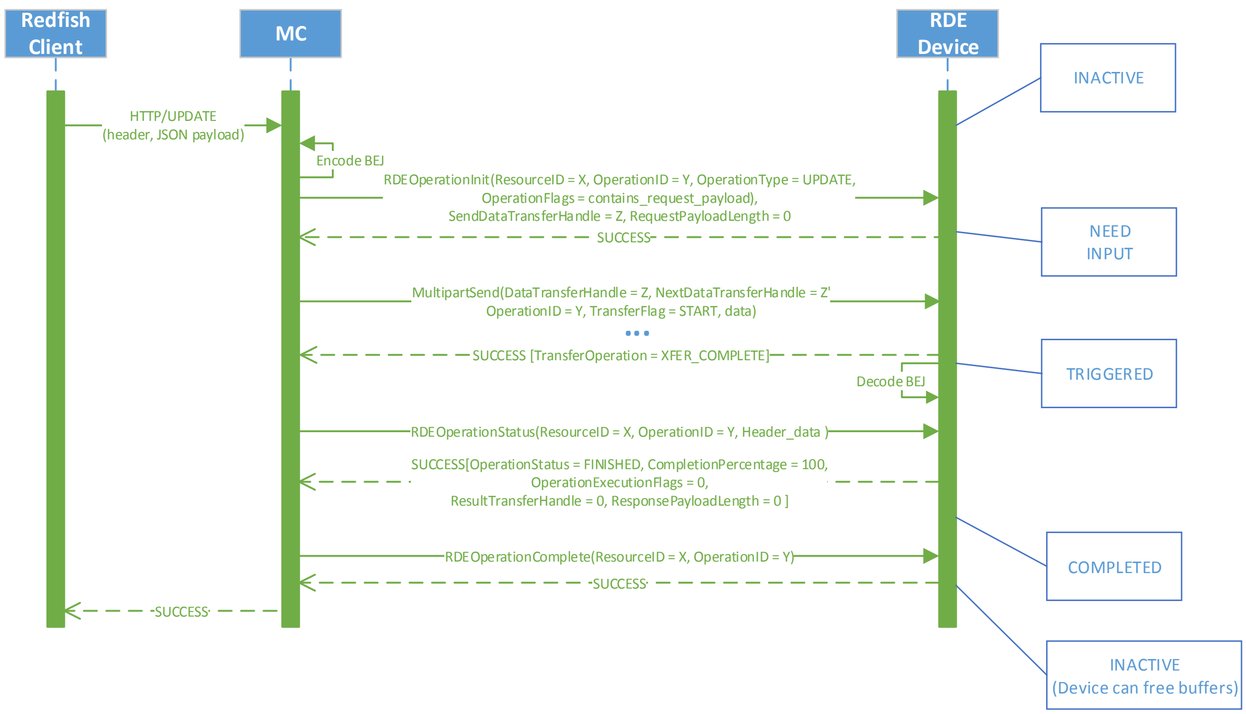


Figure 11 – Write Operation ladder diagram

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

Figure 12 presents the ladder diagram for a write Operation that spawns a long-running Task. As with the previous case, the Operation begins when the Redfish client sends an UPDATE request over an HTTP connection to the MC, and the MC decodes the URI targeted by the UDPATE Operation to pin it down to a specific resource and PDR. Before it can send the RDEOperationInit command to the RDE Device that owns the PDR, however, the MC must perform a BEJ encoding of the JSON payload it received from the

Redfish client. Unlike the previous example, the BEJ encoded payload here is small enough to fit in the maximum transfer chunk, so the MC inlines it into the RDEOperationInit request command. Again, the contains_request_parameters Operation flag is not set, so the RDE Device will not expect supplemental parameters as part of this Operation.

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

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

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

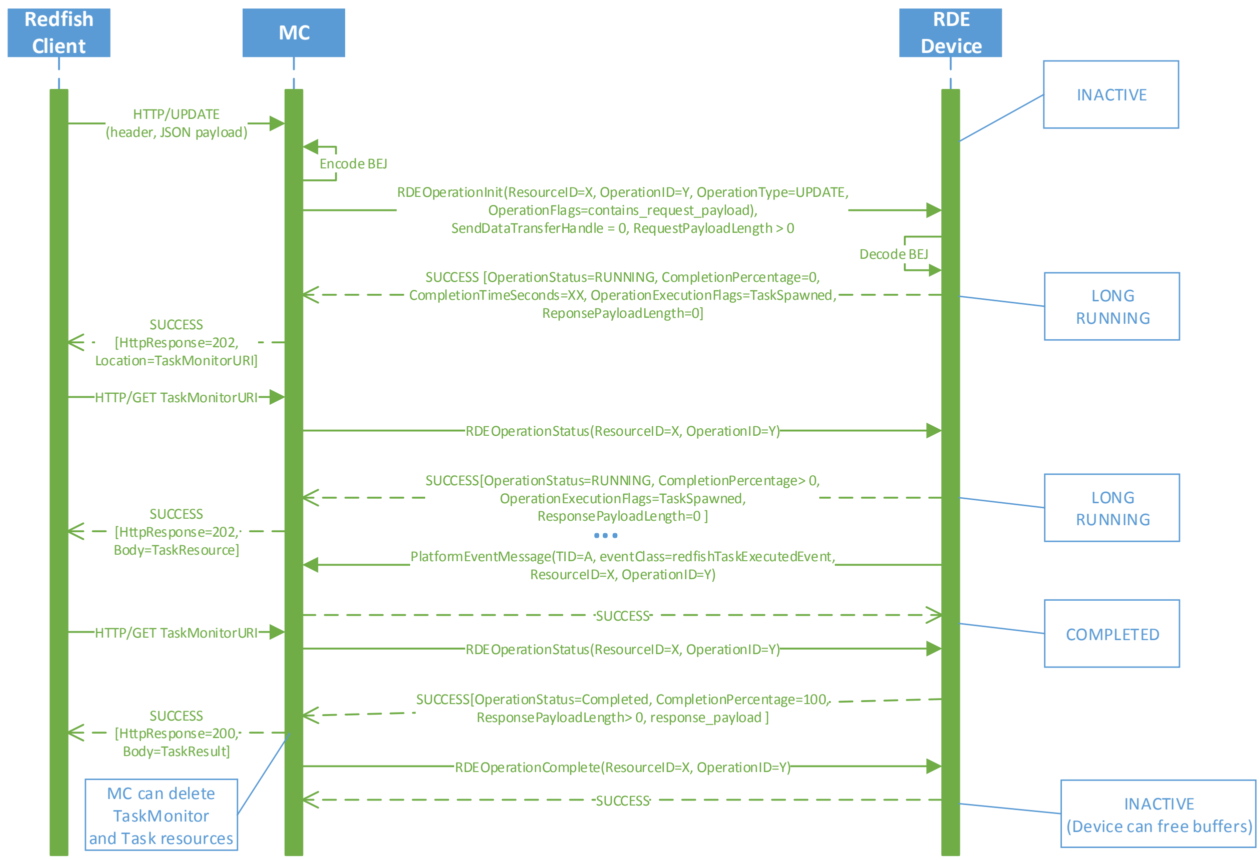


Figure 12 – Write Operation with long-running Task ladder diagram

9.2.2 Operation/Task overview workflow diagrams (Operation perspective)

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

9.2.2.1 Operation overview workflow diagram

Table 43 details the information presented visually in Figure 13.

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	Unconditional	3

Step	Description	Condition	Next Step
	command (see clause 11.1), the MC may skip this step if the read does not span multiple resources (such as through the $expand request header)		
3 – INITIALIZE_OP	The MC checks the HTTP/HTTPS operation to see if it contains JSON payload data to be transferred to the RDE Device. If so, it performs a BEJ encoding of this data. It then uses the RDEOperationInit command to begin the Operation with the RDE Device	Unconditional	4
4 – SEND_PAYLOAD_ CHK	If the RDE Operation contains BEJ payload data, it needs to be sent to the RDE Device. The payload data may be inlined in the RDEOperationInit request message if the resulting message fits within the negotiated transfer chunk limit.	If the Operation contains a non-inlined payload (that did not fit in the RDEOperationInit request message)	5
		Otherwise	6
5 – SEND_PAYLOAD	The MC uses the 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_C HK	If the RDE Operation contains uncommon request parameters or headers that need to be transferred to the RDE Device, they need to be sent to the RDE Device.	If the Operation contains supplemental request parameters	7
		Otherwise	8
7 – SEND_PARAMS	The MC uses the SupplyCustomRequestParameters command to submit the supplemental request parameters to the RDE Device	Unconditional	8
8 – TRIGGERED	The RDE Device begins executing the Operation as soon as it has all the information it needs for it	Unconditional	9
9 – COMPLETION_CH K	The RDE Device must respond to the triggering command (that provided the last bit of information needed to execute the Operation or a follow-up call to RDEOperationStatus if the last data was sent via 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 sublifecycle.	Once the Task finishes executing	11

Step	Description	Condition	Next Step
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: • 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 – RCV_PARAMS_CHK	The MC checks to see if the Operation result contains supplemental response parameters	If the Operation contains response parameters	14
		Otherwise	15
14 – RCV_PARAMS	The MC uses the RetrieveCustomResponseParameters command to obtain the supplemental response parameters. NOTE The transfer of a 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.	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	Read operation and mismatched	2

Step	Description	Condition	Next Step
	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 is has the capability to read a resource atomically in its response to the NegotiateRedfishParameters command (see clause 11.1).	ETags and retry count not exceeded	
		Not a read, no ETag collected, the ETags match, or retry count exceeded	n/a: Done

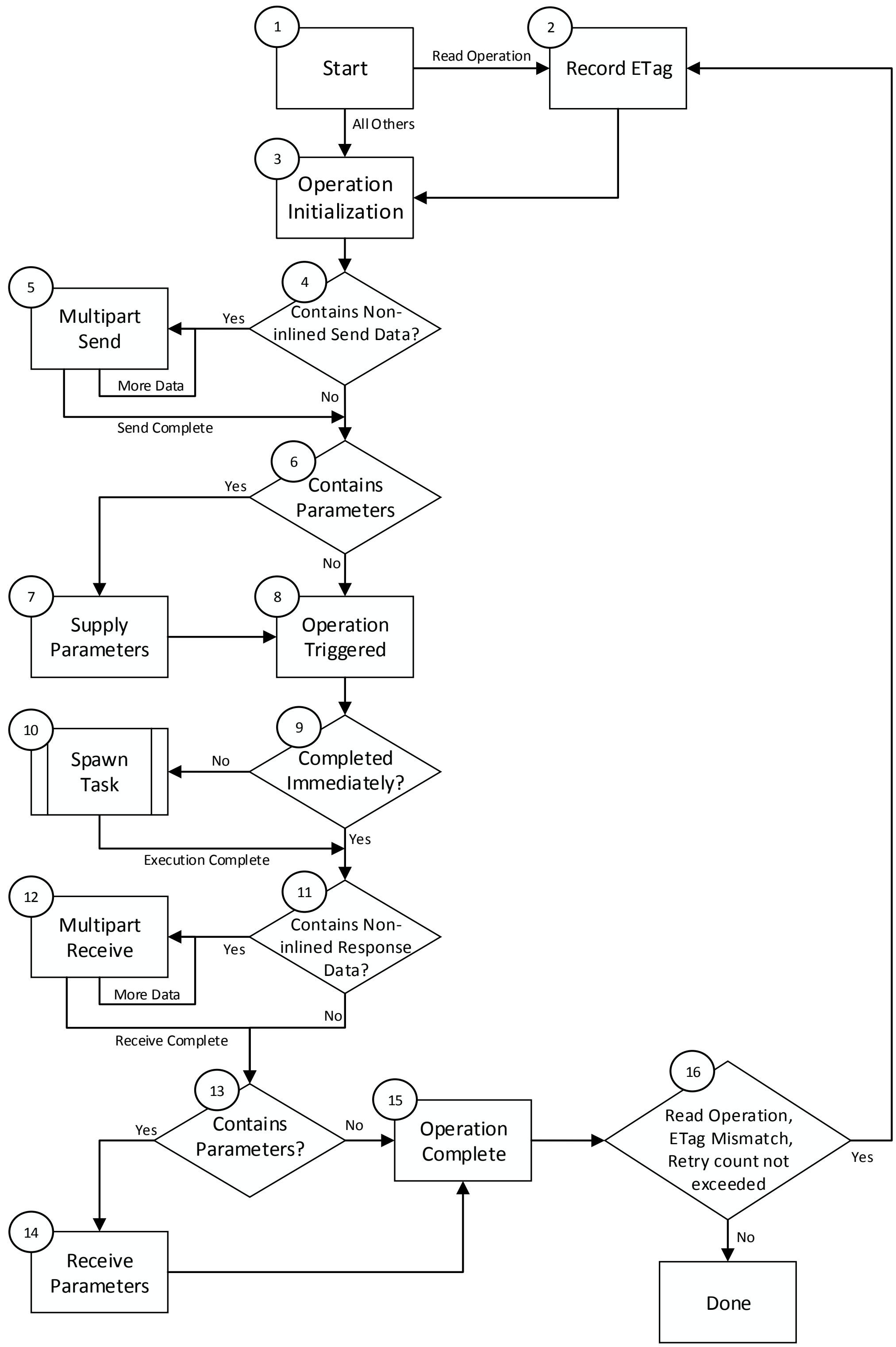


Figure 13 – RDE Operation lifecycle overview (holistic perspective)

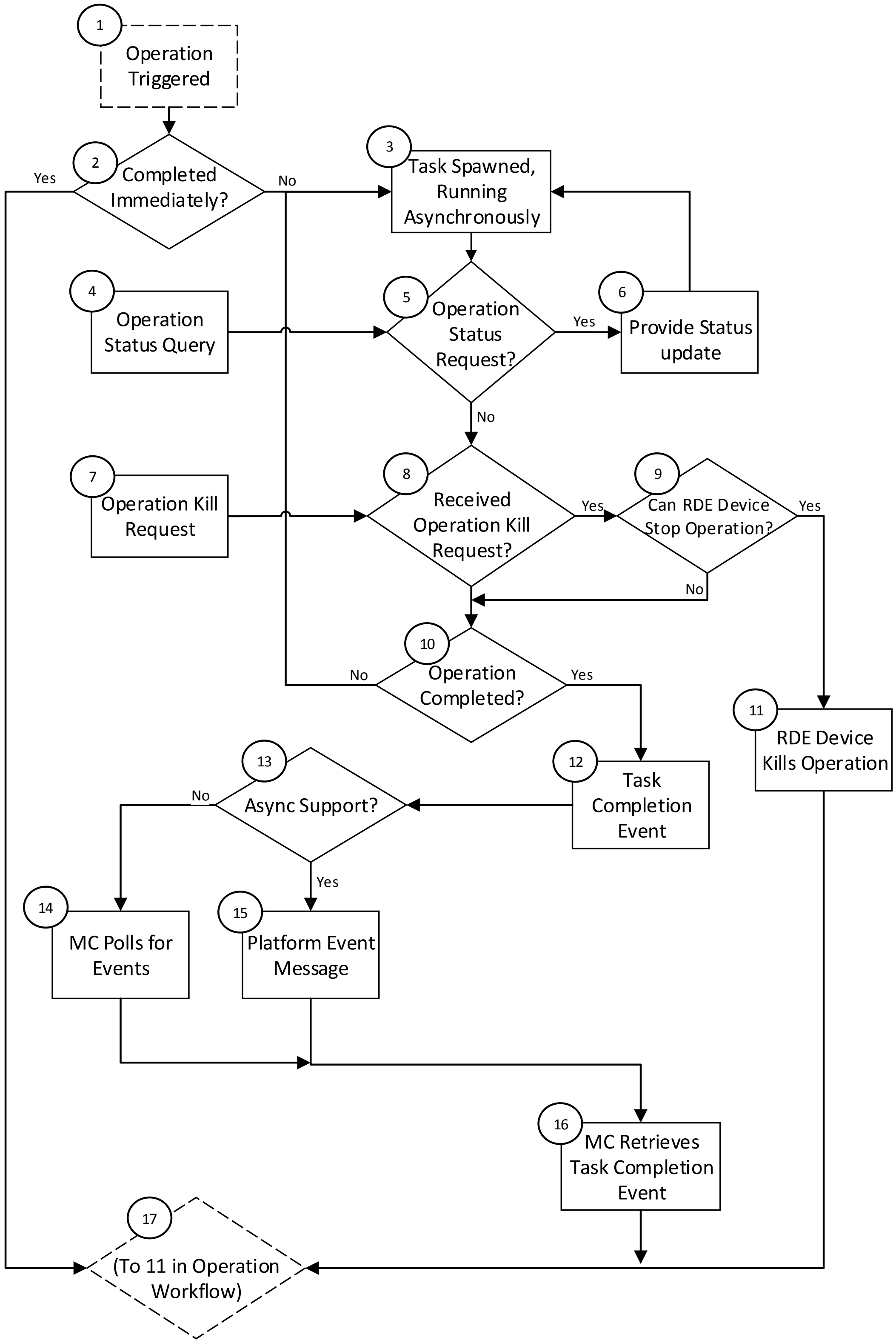
9.2.2.2 Task overview workflow diagram

Table 44 details the information presented visually in Figure 14.

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_STATUS	The RDE Device sends a response to the RDEOperationStatus command to provide a status update	Unconditional	3
7 – REQ_KILL	The MC may issue an RDEOperationKill command at any time to the RDE Device	Unconditional	8
8 –KILL_CHK	The RDE Device must be ready to respond to an RDEOperationKill command while running a Task asynchronously	Kill request received	9
		No kill request received	10
9 – PROCESS_KILL	If the RDE Device receives a kill request, it may or may not be able to abort the Task. This is an RDE Device-specific decision about whether the Task has crossed a critical boundary and must be completed	RDE Device cannot stop the Task	10
		RDE Device can stop the Task	11
10 – ASYNC_EXECUTE_FINISHED_CHK	The RDE Device should eventually complete the Task	If the Task has been completed	12
		If the Task has not been completed	3
11 – PERFORM_ABORT	The RDE Device aborts the Task in response to a request from the MC	Unconditional	17

Current Step	Description	Condition	Next Step
12 – COMPLETION_EVENT	After the Task is complete, the RDE Device generates a Task Completion Event	Unconditional	13
13 – ASYNC_CHK	The mechanism by which the Task completion Event reaches the MC depends on how the MC configured the RDE Device for Events via the PLDM for Platform Monitoring and Control SetEventReceiver command	Asynchronous Events	14
		Polled Events	15
14 – PEM_POLL	The MC uses the PollForPlatformEventMessage command to check for Events and finds the Task Completion Event	Unconditional	16
15 – PEM_SEND	The RDE Devices sends the Task Completion Event to the MC asynchronously via the PlatformEventMessage command	Unconditional	16
16 – GET_TASK_FOLLOWUP	After receiving the Task completion Event, the MC uses the RDEOperationStatus command to retrieve the outcome of the Task's execution	Unconditional	17
17 – TASK_DONE	The MC checks the response message to the RDEOperationStatus command to see if there is a response payload (This corresponds to Step 11 in Table 43.)	See Step 11 in Table 45	See Step 11 in Table 45



2400

2401 **Figure 14 – RDE Task lifecycle overview (holistic perspective)**

9.2.3 RDE Operation state machine (RDE Device perspective)

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

9.2.3.1 State definitions

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

- INACTIVE
  - INACTIVE is the default Operation state in which the RDE Device shall start after initialization. In this state, the RDE Device is not processing an Operation as it has not received an RDEOperationInit command from the MC
- NEED_INPUT
  - After receiving the RDEOperationInit command, the RDE Device moves to this state if it is expecting additional Operation-specific parameters or a payload that was not inlined in the RDEOperationInit command
- TRIGGERED
  - Once the RDE Device receives everything it needs to execute an Operation, it begins executing it immediately. If the triggering command – the command that supplied the last bit of data needed to execute the Operation – was RDEOperationInit or SupplyCustomRequestParameters, the response message to the triggering command reflects the initial results for the Operation. However, if the triggering command was a MultipartSend, initial results are deferred until the MC invokes the RDEOperationStatus command. This state captures the case where the Operation was triggered by a MultipartSend and the MC has not yet sent an RDEOperationStatus command to get initial results. In this state, the RDE Device may execute the Operation; alternatively, it may wait to receive RDEOperationStatus to begin execution.
- TASK_RUNNING
  - If the RDE Device cannot complete the Operation within the timeframe needed for the response to the command that triggered it, the RDE Device spawns a Task in which to execute the Operation asynchronously
- HAVE_RESULTS
  - When execution of the Operation produces a response parameters or a response payload that does not fit in the response message for the command that triggered the Operation (or detected its completion, if a Task was spawned or if there was a payload but no custom request parameters), the RDE Device remains in this state until the MC has collected all of these results
- COMPLETED
  - The RDE Device has completed processing of the Operation and awaits acknowledgment from the MC that it has received any Operation response data. This acknowledgment is done by the MC issuing the RDEOperationComplete command. When the RDE Device receives this command, it may discard any internal records or state it has maintained for the Operation
- FAILED
  - The MC has explicitly killed the Operation or an error prevented execution of the Operation
- ABANDONED
  - If MC fails to progress the Operation through this state machine, the RDE Device may abort the Operation and mark it as abandoned

9.2.3.2 Operation lifecycle state machine

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

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, Operation ID has MSBit clear (indicating that the MC is attempting to initiate an Operation with an ID reserved for the RDE Device)	ERROR_INVALID_DATA	INACTIVE
	RDEOperationInit, request contains any other error	Various, depending on the specific error encountered	INACTIVE
	RDEOperationStatus	OPERATION_INACTIVE	INACTIVE
	RDEOperationInit; - valid request - Operation Flags indicate request non-inlined payload or parameters to	Success	NEED_INPUT

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

Current State	Trigger	Response	Next State
	transfer chunk - 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; - run_to_completion flag set - discard_record flag not set	ERROR_INVALID_DATA	NEED_INPUT
	RDEOperationKill; - both run_to_completion and discard_record flags both set	ERROR_UNEXPECTED (can't run to completion without further input from MC, so the request is contradictory)	NEED_INPUT
	RDEOperationStatus	OPERATION_NEED_INPUT	NEED_INPUT
	MultipartSend; - data inlined or Operation flags indicate no payload data	ERROR_UNEXPECTED	NEED_INPUT
	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	Success	NEED_INPUT

Current State	Trigger	Response	Next State
	after this chunk		
	MultipartSend; - no more data to be sent from the MC to the RDE Device after this chunk - RDEOperationInit request flags indicated supplemental parameters needed - params not yet sent	Success	NEED_INPUT
	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	NEED_INPUT
	SupplyCustomRequestParameters; - Operation includes unsupported ETag operation or query option	ERROR_UNSUPPORTED	FAILED
	SupplyCustomRequestParameters; - Operation flags indicated supplemental parameters not needed or payload data remaining to be sent	ERROR_UNEXPECTED	NEED_INPUT
	SupplyCustomRequestParameters; - no payload data remaining to be sent - ETagOperation is ETAG_IF_MATCH and no ETag matches or ETagOperation is ETAG_IF_NONE_MATCH and an ETAG matches	ERROR_ETAG_MATCH	FAILED
	SupplyCustomRequestParameters; - request contains unsupported 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	Success	LONG_RUNNING

Current State	Trigger	Response	Next State
	to be sent - RDE Device cannot complete Operation within PT1		
	SupplyCustomRequestParameters; - no payload data remaining to be sent - RDE Device completes Operation within PT1 - response flags indicate response parameters or a non-inlined response payload to be retrieved from RDE Device	Success	HAVE_RESULTS
	SupplyCustomRequestParameters; - no payload data remaining to be sent - RDE Device completes Operation within PT1 - no payload to be retrieved from RDE Device or response payload fits within response message such that total response message size is within negotiated maximum transfer chunk - no response parameters	Success	COMPLETED
	MultipartReceive, RDEOperationComplete	ERROR_UNEXPECTED	NEED_INPUT
	Any other Operation command	ERROR	NEED_INPUT
2 - TRIGGERED	RDEOperationInit, same rdeOpID	ERROR_OPERATION_EXISTS; no disruption to existing Operation	TRIGGERED
	RDEOperationInit, different rdeOpID	Success or ERROR_CANNOT_CREATE_OPERATION, depending on whether the 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 - run_to_completion flag set - discard_record flag not set	ERROR_INVALID_DATA	TRIGGERED

Current State	Trigger	Response	Next State
	RDEOperationKill; - Operation executing; Operation can be killed - neither run_to_completion nor discard_record flag set	Success	FAILED
	RDEOperationKill - Operation executing - Operation can be killed - run_to_completion flag not set - discard_record flag set	Success	INACTIVE
	RDEOperationKill - Operation executing - Operation can be killed - both run_to_completion and discard_record flags set	ERROR_UNEXPECTED (can't run to completion without further input from MC to move it to TASK_RUNNING, so the request is contradictory)	TRIGGERED
	RDEOperationKill - Operation executing - Operation cannot be killed or Operation execution finished - any combination of run_to_completion and discard_record flags set	ERROR_OPERATION_UNKILLAB LE	TRIGGERED
	RDEOperationStatus; - RDE Device cannot complete Operation within PT1	OPERATION_TASK_RUNNING	TASK_RUNNING
	RDEOperationStatus; - RDE Device completes Operation within PT1 - payload to be retrieved from RDE Device or response parameters present	Success	HAVE_RESULTS
	RDEOperationStatus; - RDE Device completes 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, RetrieveCustomResponseParamete	ERROR_UNEXECTED	TRIGGERED

Current State	Trigger	Response	Next State
	rs, RDEOperationComplete		
	Any other Operation command	ERROR	TRIGGERED
3 - TASK_RUNNING	RDEOperationInit, same rdeOpID	ERROR_OPERATION_EXISTS; no disruption to existing Operation	TASK_RUNNING
	RDEOperationInit, different rdeOpID	Success or ERROR_CANNOT_CREATE_OPERATION, depending on whether the RDE Device has another slot to execute an Operation	The new Operation is tracked in a separate copy of the state machine; this Operation remains in TASK_RUNNING
	Error occurs in processing of Operation	None	FAILED
	RDEOperationKill - run_to_completion flag set - discard_record flag not set	ERROR_INVALID_DATA	TASK_RUNNING
	RDEOperationKill; - Operation can be aborted - neither run_to_completion nor discard_record flag set	Success	FAILED
	RDEOperationKill - Operation executing - Operation can be killed - run_to_completion flag not set - discard_record flag set	Success	INACTIVE
	RDEOperationKill - Operation executing - Operation can be killed - both run_to_completion and discard_record flags set	Success	TASK_RUNNING
	RDEOperationKill; - Operation cannot be aborted or has finished execution - any combination of run_to_completion and discard_record flags set	ERROR_OPERATION_UNKILLABLE	TASK RUNNING
	Execution finishes; - Operation not killed	Generate Task Completion Event (only once per Operation). Send to MC via PlatformEventMessage if MC configured the RDE Device to use asynchronous Events via SetEventReceiver; otherwise, MC will retrieve Event via	TASK_RUNNING

Current State	Trigger	Response	Next State
		PollForPlatformEventMessage. See Event lifecycle in clause 9.3 for further details	
	Execution finishes; - Operation killed	None	INACTIVE
	Execution finished; - Task Completion Event received by MC; - $T_{abandon}$ timeout waiting for RDEOperationStatus command	None	ABANDONED
	RDEOperationStatus; - execution not yet finished	OPERATION_TASK_RUNNING	TASK RUNNING
	RDEOperationStatus; - execution finished - payload to be retrieved from RDE Device or response parameters present	OPERATION_HAVE_RESULTS	HAVE_RESULTS
	RDEOperationStatus; - execution finished - no payload or payload fits in response message such that total response message size is within negotiated maximum transfer chunk - no response parameters	OPERATION_COMPLETED	COMPLETED
	MultipartSend, MultipartReceive, RDEOperationComplete	ERROR_UNEXPECTED	TASK_RUNNING
	Any other Operation command	ERROR	TASK_RUNNING
4 - HAVE_RESULTS	RDEOperationInit, same rdeOpID	ERROR_OPERATION_EXISTS; no disruption to existing Operation	HAVE_RESULTS
	RDEOperationInit, different rdeOpID	Success or ERROR_CANNOT_CREATE_OPERATION, depending on whether the RDE Device has another slot to execute an Operation	The new Operation is tracked in a separate copy of the state machine; this Operation remains in HAVE_RESULTS
	RDEOperationKill - run_to_completion flag set - discard_record flag not set	ERROR_INVALID_DATA	HAVE_RESULTS
	RDEOperationKill; - any other combination of run_to_completion and discard_record flags set	ERROR_OPERATION_UNKILLABLE	HAVE_RESULTS

Current State	Trigger	Response	Next State
	RDEOperationStatus	OPERATION_HAVE_RESULTS	HAVE_RESULTS
	MultipartReceive; - MC aborts transfer	Do not send data; Success; Prepare to restart transfer with next MultipartReceive command	HAVE_RESULTS
	MultipartReceive; - transfer error	Error specific to type of transfer failure encountered	HAVE_RESULTS (MC may retry receive or abandon Operation)
	MultipartReceive; - more data to transfer from the RDE Device to the MC after this chunk	Send data; Success	HAVE_RESULTS
	MultipartReceive; - 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; - no more data to transfer from the RDE Device to the MC after this chunk - no response parameters present	Send data; Success	COMPLETED
	$T_{abandon}$ timeout waiting for 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	Success	COMPLETED

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

Current State	Trigger	Response	Next State
		ERROR_CANNOT_CREATE_OPERATION, depending on whether the RDE Device has another slot to execute an Operation	Operation is tracked in a separate copy of the state machine; this Operation remains in ABANDONED
	RDEOperationKill - run_to_completion flag set - discard_record flag not set	ERROR_INVALID_DATA	ABANDONED
	RDEOperationKill; - any other combination of run_to_completion and discard_record flags set	ERROR_OPERATION_ABANDONED	ABANDONED
	RDEOperationStatus	OPERATION_ABANDONED	ABANDONED
	RDEOperationComplete	Success	INACTIVE
	Any other Operation command	ERROR_OPERATION_ABANDONED	ABANDONED

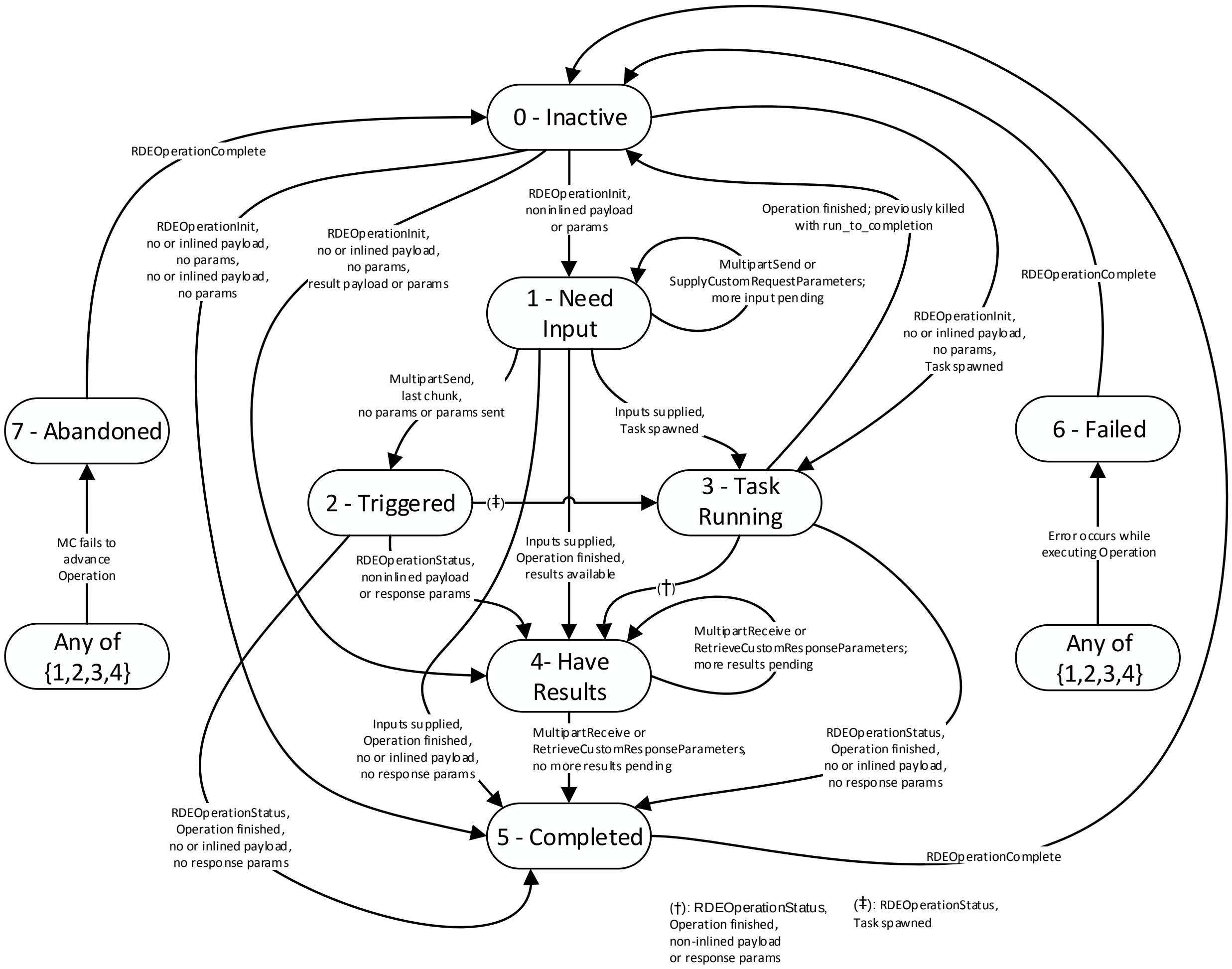


Figure 15 – Operation lifecycle state machine (RDE Device perspective)

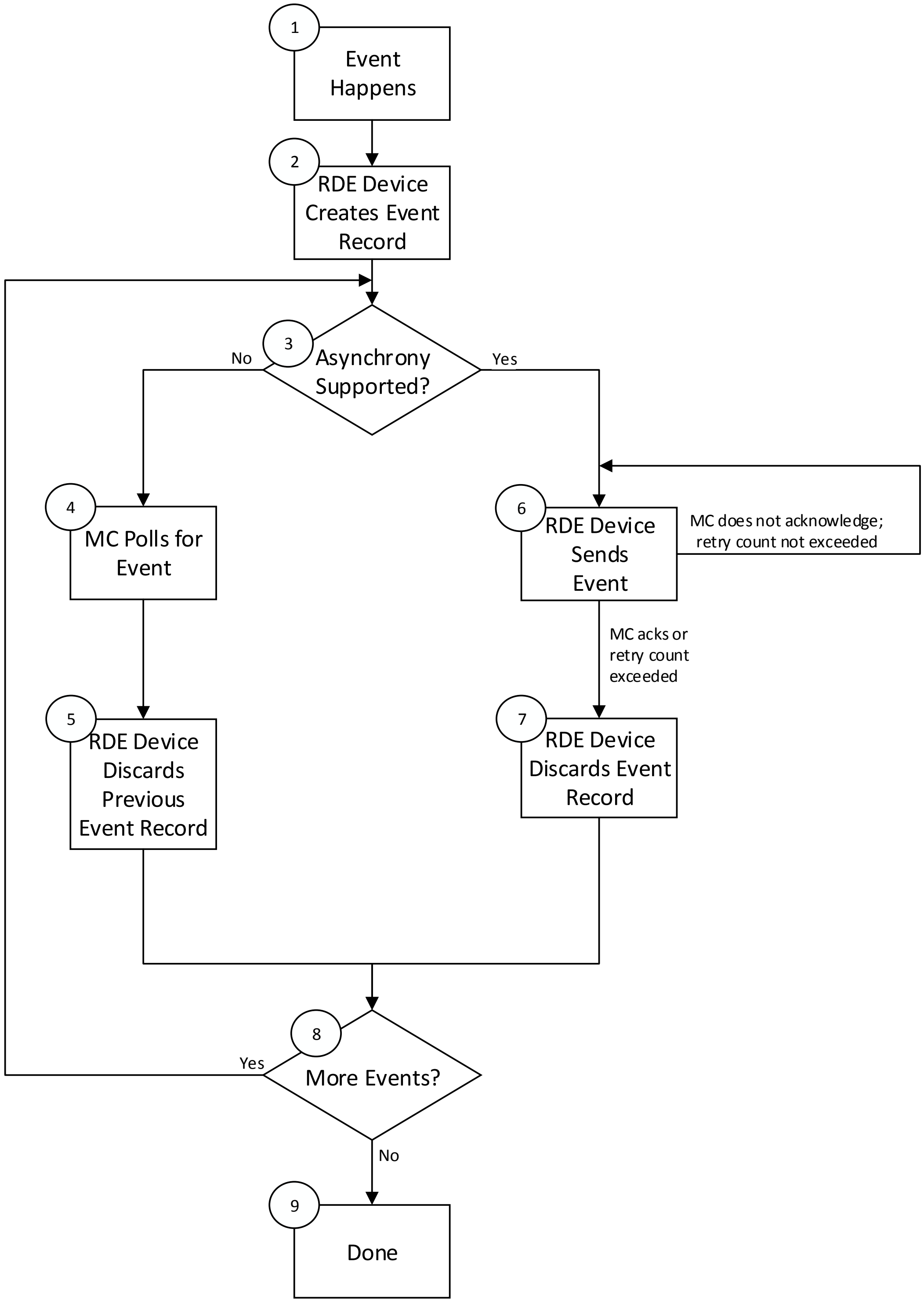
9.3 Event lifecycle

Table 46 describes the operating behavior for MCs and RDE Devices over the lifecycle of Events depicted visually in Figure 16. This sequence applies to both Task completion Events and schema-based Events. MC and RDE Device implementations of RDE shall comply with the sequences presented here.

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	-



2464

2465 **Figure 16 – Redfish event lifecycle overview**

10 PLDM commands for Redfish Device Enablement

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

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	Conditional[4]	Mandatory	MC	See 12.3
RDEOperationComplete	0x13	Mandatory	Mandatory	MC	See 12.4
RDEOperationStatus	0x14	Mandatory	Mandatory	MC	See 12.5
RDEOperationKill	0x15	Optional	Optional	MC	See 12.6
RDEOperationEnumerate	0x16	Mandatory	Optional	MC	See 12.7
Reserved	0x17-0x2F				
Multipart Transfer Commands					
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

Notes:

1) MultipartSend is required if the RDE Device intends to support write Operations.

2) SetEventReceiver is mandatory if the RDE Device intends to support asynchronous messaging for Events via PlatformEventMessage.

3) RDE Devices and MCs must support either PlatformEventMessage or PollForPlatformEventMessage in order to enable Event support.

4) SupplyCustomResponseParameter is required if the RDE Device ever sets the HaveCustomResponseParameters flag in the OperationExecutionFlags field of the response message for a triggering command.

11 PLDM for Redfish Device Enablement – Discovery and schema commands

This clause describes the commands that are used by RDE Devices and MCs that implement the discovery and schema management commands defined in this specification. The command codes for the PLDM messages are given in Table 47.

11.1 NegotiateRedfishParameters command format

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

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

2497 **Table 48 – NegotiateRedfishParameters command format**

Type	Request data
uint8	**MCConcurrencySupport** 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.
bitfield16	**MCFeatureSupport** Operations and functionality supported by the MC; for each, 1b indicates supported, 0b not: [15:8] - reserved [7] - events_supported; 1b = yes. Must be 1b if MC supports Redfish Events or Long-running Tasks. [6] - action_supported; 1b = yes [5] - replace_supported; 1b = yes [4] - update_supported; 1b = yes [3] - delete_supported; 1b = yes [2] - create_supported; 1b = yes [1] - read_supported; 1b = yes. All MCs that implement PLDM for Redfish Device Enablement shall support read Operations [0] - head_supported; 1b = yes
Type	**Response data**
enum8	**CompletionCode** value: { PLDM_BASE_CODES }
uint8	**DeviceConcurrencySupport** 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.
bitfield8	**DeviceCapabilitiesFlags** Capabilities for this RDE Device; for each, 1b indicates the RDE Device has the capability, 0b not: [7:2] - reserved [1] - expand_support: the RDE Device can process a $expand request query parameter (expressed via the **LinkExpand** field of the **SupplyCustomRequestParameters** command) [0] - atomic_resource_read: the RDE Device can respond to a read of an entire resource atomically, guaranteeing consistency of the read

Type	Response data (continued)
bitfield16	**DeviceFeatureSupport** Operations and functionality supported by this RDE Device; for each, 1b indicates supported, 0b not: [15:8] - reserved [7] - events_supported; 1b = yes. Must be 1b if RDE Device supports Redfish Events or Long-running Tasks. Shall match PLDM Event support indicated via support for PLDM for Platform Monitoring and Control (DSP0248) SetEventReceiver command [6] - action_supported; 1b = yes [5] - replace_supported; 1b = yes [4] - update_supported; 1b = yes [3] - delete_supported; 1b = yes [2] - create_supported; 1b = yes [1] - read_supported; 1b = yes. All RDE Devices shall support read Operations [0] - head_supported; 1b = yes
uint32	**DeviceConfigurationSignature** 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. 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.
varstring	**DeviceProviderName** An informal name for the RDE Device

11.2 NegotiateMediumParameters command format

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

Table 49 – NegotiateMediumParameters command format

Type	Request data
uint32	**MCMaximumTransferChunkSizeBytes** 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. All MC implementations shall support a transfer size of at least 64 bytes. NOTE for MCTP-based mediums, this is relative to the message size, not the packet size.
Type	**Response data**
enum8	**CompletionCode** value: { PLDM_BASE_CODES } If the MC reports a maximum transfer size of less than 64 bytes, the RDE Device shall respond with completion code ERROR_INVALID_DATA.
uint32	**DeviceMaximumTransferChunkSizeBytes** 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. All RDE Device implementations shall support a transfer size of at least 64 bytes. NOTE for MCTP-based mediums, this is relative to the message size, not the packet size.

11.3 GetSchemaDictionary command format

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

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

Table 50 – GetSchemaDictionary command format

Type	Request data
uint32	**ResourceID** 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.
schemaClass	**RequestedSchemaClass** The class of schema being requested

Type	Response data
enum8	**CompletionCode** value: { PLDM_BASE_CODES, ERROR_UNSUPPORTED, ERROR_NO_SUCH_RESOURCE } If the RDE Device does not support a schema of the type requested, it shall return **CompletionCode** ERROR_UNSUPPORTED. If the supplied Resource ID does not correspond to a collection, but the RequestedSchemaClass is COLLECTION_MEMBER_TYPE, the RDE Device shall return ERROR_INVALID_DATA.
uint8	**DictionaryFormat** The format of the dictionary as specified in the dictionary's **VersionTag**, defined in clause 7.2.3.2.
uint32	**TransferHandle** 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.

11.4 GetSchemaURI command format

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

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

Table 51 – GetSchemaURI command format

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

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

11.5 GetResourceETag command format

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

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

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

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

Table 52 – GetResourceETag command format

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

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

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

12.1 RDEOperationInit command format

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

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

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

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

Table 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. NOTE: Operation IDs with the most significant bit clear are reserved for use by the RDE Device; it is an error for the MC to supply such an ID.
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 }
bitfield8	**OperationFlags** Flags associated with this Operation: [7:3] - reserved for future use [2] - contains_custom_request_parameters; if 1b, the RDE Device should expect to receive a **SupplyCustomRequestParameters** command request before it may trigger the Operation [1] - contains_request_payload; if 0b, the Operation does not require data to be sent [0] - locator_valid; if 0b, the locator in the **OperationLocator** field shall be ignored
uint32	**SendDataTransferHandle** 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).
uint8	**OperationLocatorLength** 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 or if the **OperationType** field above is not OPERATION_UPDATE.
uint32	**RequestPayloadLength** Length in bytes of the request payload **in this message**. This value shall be zero under either of the following conditions: • 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	**OperationLocator** 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.
null or bejEncoding	**RequestPayload** The request payload. The format of this parameter shall be null (consisting of zero bytes) if the **RequestPayloadLength** above is zero; it shall be bejEncoding otherwise.

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

Type	Response data (continued)
bitfield8	**PermissionFlags** Indicates the access level (types of Operations; see Table 31) granted to the resource targeted by the Operation. [7: 6] - reserved for future use [5] - head access; 1b = access allowed [4] - delete access; 1b = access allowed [3] - create access; 1b = access allowed [2] - replace access; 1b = access allowed [1] - update access; 1b = access allowed [0] - read access; 1b = access allowed To process PermissionFlags, the MC shall behave as described in clause 7.2.4.2.8. 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.
uint32	**ResponsePayloadLength** Length in bytes of the response payload **in this message**. This value shall be zero under any of the following conditions: • 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	**ETag** String data for an ETag digest of the target resource; the string text format shall be UTF-8. The ETag shall be skipped (a string consisting of just the null terminator returned in this field) for any of the following actions: Action, Delete, Replace, and Update. The ETag shall also be skipped (a string consisting of just the null terminator returned in this field) if execution of the Operation has failed or not yet finished. To process an ETag, the MC shall behave as described in clause 7.2.4.2.4.
null or bejEncoding	**ResponsePayload** The response payload. The format of this parameter shall be null (consisting of zero bytes) if the **ResponsePayloadLength** above is zero; it shall be bejEncoding otherwise.

12.2 SupplyCustomRequestParameters command format

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

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

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

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

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

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

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

Table 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

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

Type	Response data (continued)
bitfield8	**PermissionFlags** Indicates the access level (types of Operations; see Table 31) granted to the resource targeted by the Operation. [7:6] - reserved for future use [5] - head access; 1b = access allowed [4] - delete access; 1b = access allowed [3] - create access; 1b = access allowed [2] - replace access; 1b = access allowed [1] - update access; 1b = access allowed [0] - read access; 1b = access allowed To process PermissionFlags, the MC shall behave as described in clause 7.2.4.2.8. 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. NOTE: The bit mapping for the PermissionFlags field was changed in version 1.0.1 of this specification to match that from the RDEOperationInit command, thereby making the entire response message identical for both of these commands.
uint32	**ResponsePayloadLength** Length in bytes of the response payload **in this message**. This value shall be zero under any of the following conditions: • 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	**ETag** String data for an ETag digest of the target resource; the string text format shall be UTF-8. The ETag may be skipped (an empty string returned in this field) for any of the following actions: Action, Delete, Replace, and Update. The ETag shall also be skipped (an empty string returned in this field) if execution of the Operation has not yet finished. This field supports the ETag Response header as described in clause 7.2.4.2.4.
null or bejEncoding	**ResponsePayload** The response payload. The format of this parameter shall be null (consisting of zero bytes) if the **ResponsePayloadLength** above is zero; it shall be bejEncoding otherwise.

12.3 RetrieveCustomResponseParameters command format

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

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

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

Table 55 – RetrieveCustomResponseParameters command format

Type	Request data
uint32	**ResourceID** The resourceID of a resource in the Redfish Resource PDR for the instance from which custom headers should be reported
rdeOpID	**OperationID** Identification number for this Operation; must match the one used for all commands relating to this Operation
Type	**Response data**
enum8	**CompletionCode** value: { PLDM_BASE_CODES, ERROR_OPERATION_ABANDONED, ERROR_OPERATION_FAILED, ERROR_UNEXPECTED, ERROR_NO_SUCH_RESOURCE }
uint32	**DeferralTimeframe** The expected length of time in seconds before the RDE Device will be able to respond to a request to start an Operation, or 0xFF if unknown. The MC shall ignore this field except when the completion code of the previous RDEOperationInit was ERROR_NOT_READY. This field supports the Retry-After Response header. 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.
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
…	**…**

12.4 RDEOperationComplete command format

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

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

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 }

12.5 RDEOperationStatus command format

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

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

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

Table 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_UNSUPPORTED , ERROR_NO_SUCH_RESOURCE, ERROR_ETAG_MATCH, ERROR_UNRECOGNIZED_CUSTOM_HEADER } The completion code for RDEOperationStatus shall be one of the following: SUCCESS: A valid active RDE Operation was referenced in the OperationID request field and it is not in the failed state. The actual current status of the RDE Operation is returned in the OperationStatus field – OR – an inactive RDE Operation was referenced in the OperationID request field. OperationStatus shall be OPERATION_INACTIVE in this case. ERROR_UNSUPPORTED, ERROR_ETAG_MATCH, ERROR_UNRECOGNIZED_CUSTOM_HEADER: A valid active RDE Operation was referenced in the OperationID request field, but the Operation failed with the specified status code. OperationStatus shall be OPERATION_FAILED in this case. These responses indicate a failure in the RDE Operation, not a failure in the RDEOperationStatus command.
enum8	**OperationStatus** values: { OPERATION_INACTIVE = 0; OPERATION_NEEDS_INPUT = 1; OPERATION_TRIGGERED= 2; OPERATION_RUNNING = 3; OPERATION_HAVE_RESULTS = 4; OPERATION_COMPLETED = 5, OPERATION_FAILED = 6, OPERATION_ABANDONED = 7 }
uint8	**CompletionPercentage** 0..100: percentage complete; 101-253: reserved for future use; 254: not supported or otherwise unable to estimate (but a valid Operation) 255: invalid Operation This value shall be zero if the Operation has not yet been triggered or if the Operation has failed.
uint32	**CompletionTimeSeconds** An estimate of the number of seconds remaining before the Operation is completed, or 0xFFFF FFFF if such an estimate cannot be provided. This value shall be 0xFFFF FFFF if the Operation has not yet been triggered or if the Operation has failed.
bitfield8	**OperationExecutionFlags** [7:4] - Reserved [3] - CacheAllowed – 1b = yes; shall be 0b for Operations other than read, head. Shall be 0b unless Operation has finished. Referring to RFC 7234, a value of yes shall be considered as equivalent to Cache-Control response header value “public” and a value of no shall be considered as equivalent to Cache-Control response header value “no-store”. Other cache directives are not supported. The decision of whether to allow caching of data is up to the RDE Device. Typically, static data is allowed to be cached unless, for example, it represents sensitive data such as login credentials; data that changes over time is generally not marked as cacheable To process the CacheAllowed flag, the MC shall behave as described in clause 7.2.4.2.7 [2] - HaveResultPayload – 1b = yes. Shall be 0b if Operation has not finished [1] - HaveCustomResponseParameters – 1b = yes. Shall be 0b if Operation has not finished [0] - TaskSpawned – 1b = yes For a failed Operation, this field shall be 0b for all flags other than HaveResultPayload, which may be 1b if a @Message.ExtendedInfo annotation is available to explain the result.

Type	Response data (continued)
uint32	**ResultTransferHandle** 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. 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.
bitfield8	**PermissionFlags** Indicates the access level (types of Operations; see Table 31) granted to the resource targeted by the Operation. [7:6] - reserved for future use [5] - head access; 1b = access allowed [4] - delete access; 1b = access allowed [3] - create access; 1b = access allowed [2] - replace access; 1b = access allowed [1] - update access; 1b = access allowed [0] - read access; 1b = access allowed To process PermissionFlags, the MC shall behave as described in clause 7.2.4.2.8. 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. NOTE: The bit mapping for the PermissionFlags field was changed in version 1.0.1 of this specification to match that from the RDEOperationInit command, thereby making the entire response message identical for both of these commands.
uint32	**ResponsePayloadLength** Length in bytes of the response payload **in this message**. This value shall be zero under any of the following conditions: • 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	**ETag** String data for an ETag digest of the target resource; the string text format shall be UTF-8. The ETag may be skipped (an empty string returned in this field) for any of the following actions: Action, Delete, Replace, and Update. The ETag shall also be skipped (an empty string returned in this field) if execution of the Operation has not yet finished. To process an ETag, the MC shall behave as described in clause 7.2.4.2.4.
null or bejEncoding	**ResponsePayload** The response payload. The format of this parameter shall be null (consisting of zero bytes) if the **ResponsePayloadLength** above is zero; it shall be bejEncoding otherwise.

12.6 RDEOperationKill command format

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

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

Table 58 – RDEOperationKill command format

Type	Request data
uint32	**ResourceID** The resourceID of a resource in the Redfish Resource PDR to which the Task's operation was targeted
rdeOpID	**OperationID** Identification number for this Operation; must match the one used for all commands relating to this Operation
bitfield8	**KillFlags** Flags for killing the Operation: [7:2] - reserved for future use [1] - run_to_completion; if 1b, the Operation should be run to completion but no further response should be sent to the MC. The MC shall not set the run_to_completion bit without also setting the discard_record bit. In the event that the MC violates this restriction, the RDE Device shall respond with completion code ERROR_INVALID_DATA and stop processing the request. [0] - discard_record; if 1b and the kill command returns success, the RDE Device shall discard internal records associated with this Operation as soon as it is killed; the RDE Device should not expect the MC to call **RedfishOperationComplete** for this Operation. If the Operation has spawned a Task, the RDE Device shall not create an Event when execution is finished.
Type	**Response data**
enum8	**CompletionCode** value: { PLDM_BASE_CODES, ERROR_OPERATION_ABANDONED, ERROR_OPERATION_FAILED, ERROR_OPERATION_UNKILLABLE, ERROR_NO_SUCH_RESOURCE }

12.7 RDEOperationEnumerate command format

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

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

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

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

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 }

13 PLDM for Redfish Device Enablement – Utility commands

13.1 MultipartSend command format

This command enables the MC to send a large volume of data to an RDE Device. In the event of a data checksum error, the MC may reissue the first MultipartSend command with the initial data transfer handle;

the RDE Device shall recognize this to mean that the transfer failed and respond as if this were the first transfer attempt. If the MC chooses not to restart the transfer, or in any other error occurs, the MC should abandon the transfer. In the latter case, if the transfer is part of an Operation, the MC shall explicitly abort and then finalize the Operation via the RDEOperationKill and RDEOperationComplete commands (see clauses 12.6 and 12.4).

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

Table 60 – MultipartSend command format

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

Type	Request data (continued)
uint32	**DataIntegrityChecksum** 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. 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.
Type	**Response data**
enum8	**CompletionCode** value: { PLDM_BASE_CODES, ERROR_OPERATION_ABANDONED, ERROR_OPERATION_FAILED, ERROR_UNEXPECTED, ERROR_BAD_CHECKSUM } If the DataTransferHandle does not correspond to a valid chunk, the RDE Device shall return CompletionCode ERROR_INVALID_DATA.
enum8	**TransferOperation** The follow-up action that the RDE Device is requesting of the MC: • 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 value: { XFER_FIRST_PART = 0, XFER_NEXT_PART = 1, XFER_ABORT = 2, XFER_COMPLETE = 3 }

13.2 MultipartReceive command format

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

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

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

Table 61 – MultipartReceive command format

Type	Request data
uint32	**DataTransferHandle** A handle to uniquely identify the chunk of data to be retrieved. If TransferOperation below is XFER_FIRST_PART and the OperationID below is zero, this must match the TransferHandle supplied by the RDE Device in the response to the GetSchemaDictionary 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. The DataTransferHandle supplied shall be either the initial handle to begin or restart a transfer or the NextDataTransferHandle supplied with the previous chunk.
rdeOpID	**OperationID** Identification number for this Operation; must match the one previously used for all commands relating to this Operation; 0x0000 if this transfer is not part of an Operation
enum8	**TransferOperation** The portion of data requested for the transfer: • 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 value: { XFER_FIRST_PART = 0, XFER_NEXT_PART = 1, XFER_ABORT = 2 }
Type	**Response data**
enum8	**CompletionCode** value: { PLDM_BASE_CODES, ERROR_OPERATION_ABANDONED, ERROR_OPERATION_FAILED, ERROR_UNEXPECTED, ERROR_BAD_CHECKSUM } If the DataTransferHandle does not correspond to a valid chunk, the RDE Device shall return CompletionCode ERROR_INVALID_DATA. If the transfer is aborted, the RDE Device shall acknowledge this status by returning SUCCESS.
enum8	**TransferFlag** value: { START = 0, MIDDLE = 1, END = 2, START_AND_END = 3 } This field shall be omitted for a non-SUCCESS **CompletionCode** or if the transfer has been aborted
uint32	**NextDataTransferHandle** The handle for the next chunk of data for this transfer; zero (0x00000000) if no further data This field shall be omitted for a non-SUCCESS **CompletionCode** or if the transfer has been aborted
uint32	**DataLengthBytes** The length in bytes N of data being sent in this chunk, including both the Data and DataIntegrityChecksum (if present) fields. This value and the data bytes associated with it shall not cause this response message to exceed the negotiated maximum transfer chunk size (clause 11.2). This field shall be omitted for a non-SUCCESS **CompletionCode** or if the transfer has been aborted
uint8	**Data [0]** The first byte of current chunk of data. Shall be omitted if only the DataIntegrityChecksum is present. This field shall be omitted for a non-SUCCESS **CompletionCode** or if the transfer has been aborted
…	**…**

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

14 Additional Information

14.1 Multipart transfers

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

14.1.1 Flag usage for MultipartSend

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

- To prepare a large data send for use in an RDE command, a DataTransferHandle shall be sent by the MC in the request message of the RDEOperationInit command.
- To reflect a data transfer (re)initiated with a MultipartSend command, the TransferOperation shall be set to XFER_FIRST_PART in the response message.
- 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 request for the previous MultipartSend command for this data transfer.
- The TransferFlag specified in the request for a MultipartSend command has the following meanings:
  - START, which is the first part of the data transfer
  - MIDDLE, which is neither the first nor the last part of the data transfer
  - END, which is the last part of the data transfer
  - START_AND_END, which is the first and the last part of the data transfer. In this case, the transfer consists of a single chunk

- 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.

14.1.2 Flag usage for MultipartReceive

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

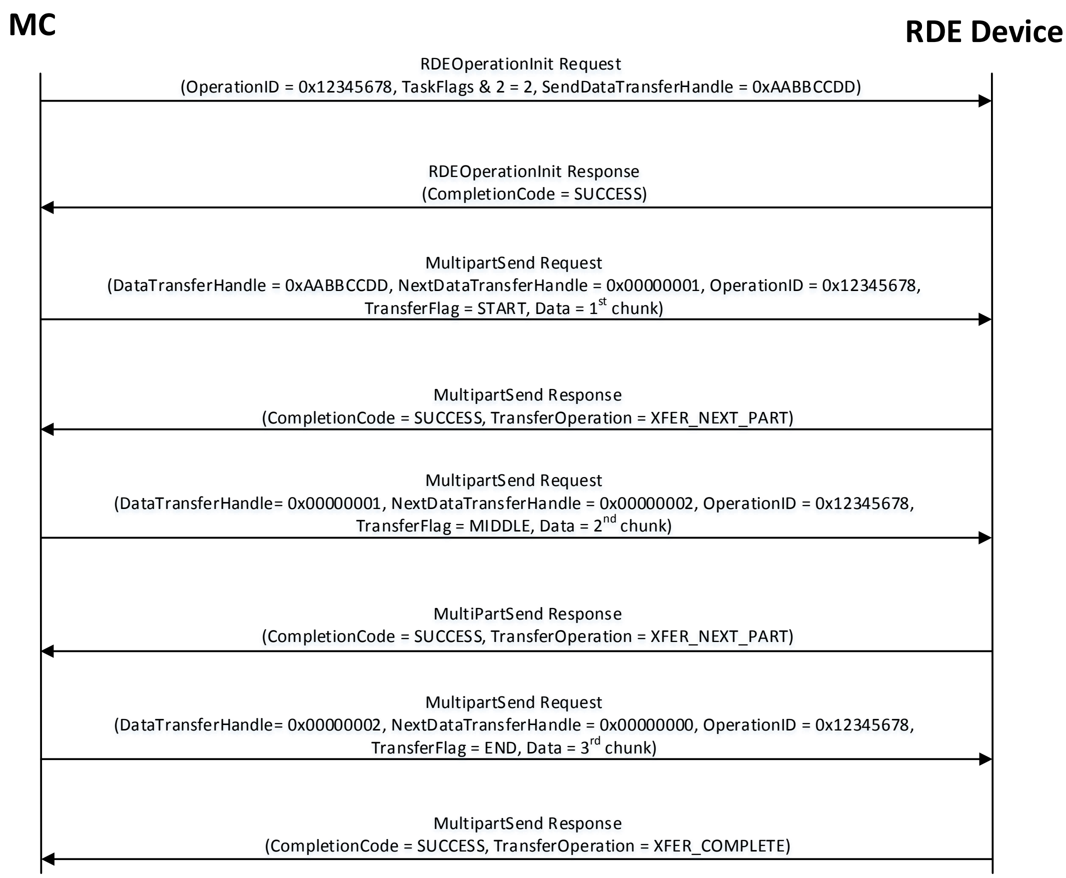
- To prepare a large data transfer receive for use in an RDE command, a DataTransferHandle shall be sent by the RDE Device in the response message to the RDEOperationInit, SupplyCustomRequestParameters, or RDEOperationStatus command after an Operation has finished execution and results are ready for pick-up.
- To initiate a data transfer with either a MultipartReceive command, the TransferOperation shall be set to XFER_FIRST_PART in the request message.
- 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.
- The TransferFlag specified in the response of a MultipartReceive command has the following meanings:
  - START, which is the first part of the data transfer
  - MIDDLE, which is neither the first nor the last part of the data transfer
  - END, which is the last part of the data transfer
  - START_AND_END, which is the first and the last part of the data transfer
- For a MultipartReceive, the requester shall consider a data transfer complete when the TransferFlag in the response is set to End or StartAndEnd.

14.1.3 Multipart transfer examples

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

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

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



2749

2750 **Figure 17 – MultipartSend example**

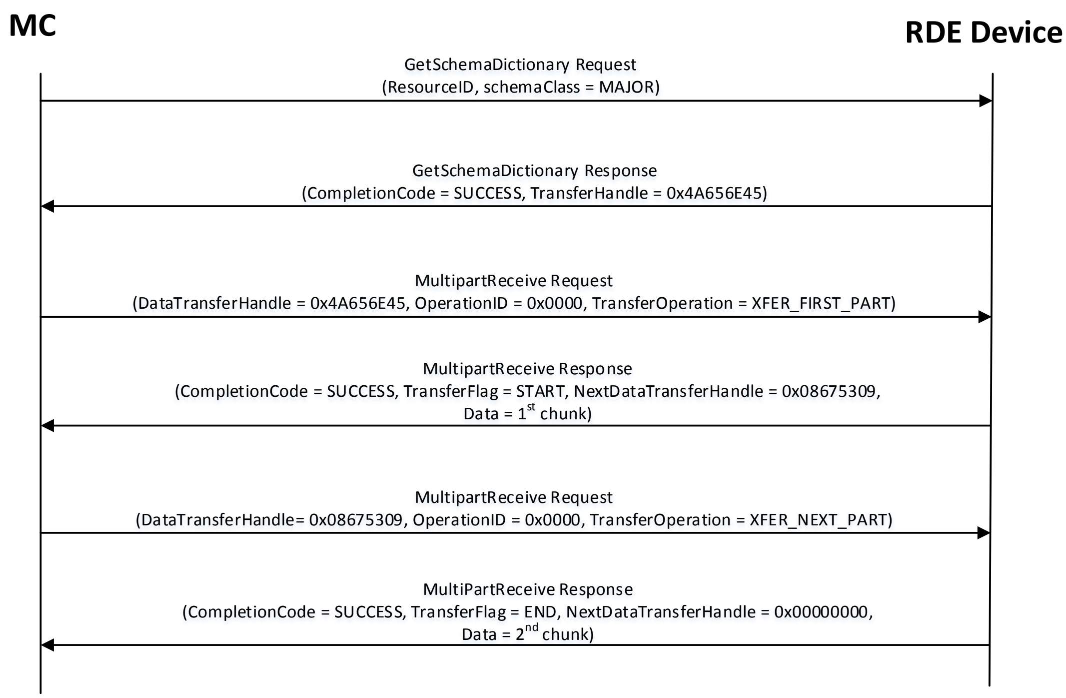


Figure 18 – MultipartReceive example

14.2 Implementation notes

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

14.2.1 Schema updates

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

14.2.2 Storage of dictionaries

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

14.2.3 Dictionaries for related schemas

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

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

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

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

14.2.4 [MC] HTTP/HTTPS POST Operations

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

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

14.2.4.1 Support for Actions

When a Redfish client issues a Redfish Operation for an Action, the URI target for the Action will be a POST of the form /redfish/v1/{path to root of RDE Device component}/{path to RDE Device owned resource}/Actions/schema_name.action_name. To process this, the MC may translate {path to root of RDE Device component} and {path to RDE Device owned resource} normally to identify the PDR against which the Operation should be executed. (If the URI is not in this format, this is another indication that the POST operation is probably a CREATE.) After it has performed this step, the MC can then check its PDR hierarchy to find the Redfish Action PDR containing an action named schema_name.action_name. If it doesn't find one, the MC shall respond with HTTP status code 404, Not Found and stop processing the Operation.

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

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

14.2.5 Consistency checking of read Operations

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

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

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

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

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

When retrieving these PDRs for RDE Device components, the MC should read the ProposedContainingResourceName from the PDR. While following this recommendation is not mandatory, the MC should use it to inform a placement decision. If the MC does not follow the placement recommendation, it should read the MajorSchemaName field to identify the type of RDE Device they correspond to. Within the canon of standard Redfish schemas, there are comparatively few that reside at the top level, and each has a well-defined place it should appear within the hierarchy. The MC should thus make a simple map of which top-level schema types map to which places in the hierarchy and use this to place RDE Devices. In making these placement decisions, the MC should take information about the hardware platform topology into account so as to best reflect the overall Redfish system.

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

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

14.2.7 LogEntry and LogEntryCollection resources

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

14.2.8 On-demand pagination

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

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

14.2.9 Considerations for Redfish clients

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

ANNEX A
(informative)

Change log

Version	Date	Description
1.0.0	2019-06-25	
1.0.1	2020-01-21	Errata update