Document Identifier: DSP0218

Date: 2018-02-20

Version: 0.8.0a

Platform Level Data Model (PLDM) for Redfish Device Enablement Specification

Information for Work-in-Progress version:

IMPORTANT: This document is not a standard. It does not necessarily reflect the views of the DMTF or its members. Because this document is a Work in Progress, this document may still change, perhaps profoundly and without notice. This document is available for public review and comment until superseded.

Provide any comments through the DMTF Feedback Portal:
http://www.dmtf.org/standards/feedback

Supersedes:

Document Class: Normative

Document Status: Work in Progress

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 authors wish to acknowledge the following people.

Editor:

- Bill Scherer – Hewlett Packard Enterprise

Contributors:

- Richelle Ahlvers – Broadcom Limited
- Jeff Autor – Hewlett Packard Enterprise
- Patrick Caporale – Lenovo
- Mike Garrett – Hewlett Packard Enterprise
- Jeff Hilland – Hewlett Packard Enterprise
- Yuval Itkin – Mellanox
- Ira Kalman – Intel
- Eliel Louzoun – Intel
- Balaji Natrajan – Microsemi
- Edward Newman – Hewlett Packard Enterprise
- Zvika Perry Peleg – Cavium
- Jeffrey Plank – Microsemi
- Patrick Schoeller – Hewlett Packard Enterprise
- Hemal Shah – Broadcom Limited
- 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 devices to participate in Redfish-based management without needing to support either JavaScript Object Notation (JSON, used for operation data payloads) or the [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 device, which ultimately services requests; and the management controller (MC), which translates requests and serves as an intermediary between the client and the device.

Document conventions

Section naming conventions

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

- The titles of sections that are primarily of interest to MCs are prefixed with “[MC]”.
- The titles of sections that are primarily of interest to devices are prefixed with “[Dev]”
- The titles of sections that are primarily of interest to clients are prefixed with “[Cli]”
- Unless explicitly marked, the subsections of a section marked as being primarily of interest to one role are also primarily of interest to that same role
- Sections that are of primary interest to more than one role are not prefixed

NOTE: A design goal of this specification is that clients shall not need to be aware whether the device whose data they are interacting with is supporting Redfish directly or through an MC proxy. Consequently, there is only one section in this document of primary interest to clients.

Typographical conventions

This document uses the following typographical conventions:

- Document titles are marked in *italics*.

Platform Level Data Model (PLDM) for Redfish Device Enablement Specification

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 the [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 devices or any details of handling the Redfish security model.

In this specification, Redfish management functionality is divided between the three roles: the client, which initiates management operations; the device, which ultimately services requests; and the management controller (MC), which translates requests and serves as an intermediary between the client and the device. An implementor of this specification is only required to support the features of one of these roles. In particular, a 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 Redfish PDRs referenced in this specification*
- DMTF DSP0249, *Platform Level Data Model (PLDM) State Sets* Specification, defines the Schema Entity type 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.

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

DMTF DSP0266, *Redfish Scalable Platforms Management API Specification* 1.2.1, http://www.dmtf.org/sites/default/files/standards/documents/DSP0266_1.2.1.pdf

DMTF DSP0236, *MCTP Base Specification 1.2.0*, http://dmtf.org/sites/default/files/standards/documents/DSP0236_1.2.x.pdf

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

DMTF DSP0245, *Platform Level Data Model (PLDM) IDs and Codes Specification 1.2.0*, http://dmtf.org/sites/default/files/standards/documents/DSP0245_1.2.x.pdf

DMTF DSP0248, *Platform Level Data Model (PLDM) for Platform Monitoring and Control Specification 1.1.0*, http://dmtf.org/sites/default/files/standards/documents/DSP0248_1.1.x.pdf

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

DMTF DSP4004, *DMTF Release Process 2.4,* http://dmtf.org/sites/default/files/standards/documents/DSP4004_2.4.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 RFC 7232, R. Fielding et al., Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests, http://www.ietf.org/rfc/rfc7232.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

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 to 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
For purposes of this specification, any executive agent that enables a user to manage Redfish-compliant systems and 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
For purposes of this document, any executive agent endpoint Redfish provider that requires the intervention of an MC to receive Redfish communications

3.6
Device Component
A top-level entry point into the schema hierarchy presented by a device

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

3.8
Discovery
The process by which an MC determines that a device supports PLDM for Redfish Device Enablement

3.9
Long-running Task
Any Task for which a device cannot complete execution in the time allotted to respond to the PLDM trigger command message sent from the MC

3.10
Major Schema
The primary schema defining the format of a collection of data, usually a published standard Redfish schema. See OEM Extension.

3.11
Management Controller (MC)
For purposes of this specification, any executive agent that serves as an intermediary between devices and clients

3.12
Member
Any of the independent resources contained within a collection

3.13
Metadata
Information that decribes data of interest, such as its type format, length in bytes, or encoding method

3.14
OData
The Open Data protocol, a source of annotations in Redfish, as defined by OASIS.

3.15
OEM Extension
Any manufacturer-specific addition to major schema

3.16
Operation
Any Redfish operation transmitted via HTTP or HTTPS from a client to an MC for execution

3.17
Property
An individual datum contained within a Resource

3.18
Provider
Any device or software that responds to Redfish commands

3.19
Registration
The process of enabling a Redfish provider device with an MC

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
A sequence of PLDM messages and operations triggered by PLDM messages that represent a Redfish operation being executed by an MC and/or a device on behalf of a client. This differs from the standard Redfish notion of a Task.

397 **3.23**
398 **Trigger**
399 Any of several command messages sent from the MC to a device to authorize execution of a Task.
400 Specifically

401 4 Symbols and abbreviated terms

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

404 **4.1**
405 **BEJ**
406 Binary Encoded JSON, a compressed binary format for encoding JSON data

407 **4.2**
408 **JSON**
409 JavaScript Object Notation

410 **4.3**
411 **RDE**
412 Redfish Device Enablement

413 5 Conventions

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

416 5.1 Reserved and Unassigned Values

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

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

421 5.2 Byte Ordering

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

425 5.3 PLDM for Redfish Device Enablement Data Types

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

428 **Table 1 – PLDM for Redfish Device Enablement Data Types and Structures**

Data Type	Interpretation
varstring	A multiformat text string per Section 5.3.1
bytestream	A string of unsigned 8-bit values of variable length
utf8string	A null-terminated text string in UTF-8 format

schemaClass	An enumeration of the various schemas associated with a collection of data, encoded per Section 5.3.2
nnint	A non-negative integer encoded for BEJ per Section 5.3.3
bejTuple	A BEJ tuple, encoded per Section 5.3.4
bejTupleS	A BEJ Sequence Number tuple element, encoded per Section 5.3.5
bejTupleF	A BEJ Format tuple element, encoded per Section 5.3.6
bejTupleL	A BEJ Length tuple element, encoded per Section 5.3.7
bejTupleV	A BEJ Value tuple element, encoded per Section 5.3.8
bejNull	Null data encoded for BEJ per Section 5.3.9
bejInteger	Integer data encoded for BEJ per Section 5.3.10
bejEnum	Enumeration data encoded for BEJ per Section 5.3.11
bejString	String data encoded for BEJ per Section 5.3.12
bejReal	Real data encoded for BEJ per Section 5.3.13
bejBoolean	Boolean data encoded for BEJ per Section 5.3.14
bejBytestring	Bytestring data encoded for BEJ per Section 5.3.15
bejSet	Set data encoded for BEJ per Section 5.3.16
bejArray	Array data encoded for BEJ per Section 5.3.17
bejChoice	Choice data encoded for BEJ per Section 5.3.18
bejSchemaLink	Schema Link data encoded for BEJ per Section 5.3.19
bejLocator	An intra-schema locator for operation targeting; formatted per Section 5.3.20

5.3.1 varstring PLDM Type

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, SHIFT-JIS = 6, EBCDIC = 7 }
uint16	**stringLengthBytes** Including any null terminator
variable	**stringData**

5.3.2 schemaClass PLDM Type

Table 3 – schemaClass Enumeration

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

The schemaClass type will typically be paired with a schema index to identify a particular schema in the case of OEM extensions; the schema index is a 1-based index into a list of OEM extensions to the major schema or to the Event schema.

5.3.3 nnint PLDM Type

The nnint type captures the BEJ encoding of Non-Negative 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.

Note that this type is NOT to be used for the generalized encoding of BEJ Integer data – even with non-negative values – in the Value tuple element, bejTupleV (Section 5.3.8).

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 bejTuple PLDM Type

Table 5 – bejTuple Encoding for BEJ

Type	Description
bejTupleS	Tuple element for the Sequence Number field, described in Section 8.2.1
bejTupleF	Tuple element for the Format field, described in Section 8.2.2
bejTupleL	Tuple element for the Length field, described in Section 8.2.3
bejTupleV	Tuple element for the Value field, described in Section 8.2.4

448 ### 5.3.5 bejTupleS PLDM Type

449 **Table 6 – bejTupleS Encoding for BEJ**

Type	Description
nnint	Sequence number enhanced to indicate the schema to which it refers. More specifically, except in schema selector case 11 below (where the sequence number belongs to an OEM extension to the major schema, there is more than one OEM extension to this schema, and the one for which this data is encoded is not the first one): Bits 0,1: Schema selector (see below) Bits 2+ Sequence number within the schema The schema selector conveys which schema the sequence number references and will be one of the following values: 00: Major schema 01: First OEM extension to a major schema (index 1) 10: Annotation schema 11: OEM extension 2+ to a major schema For schema selector 11, the sequence number is encoded in the following extended format: Bits 0,1: Schema selector Bits 2..7: OEM extension index; add 2 to get the actual one-based OEM extension index Bits 8+ Sequence number within the schema The OEM extension index may be any value 0..62; the value 63 is reserved for future use. This format allows for a maximum of 64 OEM extensions to be selected from. Should the number of OEM extensions exceed this amount, 63 shall be employed in the future for a secondary escape.

450 ### 5.3.6 bejTupleF PLDM Type

451 **Table 7 – bejTupleF Encoding for BEJ**

Type	Description
uint8	Format code; the low nibble represents the datatype and the high nibble represents a series of flag bits

452 **Table 8 – BEJ Format Codes (Low Nibble: Data Types)**

Code	BEJ Type	PLDM Type in Value Tuple Field *
0x00	BEJ Set	bejSet
0x01	BEJ Array	bejArray
0x02	BEJ Null	bejNull
0x03	BEJ Integer	bejInteger
0x04	BEJ Enum	bejEnum
0x05	BEJ String	bejString
0x06	BEJ Real	bejReal
0x07	BEJ Boolean	bejBoolean
0x08	BEJ Bytestring	bejBytestring
0x09	BEJ Choice	bejChoice

0x0A-0x0E	Reserved	
0x0F	BEJ Schema Link	bejSchemaLink

Table 9 – BEJ Format Codes (High Nibble: Flag Bits)

Code	Type
0x10	Flag bit 1 (deferred binding) *
0x20	Flag bit 2 (reserved)
0x40	Flag bit 3 (reserved)
0x80	Flag bit 4 (reserved)

* Flag bit 1 in this specification represents deferred binding of data. This flag shall only be set in conjunction with BEJ String data. See Section 8.3.

Flag bits 2-4 are reserved for future use.

5.3.7 bejTupleL PLDM Type

Table 10 – bejTupleL Encoding for BEJ

Type	Description
nnint	Length in bytes of value tuple field

5.3.8 bejTupleV PLDM Type

Table 11 – bejTupleV Encoding for BEJ

Type	Description
bejNull, bejInteger, bejEnum, bejString, bejReal, bejBoolean, bejBytestring, bejSet, bejArray, bejChoice, or bejResourceLink	Value tuple field; exact type shall match that of the Format tuple element contained within the same tuple per Table 8.

5.3.9 bejNull PLDM Type

Table 12 – bejNull Encoding for BEJ

Type	Description
(none)	No fields

5.3.10 bejInteger PLDM 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.

469 **Table 13 – bejInteger 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)

470 ### 5.3.11 bejEnum PLDM Type

471 **Table 14 – bejEnum Encoding for BEJ**

Type	Description
uint8	Integer value of the sequence number for the enumeration option selected

472 ### 5.3.12 bejString PLDM Type

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

474 **Table 15 – bejString 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

475 ### 5.3.13 bejReal PLDM Type

476 BEJ encoding for *whole, fract*, and *exp* that represent the base 10 encoding *whole.fract* × 10^{exp}. Note that
477 there is no need to express special values (positive infinity, negative infinity, NaN, negative zero) because
478 these cannot be expressed in JSON.

479 **Table 16 – bejReal Encoding for BEJ**

Type	Description
nnint	Length of *whole*
bejInteger	*whole*
nnint	leading zero count for fract
nnint	*fract*
nnint	Length of *exp*
bejInteger	*Exp*

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

5.3.14 bejBoolean PLDM Type

Table 17 – bejBoolean Encoding for BEJ

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

5.3.15 bejBytestring PLDM Type

Table 18 – bejBytestring 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)

5.3.16 bejSet PLDM Type

Table 19 – bejSet 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)

5.3.17 bejArray PLDM Type

Table 20 – bejArray 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)

5.3.18 bejChoice PLDM Type

Table 21 – bejChoice Encoding for BEJ

Type	Description
bejTuple	Selected option

5.3.19 bejResourceLink PLDM Type

Flag bits must not be set in conjunction with a Resource Link encoding.

Table 22 – bejSchemaLink Encoding for BEJ

Type	Description
nnint	ResourceID of Redfish Resource PDR for linked schema

5.3.20 bejLocator PLDM Type

The use of BEJ locators is detailed in Section 8.5. All sequence numbers within a BEJ locator shall reference the same schema dictionary.

Table 23 – bejLocator Encoding

Type	Description
nnint	**LengthBytes** The length in bytes of the series of sequence numbers comprising this locator
bejTuplesS	Sequence number [0]
bejTuplesS	Sequence number [1]
bejTuplesS	Sequence number [2]
…	…
bejTuplesS	Sequence number [N - 1]

6 PLDM for Redfish Device Enablement Version

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

In response to the GetPLDMVersion command described in DSP0240, the reported version of this specification shall be encoded as 0xF1F0F000.

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 devices contained within a platform management subsystem. These messages are designed to be delivered using PLDM messaging.

Traditionally, management has been effected via a myriad of proprietary approaches for limited classes of devices. These disparate solutions differ in feature sets and APIs, creating a implementation and integration issues for the management controller, which ends up needing custom code to support each each one separately. This consumes resoures 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 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 and it 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 Section 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. 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 doesn't 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

- Device Registration for Redfish
  - A mechanism to determine the schemas the device supports, including OEM custom extensions
  - A mechanism to determine parameters for limitations on the types of communication the 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 Task operations
  - Asynchrony support for long-running Tasks
  - Notification Events for completion of Tasks and for other device-specific happenings
  - 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 device for servicing. The 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.

7.1.1 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 service provider for the device. In order to perform this role, the MC discovers and registers the device by interrogating its schema support and building a representation of the device's management topology. Once this is done, the MC is responsible for receiving Redfish messages from the client, identifying the device that supplies the data relevant to the request, encoding any payloads into the binary BEJ format, and delivering them to the device via PLDM. Finally, the MC is responsible for interacting with the 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 **Device** is a data provider. To perform this role, the 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 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 sections.

7.2.1 Device Discovery and Registration

The processes by which a 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 a device and recognizing that it supports Redfish management. Registration consists of the MC interrogating specific details of the device's Redfish capabilities and then making it visible to external clients.

7.2.1.1 Redfish Device Discovery

The first step of the discovery process begins when the MC detects the presence of a PLDM capable device. The technique by which the MC determines that a device supports PLDM is outside the scope of this specification.

Once the MC knows that the 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 PLDM for Redfish Device Enablement, a device shall respond to the GetPLDMTypes request with a response indicating that it supports both Monitoring and Control (type 2, DSP0248) and Redfish Device Enablement (type 6, this document).

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 device supports the required commands. The required commands for each PLDM Type are listed in Table 38. As with the GetPLDMTypes command, use of this command is optional if the MC has some other technique to understand which commands the device supports.

The final step in discovery is for the MC to invoke the NegotiateRedfishParameters command (Section 11.1) in order to negotiate some baseline details for the device. Unlike some of the previous steps, this one is mandatory. Baseline details include the following:

- The device's Redfish provider name
- The level of support the device has for asynchrony. This indicates whether the device can issue asynchronous alerts in response to happenings such as task completion or device-specific Redfish events.
- The device's support for concurrency. This is the number of outstanding long-running Tasks the device can support simultaneously

7.2.1.2 Redfish Device Registration

In the registration process, the MC interrogates the device about the hierarchy of Redfish resources it supports in order to act as a proxy, transparently mirroring them to external clients.

In PLDM for Redfish Device Enablement, there is a one-to-one mapping of Redfish Resource PDRs to resources; the PDRs represent the collection of data, not its organization. These instances link together to form a management topology of one or more trees called device components; each instance corresponds to a single node in a tree.

The first step in performing the registration is for the MC to collect an inventory of the PDRs supported by devices. There are three main PDRs of interest here: Redfish Resource PDRs, that represent an instance of data provided by the device; Redfish Entity Association PDRs, that represent the logical linking of data; and Redfish Action PDRs, that represent special functions the device supports. The MC shall collect this

information by first calling the PLDM Monitoring and Control GetPDRRepositoryInfo command to determine the total number of PDRs the device supports. It shall then use the PLDM Monitoring and Control GetPDR command to retrieve details for each PDR from the device.

As it retrieves the PDR information, the MC should build an internal representation of the data hierarchy for the 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 device.

Once the MC has built up a representation of the 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 s first check the schema name and version indicated in each Redfish Resource PDR to understand what the device supports. For any of these schemas where the MC does not already have one, it may optionally download 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 command with the corresponding Redfish Resource PDR.

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

Each resource is represented in this specification by a separate instance of 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 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. Every device shall proffer a single Redfish Resource PDR as the logical root of its management topology for each device component it presents.

Links between resources can be modeled in two different ways. Direct 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 the device component is set to EXTERNAL, 0x0000.

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 1, the drives in a RAID subsystem are subordinate to the storage controller that manages them, but are also linked to the standard Chassis object.

7.2.2.1 Example Linking of PDRs within Devices

This clause presents an example of the way a device can link Redfish Resource PDRs together to present its data for management.

The example in Figure 1 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 StorageController

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

The StorageController has five Redfish Resource PDRs that list it as their container: one that offers data in the VolumeCollection resourceand four that offer data in the Disk resource. 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 are modeled using Redfish Entity Association PDRs.

To show that the drives are related to the volume, the device offers a Redfish Entity Association PDR with ContainingResourceID set to 567, the ID of the volume. This PDR then lists the four drives as the four ContainedResourceIDs for the association, marking them as being part of the volume.

The relation between the drives and the outside Chassis object is also marked with a Redfish Entity Association PDR. The only difference is that the ContainingResourceID for this relation contains the value EXTERNAL, to show that the drives are visible outside the device.

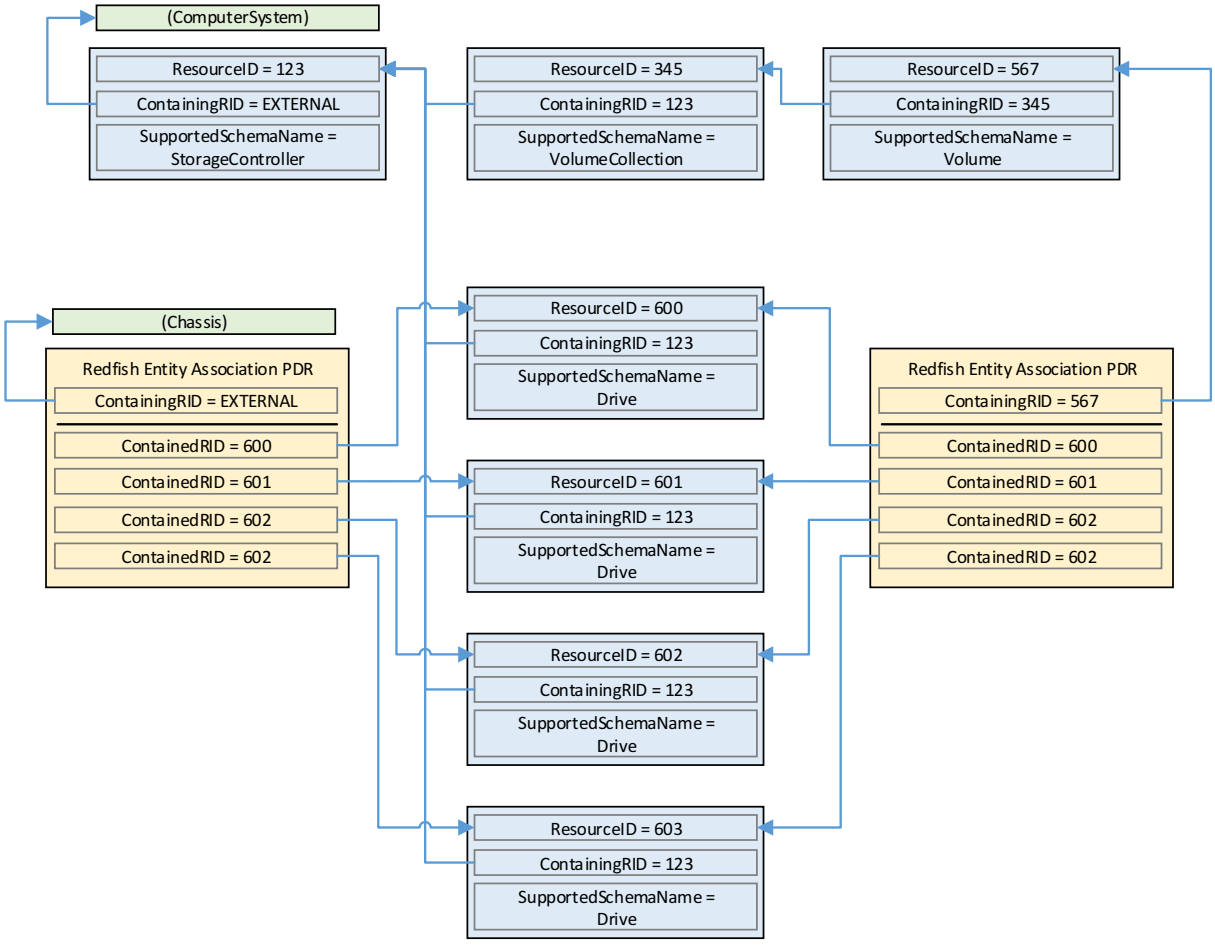


Figure 1 – Example linking of Redfish Resource and Redfish Entity Association PDRs

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 Section 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 devices with limited memory and computational resources. In particular, the names of properties are skipped in the BEJ encoding.

Each Redfish Resource PDR can reference up to five classes of dictionaries for the schemas it can use:

1. Standard Redfish data schema (aka the major schema)
2. OEM extensions for the major schema (multiple OEM extensions are possible)
3. Standard Redfish Event schema
4. OEM extensions of the standard Redfish Event schema (multiple OEM extensions are possible)
5. Standard Redfish Annotation schema

7.2.3.1 Canonicalizing a Schema into a Dictionary

In JSON and CSDL 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, 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.
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.

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

Note that 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. As 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.

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

7.2.3.2 Dictionary Binary Format

The binary format of dictionaries shall be as follows:

Table 24 – Redfish Dictionary binary format

Type	Dictionary Data data
uint8	Dictionary format version tag: 0x00 for the current spec
uint8	Reserved
uint16	Number of dictionary entries
bejTupleF	Entry 0 property format
uint8	Entry 0 property sequence number
uint16	Entry 0 property child pointer offset
uint8	Entry 0 property name string length
uint8	Entry 0 property display format string length
uint16	Entry 0 property name string offset
uint16	Entry 0 property display format string offset
bejTupleF	Entry 1 property format
uint8	Entry 1 property sequence number
uint16	Entry 1 property child pointer offset
uint8	Entry 1 property name string length
uint8	Entry 1 property display format string length
uint16	Entry 1 property name string offset
uint16	Entry 1 property display format string offset
…	…
utf8string	Entry 0 property name string
uint8 (x0-3)	Null byte padding to reach a DWORD boundary
utf8string	Entry 0 property display format string
uint8 (x0-3)	Null byte padding to reach a DWORD boundary

utf8string	Entry 1 property name string
uint8 (x0-3)	Null byte padding to reach a DWORD boundary
utf8string	Entry 1 property display format string
uint8 (x0-3)	Null byte padding to reach a DWORD boundary
…	…

7.2.4 [MC] Redfish Operations

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

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

7.2.4.1 Primary Operations

There are seven primary operations in Redfish. These are summarized in Table 25.

Table 25 – Redfish Operations

Operation	Verb	Description
Read	GET	Retrieves data values for all properties contained within a resource
Update	PATCH	Writes 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	Writes 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	Retrieves just headers for the data contained in a schema

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 HTTP/HTTPS 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 25. The URI may consist of several parts of interest for purposes of this specification: a prefix that points to the device being managed, a subpath within the 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 HTTP/HTTPS request operation will contain a series of headers each of which modifies it in some fashion.

7.2.4.1.1.2 Redfish HTTP/HTTPS 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 HTTP/HTTPS 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 Section 9.):

1. Parse the prefix of the supplied URI to pinpoint the device that the operation targets
2. Parse the device portion of the URI to identify the specific place in the 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 (Section 7.2.4.2) and query options (Section 7.2.4.3) into parameters to the corresponding PLDM request message
6. Translate the JSON payload, if present, into a corresponding BEJ (Section 8) payload for the request, using a dictionary appropriate for the target Redfish Resource PDR
7. Send the PLDM for Redfish Device Enablement RedfishTaskInit command (Section 13.1) to initialize the Task
8. Send any BEJ payload to the device via one or more PLDM for Redfish Device Enablement MultipartSend commands (Section 14.1)
9. Send the appropriate PLDM for Redfish Device Enablement Task Trigger command (Section 13) to the device to cause it to execute the Task
10. Retrieve and decode the BEJ-encoded JSON data for any Task response payloads via one or more PLDM for Redfish Device Enablement MultipartReceive commands (Section 14.2)
11. Send the PLDM for Redfish Device Enablement RedfishTaskComplete command (Section 13.10) to inform the device that it may discard any data structures associated with the Task
12. Prepare and send the final response to the client, adding the various HTTP/HTTPS response headers (Section 7.2.4.2) appropriate to the type of Redfish operation that was just performed

7.2.4.2 Redfish Operation HTTP(S) Headers

Several headers modify Redfish operations when transmitted in the HTTP/HTTPS transport layer. These are summarized in Table 26. Implementation notes for how the MC shall support some of these modifiers – when attached to Redfish operations – may be found in the indicated subsections. For those headers where the referred section is listed as "n/a", the implementation is outside the scope of this specification; implementors shall refer to DSP0266 and standard HTTP/HTTPS documentation for more information on processing these headers.

Table 26 – Redfish Operation Headers

Header	Section	Where Used	Description
Request Headers			

Accept	n/a	Request	Indicates to the server what media type(s) this client is prepared to accept.
Accept-Encoding	n/a	Request	Indicates if gzip encoding can be handled by the client.
Accept-Language	n/a	Request	This header is used to indicate the language(s) requested in the response.
Content-Type	n/a	Request	Describes the type of representation used in the message body.
Content-Length	n/a	Request	Describes the size of the message body.
OData-MaxVersion	n/a	Request	Indicates the maximum version of OData that an odata-aware client understands
OData-Version	n/a	Request	Services shall reject requests which specify an unsupported OData version. If a service encounters a version that it does not support, the service should reject the request with status code [412] (#status-412). If client does not specify an Odata-Version header, the client is outside the boundaries of this specification.
Authorization	n/a	Request	Required for HTTP session management
User-Agent	n/a	Request	Required for tracing product tokens and their version. Multiple product tokens may be listed.
Host	n/a	Request	Required to allow support of multiple origin hosts at a single IP address.
Origin	n/a	Request	Used to allow web applications to consume Redfish Service while preventing CSRF attacks.
Via	n/a	Request	Indicates network hierarchy and recognizes message loops. Each pass inserts its own VIA.
Max-Forwards	n/a	Request	Limits gateway and proxy hops. Prevents messages from remaining in the network indefinitely. The communication between the MC and the device shall not be considered as a hop for purposes of this header.
If-Match	7.2.4.2.1	Request	If-Match shall be supported on PUT and PATCH requests for resources for which the service 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 service 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.
X-Auth-Token	n/a	Request	Used for authentication of user sessions
Custom HTTP/ HTTPS Headers	7.2.4.2.3	Request and Response	Non-standard headers used for custom purposes
Response Headers			
OData-Version	n/a	Response	Describes the OData version of the payload that the response conforms to.
Content-Type	n/a	Response	Describes the type of representation used in the message body
Content-Encoding	n/a	Response	Describes the encoding that has been performed on the media type
Content-Length	n/a	Response	Describes the size of the message body

ETag	7.2.4.2.4	Response	An identifier for a specific version of a resource, often a message digest.
Server	n/a	Response	Required to describe a product token and its version. Multiple product tokens may be listed.
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
Via	n/a	Response	Indicates network hierarchy and recognizes message loops. Each pass inserts its own VIA.
Max-Forwards	n/a	Response	Limits gateway and proxy hops. Prevents messages from remaining in the network indefinitely.
Access-Control-Allow-Origin	n/a	Response	Prevents or allows requests based on originating domain. Used to prevent CSRF attacks.
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.
WWW-Authenticate	n/a	Response	Required for Basic and other optional authentication mechanisms. See the Security clause in DSP0266 for details.
X-Auth-Token	n/a	Response	Used for authentication of user sessions. The token value shall be indistinguishable from random.
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 device via the ETagFormat[0], ETagLengthBytes[0], ETag[0] fields of the PLDM SupplyCustomParameters command (Section 13.1) 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. The device shall verify that the ETag matches the current state of the targeted schema data instance before effecting the operation.

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 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 PUT and PATCH 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 device via the ETagFormat[i], ETagLengthBytes[i], and ETag[i] fields of the PLDM PLDM SupplyCustomRequestParameters command (Section 13) 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. The device shall verify that none of the supplied ETags matches the current state of the targeted schema data instance before effecting the operation.

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 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 in either 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 (Section 13.8). The MC shall wait for the response to the SupplyCustomRequestParameters command before sending the main operation command (from Section 13) request.

Following completion of the main operation, the MC shall check the HaveCustomResponseParameters flag to see if the device is supplying custom response headers. If and only if the flag is set to a value of logical true, the MC shall use the RetrieveCustomResponseParameters command (Section 13.9) to recover them from the device. The MC shall then append the recovered headers to the HTTP/HTTPS 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 device shall generate a digest of the schema data instance after each modification to the data in accordance with RFC 7232. When the MC sends a GET or HEAD operation to the device via a PLDM RedfishRead (Section 13.1) or RedfishHead (Section 13.7) command, the device shall populate the ETag field in the response message with this digest. 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.

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 (Section 11.3)

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 usedthe MC may retrieve the schema reference directly from the PDR encapsulating the instance of the schema data by invoking the PLDM GetSchemaURI command (Section 11.3)

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 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 create operation.

The parameter for this header is a URI.

In order to support this header, the device shall populate the NewSequenceNumber field of its response to the RedfishCreate command (Section 13.1) with the array index of the newly created collection element. Upon receipt, the MC shall combine this sequence number with the topology information contained in the Redfish Resource PDRs for the targeted PDR up through the device root to create a local URI portion that it shall then combine with its external management URI for the device to build a complete URI for the newly-added collection element. The MC shall then populate this header with the resulting URI.

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 device shall mark the CacheAllowed field in the RedfishRead response message (Section 13.1) with an indication of the caching status of data read. In order to support this header for HTTP/HTTPS HEAD operations, the device shall mark the CacheAllowed field in the RedfishHead response message (Section 13.7) with an indication of the caching status of data that would be contained in a RedfishRead operation performed on the same object. In either case, the MC shall then populate this header with the supplied information.

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 device specifically for the reason of being a disallowed operation, giving the ERROR_NOT_ALLOWED completion code (Section 7.4). The MC shall additionally provide an Allow Response header in response to every Redfish HTTP/HTTPS HEAD operation.

In order to support this header, the device shall as part of responding to Task trigger commands, populate the PermissionFlags field with an indication of the operations that are permitted. The MC shall then 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 Redfish HTTP/HTTPS operation that when conveyed to the device results in any transient failure (ERROR_NOT_READY, see Section 7.4).

The parameter for this header is a representation defined in RFC 7231 of the minimum amount of time the client should wait before retrying the request.

In order to support this header, the device shall provide a deferral timeframe in response to any operation that results in a transient failure. This shall be done via the DeferralTimeframe and DeferralUnits fields of the RedfishTaskInit response message (See Section 13.1). The MC shall populate this header with the value supplied by the device. The MC and 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 HTTP/HTTPS GET operations. The MC is not required to support any query options not listed here. In particular, the $filter query option is too large a burden for most RDE devices to support.

Table 27 – Redfish Operation Request Query Options

Query Option	Section	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 RedfishRead command (Section 13.3) 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.

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.

To support this query option, the MC shall supply the $top parameter in the CollectionTop field of the RedfishRead command (Section 13.3) 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. The 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.

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 link 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 either of two wildcards. The first wildcard, an asterisk (*), means that all links should be expanded. The second wildcard, a dot (.), means that all links except those found in the Links 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 $level and $select qualifiers.

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.

The parameter for this query option is an integer representing the number of schema links to expand into.

To support this parameter, the MC 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 bejSchemaLink (Section 5.3.19). The corresponding value of the node represents the Schema Instance ID for the PDR representing the data to embed within the structure of data already read. The $levels qualifier dictates the depth of recursion for this process. If no $level qualifier is present, the MC shall interpret this as equivalent to $levels=1.

7.2.4.3.5 $select Query option and 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 shall perform the GET operation normally up to the point of retrieving BEJ-formatted data from the device. When decoding the BEJ data, however, the MC shall silently discard any property not part of the $select list.

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.

7.2.5 Events

An Event is an abstract representation of any happening that transpires in the context of the 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 Devices to report that a long-running Task has finished executing and that the MC should retrieve Task results. The data for these events includes elements such as the Task identifier.

There are two methods by which the MC might know that events are available. If the device, the MC, and the underlying hardware all support asynchronous commands issued from the device, it shall use the PLDM Monitoring and Control PlatformEventMessage command (See DSP0248) to inform the MC. The device and the MC must both have set their respective asynchrony support flags to true in the NegotiateRedfishParameters (Section 11.1) registration exchange for this to be an option.

NOTE: the MC is responsible for knowing whether the underlying hardware supports asynchronous message traffic.

The alternative to asynchronous notifications of Event availability from the device is that the MC may poll for them instead. To do so, it uses the QueryRedfishEvents (Section 11.3) command. 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. The MC may use the MultipartReceive command (Section 14.2) to obtain the payload for a Redfish Message Event.

Once the MC has retrieved an Event and any associated payload, it shall then use the RedfishEventComplete command (Section 12.2) to inform the device that it may safely discard any Event records it has constructed for the Event.

Handling of Task Executed Events is described with Tasks in Section 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. MCs shall be aware that the version of the standard Redfish Event schema and the list of OEM extensions to that schema – and their versions – may all vary by PDR.

For a more detailed view of the Event lifecycle, see Section 9.2.

Note: Events are optional in standard Redfish; however, support for Task Executed Events is mandatory in this specification.

7.2.5.1 [MC] Event Subscriptions

In Redfish, a client may request to be notified whenever an Event occurs. 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.

Event types are global across all schemas; there is no provision at this time in Redfish for a client to subscribe to just one schema at a time. Further, there is generally no capacity for a 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 service 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 a 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, of course, from BEJ to JSON) to the client as a standard Redfish Event service provider.

7.2.6 Tasks

In PLDM for Redfish Device Enablement, every Redfish HTTP/HTTPS operation is effected by a Task. Most Tasks, once sent to the 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 a device to complete a Task, it requires more time than the available window within which the device is required to send a response. In this case, the device has two possible paths to follow. If the current number of extant Tasks is less than the device/MC capability intersection (as determined from the call to NegotiateRedfishParameters, see Section 11.1), the device shall mark the Task as long-running and execute it asynchronously. Otherwise, the device shall return ERROR_CANNOT_CREATE_TASK in its response message to indicate that this is a lengthy operation but no new task slots are available (See Section 7.4).

While the internal data structures used by a device to manage a long-running Task are outside the scope of this specification, they should include at a minimum the TaskID assigned (usually by the MC) when the Task was first created. This allows the MC to reference the Task in subsequent commands to kill the Task (RedfishTaskKill, Section 13.12) or query as to its status (RedfishTaskStatus, Section 13.11).

When the device finishes execution of a long-running Task, it generates a Task Executed Event to inform the MC of this status change. The MC can then retrieve the results and forward them to the client. To mark the Task as complete and allow the device to discard any internal data structures used to manage the Task, the MC calls RedfishTaskComplete (Section 13.10).

For a more detailed overview of the Task lifecycle from the MC's perspective, see Section 7.2.4.1.1.3. A detailed flowchart of the Task lifecycle may be found in Section 9.1.1, and a finiste state machine for the Task lifecycle (from the device's perspective) may be found in Section 9.1.2.

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 Error Completion Codes

PLDM completion codes for Redfish Device Enablement that are beyond the scope of PLDM_BASE_CODES in DSP0240 are defined in the list below. The usage of individual error completion codes are defined within each of the PLDM command sections.

Table 28 – PLDM Redfish Device Enablement Completion Codes

Value	Name	Description
Various	PLDM_BASE_CODES	Refer to DSP0240 for a full list of PLDM Base Code Completion values that are supported.
0x80	ERROR_BAD_CHECKSUM	A transfer failed due to a bad checksum and should be restarted
0x81	ERROR_BAD_LOCATOR	An invalid BEJ Locator was supplied for a Task-based command
0x82	ERROR_BAD_RESOURCE_ID	A reference was made to a schema instance ID that does not correspond to a Redfish Resource PDR
0x83	ERROR_BAD_XFER_HANDLE	The MC supplied an invalid transfer handle when attempting a multipart transfer
0x84	ERROR_CANNOT_CREATE_TASK	A Task-based command failed because the device could not instantiate another long-running Task at this time
0x85	ERROR_NO_DATA	The MC attempted to progress a Task but neglected to send a required request payload or to receive a required response payload
0x86	ERROR_NO_SUCH_EVENT	A reference was made to an Event that does not exist
0x87	ERROR_NO_SUCH_TASK	A Task-based command other than initialization was attempted with a TaskID that is not currently active
0x88	ERROR_NOT_ALLOWED	The client and/or MC is not allowed to perform the requested operation

0x89	ERROR_NOT_COLLECTION	A Create or Delete operation was attempted at a location that does not correspond to a Redfish collection
0x8A	ERROR_NOT_ACTION	An Action operation was attempted at a location that does not correspond to a Redfish action
0x8B	ERROR_STRING_FORMAT	A command was issued containing a text string in a format that the recipient cannot interpret
0x8C	ERROR_TASK_ABANDONED	A Task-based command other than completion was attempted with a Task that has timed out waiting for the MC to progress it in the Task lifecycle
0x8D	ERROR_TASK_EXECUTED	An attempt was made to kill a Task that has already finished execution
0x8E	ERROR_TASK_EXISTS	A Task initialization was attempted with a TaskID that is currently active
0x8F	ERROR_TASK_FAILED	A Task-based command other than completion was attempted with a Task that has encountered an error in the Task lifecycle
0x90	ERROR_TASK_INEVITABLE	An attempt to kill a Task could not be completed because the Task cannot be aborted
0x91	ERROR_TASK_NOT_FINISHED	An attempt was made to retrieve results from a Task that has not yet completed
0x92	ERROR_UNSUPPORTED_PROPERTY	An attempt was made to write to a property that the device does not support
0x93	ERROR_WRITE_TO_READ_ONLY	An attempt was made to write to a property that is marked as read-only

7.5 Timing Specification

Table 29 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 and number of retries shall also be followed.

Table 29 – Timing Specification

Timing specification	Symbol	Min	Max	Description
PLDM Base Timing	PNx PTx			Refer to DSP0240 for the details on these timing values which are applicable to PLDM message timeouts where a response is not received by the UA or FD after sending a request.
First Response Byte	T_{resp_0}	0	500 msecs	Timing between when any command request is sent and the first byte of the response must be received
Task Completion	T_{task}	0	Open ended	Allowable duration for a long-running task (Section 7.2.6)

Time until active	T_{active}	0	15 seconds	Time between when a device receives full power and when it must be able to respond to the NegotiateRedfishParameters command
Max retries	N_{retry}	3	5	Maximum number of times a device may indicate that it is transiently busy (via responding ERROR_NOT_READY) and that the MC must retry a request later (See the DeferalTimeFrame response message field from the RedfishTaskInit command in Section 13.1.)
MC abandonment	$T_{abandon}$	120 seconds	240 seconds	Time between when the device is ready to advance a Task through the Task lifecycle and when the MC must have initiated the next step. If the MC fails to do so, the device may consider the Task as abandoned.
Read retries	N_{read}	3	7	Number of times the MC will repeat a GET command that results in consistency failures before reporting an error to the client

8 Binary Encoded JSON (BEJ)

This section defines a binary encoding of Redfish JSON data that will be used for communicating with devices. At its core, BEJis 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 trade off 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.

3) There is no need to support multiple encoding techniques; there is therefore no need to distinguish which encoding technique is in use.

4) Schema information – such as the names of data items – does not need to be encoded into BEJ because the recipient can use a priori 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.

5) 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, 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

8.2.1 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.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.

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

Table 30 – Sequence Number Dictionary Indication

Bit Pattern	Dictionary
00	Major Schema
01	Extension to major schema for first OEM
10	Annotation
11	OEM extension to major schema for OEM other than first

Further details for handling of OEM Extensions beyond the first may be found in Section 5.3.5.

8.2.2 Format

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

8.2.3 Length

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

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 31 –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
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

If the deferred binding flag (flag bit 1) is set, the string encoded in the value tuple element contains substitution macros that the MC is to supply on behalf of the device when populating a message back to the client. See Section 8.3 for more details.

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 device discovery and registration. In practice, the only way for a 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 device may use to ask the MC to supply these bits of information on its behalf.

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 an inefficient waste of MC processing time. Instead, the 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 binding as listed in Table 32.

Table 32 – BEJ Deferred Binding Substitution Parameters

Macro	Data to be substituted	Example substitutions
%%	A single % character	%
%INSTANCE_ID%	The instance ID portion of the provide ID	1 437XR1138R2
%DEVICE_PREFIX%	The URI prefix leading up to the device	/redfish/v1/Systems/437XR1138R2
%SCHEMA_TYPE%	The full schema type. This macro is shorthand for: %schema_parent_name%.%schema_version%.%schema_name%	Storage.v1_0_0.StorageController
%SCHEMA_NAME%	The name of the current schema, from the SupportedSchemaName field of the Redfish Resource PDR	StorageController
%SCHEMA_PARENT_NAME%	The name of the schema in which the definition for the present schema is located	Storage
%SCHEMA_VERSION%	The supported version string for the schema, in the format “v<major>_<minor>_<release>”, from the SupportedSchemaVersion field of the Redfish Resource PDR	v1_0_0
%ACTION.<n>%	The name of the nth action, one-based, from the Schema Action PDR linked to a Redfish Resource PDR, or “InvalidAction.<n>” if no such action exists	SetEncryptionKey InvalidAction.300
%OEM_NAME.<n>%	The name of the nth OEM, one-based, from the OEMName field of the Redfish Resource PDR, or “InvalidOEM.<n>” if no such OEM Extension exists	Contoso InvalidOEM.5
%OEM_VERSION.<n>%	The supported version string for the nth OEM extension to the schema, in the format “v<major>_<minor>_<release>”, from the OEMSchemaVersion[n] field of the Redfish Resource PDR, or “vInvalid” if no such OEM extension exists”	v1_2_1 vInvalid
%LINK.P<resource-ID>%	The MC-assigned URI of a provider defined resource, or /invalid.P<resource-ID> if unrecognized	/invalid.P123
%LINK.SELF%	The MC-assigned link to the current resource	/redfish/v1/Systems/437XR1138R2/Storage/1
%LINK.SYSTEM%	The MC-assigned link to the ComputerSystem resource within which the device is located	/redfish/v1/Systems/437XR1138R2
%LINK.CHASSIS%	The MC-assigned link to the Chassis resource within which the device is located	/redfish/v1/Chassis/1U

%TARGET.<n>%	The MC-assigned target URI for the nth major schema action from the Redfish Action PDR linked to a Redfish Resource PDR, or "/invalid.<n>" if no such action exists	/redfish/v1/Systems/437XR1138R2/Storage/1/Actions/Storage.SetEncryptionKey /invalid.6
%TARGET.O<oem>.<n>%	The MC-assigned target URI for the nth action in the oemth OEM extension to the major schema from the Redfish Action PDR linked to a Redfish Resource PDR, or "/invalid.o<oem>.<n>" if no such action exists	/redfish/v1/Systems/437XR1138R2/Storage/1/Actions/Storage.TiltPlatterSpinAxis /invalid.o3.6
%UEFI_DEVICE_PATH%	The UEFI Device Path assigned to the device by the MC and/or BIOS	PciRoot(0x0)/Pci(0x1,0x0)/Pci(0x0,0x0)/Scsi(0xA, 0x0)
Anything else bracketed in % characters, or any macro lacking a closing % character	None – the MC shall pass the sequence exactly as found	%device_prefix %unknown_substititution%

Any substitution macro may be modified by appending ".P<p>" to the text between the % characters; this redirects the lookup to be relative to the Redfish Resource PDR with ResourceID <p>. For example, %SCHEMA_NAME% directs the MC to substitute the type of the current schema, but %SCHEMA_NAME.P34% redirects this substitution to the schema type for Redfish Resource PDR 34.

8.4 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 device would do so.

8.4.1 Example Dictionary

For these examples, we use the following data dictionary (converted to tabular form).

Table 33 – Example Dictionary (Tabular Form)

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

NOTE: This is not a published DMTF Redfish schema.

8.4.2 Example Encoding

For this example, we start with the following data (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.

```
<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. For encoding purposes, we encode the fact that these sequence numbers came from a data dictionary by shifting them left two bits to insert “00” as the low order bits per Section 8.2.1.

```
<Item name="DummySimple" type="set" seqno="0">
   <Item name="ChildArrayProperty" type="array" seqno="0">
      <Item name="array element 0" seqno="0">
         <Item name="AnotherBoolean" type="boolean" value="true" seqno="0"/>
         <Item name="LinkStatus" type="enum" enumtype="String" seqno="4">
            <Enumeration value="NoLink" seqno="8"/>
         </Item>
      </Item>
      <Item name="array element 1" seqno="4">
         <Item name="LinkStatus" type="enum" enumtype="String" seqno="4">
            <Enumeration value="LinkDown" seqno="0"/>
          </Item>
       </Item>
   </Item>
   <Item name="Id" type="string" value="Dummy ID" seqno="4"/>
   <Item name="SampleIntegerProperty" type="integer" value="12" seqno="8"/>
</Item>
```

Once the sequence numbers are 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="4">
            <Enumeration seqno="8"/>
         </Item>
      </Item>
      <Item seqno="4">
         <Item type="enum" enumtype="String" seqno="4">
            <Enumeration seqno="0"/>
```

```
          </Item>
       </Item>
   </Item>
   <Item type="string" value="Dummy ID" seqno="4"/>
   <Item type="integer" value="12" seqno="8"/>
</Item>
```

The next step is to convert everything into BEJ SFLV Tuples. Per Section 5.3.11, the value of an enumeration is the sequence number for the selected option.

```
{0x00, set, [length placeholder], value={count=3,
   {0x00, array, [length placeholder], value={count=2,
      {0x00, set, [length placeholder], value={count=2,
         {0x00, boolean, [length placeholder], value=true}
         {0x04, enum, [length placeholder], value=2}
      }}
      {0x04, set, [length placeholder], value={count=1,
         {0x04, enum, [length placeholder], value=0}
      }}
   }}
   {0x04, string, [length placeholder], value="Dummy ID"}
   {0x08, integer, [length placeholder], value=12}
}}
```

We now encode the formats and the leaf nodes, following Table 8. For sets and arrays, the value encoding count prefix is a Non-Negative Integer; we can encode that now as well per Table 4. Note the null terminator for the string.

```
{0x00, 0x00, [length placeholder], {0x01 0x03,
   {0x00, 0x01, [length placeholder], {0x01 0x02,
      {0x00, 0x00, [length placeholder], {0x01 0x02,
         {0x00, 0x07, [length placeholder], 0xFF}
         {0x04, 0x04, [length placeholder], 0x01 0x08}
      }}
      {0x04, 0x00, [length placeholder], {0x01 0x01,
         {0x04, 0x04, [length placeholder], 0x01 0x00}
      }}
   }}
   {0x04, 0x05, [length placeholder],
      0x44 0x75 0x6D 0x6D 0x79 0x20 0x49 0x44 0x00}
   {0x08, 0x03, [length placeholder], 0x0C}
}}
```

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

```
{0x00, 0x00, [length placeholder], {0x01 0x03,
   {0x00, 0x01, [length placeholder], {0x01 0x02,
      {0x00, 0x00, [length placeholder], {0x01 0x02,
         {0x00, 0x07, 0x01 0x01, 0xFF}
         {0x04, 0x04, 0x01 0x02, 0x01 0x08}
      }}
      {0x04, 0x00, [length placeholder], {0x01 0x01,
         {0x04, 0x04, 0x01 0x02, 0x01 0x00}
      }}
   }}
   {0x04, 0x05, 0x01 0x09,
      0x44 0x75 0x6D 0x6D 0x79 0x20 0x49 0x44 0x00}
   {0x08, 0x03, 0x01 0x01, 0x0C}
}}
```

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

```
{0x00, 0x00, [length placeholder], {0x01 0x03,
   {0x00, 0x01, [length placeholder], {0x01 0x02,
```

```
      {0x00, 0x00, 0x01 0x0D, {0x01 0x02,
         {0x00, 0x07, 0x01 0x01, 0xFF}
         {0x04, 0x04, 0x01 0x02, 0x01 0x08}
      }}
      {0x04, 0x00, 0x01 0x08, {0x01 0x01,
         {0x04, 0x04, 0x01 0x02, 0x01 0x00}
      }}
   }}
   {0x04, 0x05, 0x01 0x09,
      0x44 0x75 0x6D 0x6D 0x79 0x20 0x49 0x44 0x00}
   {0x08, 0x03, 0x01 0x01, 0x0C}
}}
```

Next, the array:

```
{0x00, 0x00, [length placeholder], {0x01 0x03,
   {0x00, 0x01, 0x01 0x1F, {0x01 0x02,
      {0x00, 0x00, 0x01 0x0D, {0x01 0x02,
         {0x00, 0x07, 0x01 0x01, 0xFF}
         {0x04, 0x04, 0x01 0x02, 0x01 0x08}
      }}
      {0x04, 0x00, 0x01 0x08, {0x01 0x01,
         {0x04, 0x04, 0x01 0x02, 0x01 0x00}
      }}
   }}
   {0x04, 0x05, 0x01 0x09,
      0x44 0x75 0x6D 0x6D 0x79 0x20 0x49 0x44 0x00}
   {0x08, 0x03, 0x01 0x01, 0x0C}
}}
```

Finally, the outermost set:

```
{0x00, 0x00, 0x01 0x37, {0x01 0x03,
   {0x00, 0x01, 0x01 0x1F, {0x01 0x02,
      {0x00, 0x00, 0x01 0x0D, {0x01 0x02,
         {0x00, 0x07, 0x01 0x01, 0xFF}
         {0x04, 0x04, 0x01 0x02, 0x01 0x08}
      }}
      {0x04, 0x00, 0x01 0x08, {0x01 0x01,
         {0x04, 0x04, 0x01 0x02, 0x01 0x00}
      }}
   }}
   {0x04, 0x05, 0x01 0x09,
      0x44 0x75 0x6D 0x6D 0x79 0x20 0x49 0x44 0x00}
   {0x08, 0x03, 0x01 0x01, 0x0C}
}}
```

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

```
0x00 0x00 0x01 0x37 0x01 0x03 0x00 0x01
0x01 0x1F 0x01 0x02 0x00 0x00 0x01 0x0D
0x01 0x02 0x00 0x07 0x01 0x01 0xFF 0x04
0x04 0x01 0x02 0x01 0x08 0x04 0x00 0x01
0x08 0x01 0x01 0x04 0x04 0x01 0x02 0x01
0x00 0x04 0x05 0x01 0x09 0x44 0x75 0x6D
0x6D 0x79 0x20 0x49 0x44 0x00 0x08 0x03
0x01 0x01 0x0C
```

8.4.3 Example Decoding

The decoding process is largely the inverse of the encoding process. For this example, we start with the final encoded data from Section 8.4.2:

```
0x00 0x00 0x01 0x37 0x01 0x03 0x00 0x01
0x01 0x1F 0x01 0x02 0x00 0x00 0x01 0x0D
```

```
0x01 0x02 0x00 0x07 0x01 0x01 0xFF 0x04
0x04 0x01 0x02 0x01 0x08 0x04 0x00 0x01
0x08 0x01 0x01 0x04 0x04 0x01 0x02 0x01
0x00 0x04 0x05 0x01 0x09 0x44 0x75 0x6D
0x6D 0x79 0x20 0x49 0x44 0x00 0x08 0x03
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=0x00, F=0x00, L=0x01 0x37, V={0x01 0x03,
   {S=0x00, F=0x01, L=0x01 0x1F, V={0x01 0x02,
      {S=0x00, F=0x00, L=0x01 0x0D, V={0x01 0x02,
         {S=0x00, F=0x07, L=0x01 0x01, V=0xFF}
         {S=0x04, F=0x04, L=0x01 0x02, V=0x01 0x08}
      }}
      {S=0x04, F=0x00, L=0x01 0x08, V={0x01 0x01,
         {S=0x04, F=0x04, L=0x01 0x02, V=0x01 0x00}
      }}
   }}
   {S=0x04, F=0x05, L=0x01 0x09,
      V=0x44 0x75 0x6D 0x6D 0x79 0x20 0x49 0x44 0x00}
   {S=0x08, F=0x03, L=0x01 0x01, V=0x0C}
}}
```

Once the tuple boundaries are understood, the length and count bytes are no longer needed:

```
{S=0x00, F=0x00, V={
   {S=0x00, F=0x01, V={
      {S=0x00, F=0x00, V={
         {S=0x00, F=0x07, V=0xFF}
         {S=0x04, F=0x04, V=0x01 0x08}
      }}
      {S=0x04, F=0x00, V={
         {S=0x04, F=0x04, V=0x01 0x00}
      }}
   }}
   {S=0x04, F=0x05, V=0x44 0x75 0x6D 0x6D 0x79 0x20 0x49 0x44 0x00}
   {S=0x08, F=0x03, V=0x0C}
}}
```

The next step is to decode format tuple bytes using Table 8. This will tell us how to decode the value data:

```
{S=0x00, set, V={
   {S=0x00, array, V={
      {S=0x00, set, V={
         {S=0x00, boolean, V=0xFF}
         {S=0x04, enum, V=0x01 0x08}
      }}
      {S=0x04, set, V={
         {S=0x04, enum, V=0x01 0x00}
      }}
   }}
   {S=0x04, string, V=0x44 0x75 0x6D 0x6D 0x79 0x20 0x49 0x44 0x00}
   {S=0x08, integer, V=0x0C}
}}
```

We now decode value data:

```
{S=0x00, set, {
   {S=0x00, array, {
      {S=0x00, set, {
         {S=0x00, boolean, true}
         {S=0x04, enum, <value 8>}
```

```
        }}
        {S=0x04, set, {
           {S=0x04, enum, <value 0>}
        }}
    }}
    {S=0x04, string, "Dummy ID"}
    {S=0x08, integer, 12}
}}
```

Next we use the dictionary to replace 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 a device's response to a client, but a device may not need this step):

```
{
    "DummySimple" : {
        "ChildArrayProperty" : [
            {
                "AnotherBoolean" : true,
                "LinkStatus" : "NoLink"
            },
            {
                "LinkStatus" : "LinkDown"
            }
        ],
        "Id" : "Dummy ID",
        "SampleIntegerProperty" : 12
    }
}
```

8.5 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 device supports data. This enables a locator to be unambiguous.

As an example, consider a locator, encoded for the example dictionary of Section 8.4.1:

0x01 0x08 0x01 0x00 0x01 0x00 0x01 0x0C 0x01 0x04

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 two low-order bits of each sequence number reference the schema to

which it refers; in this case, the pattern 00b 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 DummySchema
- The second zero points to the first child of DummySchema, or ChildArrayProperty
- The three points to the fourth element in the ChildArrayProperty array, an anonymous instance of the array type
- The one points to the second child inside the ChildArray element type, or LinkStatus

9 Operational Behaviors

This clause describes the operational behavior for Tasks and Events.

9.1 Task Lifecycle

The following sections present the Task lifecycle from two perspectives, first from a Task-centric viewpoint and then from the device perspective. MC and device implementations of RDE shall comply to the sequences presented here.

9.1.1 Task Overview Flowcharts

The following Tables describe the operating behavior for MCs and Devices over the lifecycle of Tasks from a Task-centric perspective.Table 34 and Table 35 detail the information presented visually in Figure 2 and Figure 3, respectively.

Table 34 – Task Lifecycle Overview

Current State	Description	Condition	Next State
1	The lifecycle of a Task begins when the MC receives an HTTP/HTTPS operation from the client and sends the RedfishTaskInit command to the device.	For any Redfish Read (HTTP/HTTPS GET) operations	2
		For any other operation	3
		If the MC decides to kill the task after initialization	6
2	For Read operations, the MC uses the GetSchemaInstanceETag command to record a digest snapshot	Unconditional	3
3	The MC checks the HTTP/HTTPS operation to see if it contains JSON payload data to be transferred to the device. If so, it performs a BEJ encoding of this data.	BEJ data to be sent to the device	4
		No BEJ data to send	5
4	The MC uses the MultipartSend command to send BEJ-encoded payload data to the device	The last chunk of payload data has been sent	5
		More data remains to be sent	4
5	Once any payload has been sent down to the device, the MC has one last chance to abort the operation before triggering it by sending a RedfishKillTask command request message to the device	If the MC sends a kill task message	6
		If the MC does not	7

6	If the device receives a kill task message before receiving the command to trigger the operation, it knows that the operation will never be triggered	Unconditional	10
7	The MC sends the appropriate trigger command (RedfishCreate, RedfishRead, RedfishWrite, RedfishAction, RedfishDelete, or RedfishHead) for the operation to the device	Unconditional	8
8	When the device receives the operation trigger, it is cleared to begin the operation. In most cases, the device will be able to complete the operation quickly (and be able to respond within T_{resp_0}; see Section 7.5)	If the device is able to complete the operation “quickly”, or if the operation cannot be completed (such as an invalid operation)	10
		Otherwise	9
9	If the device was not able to complete the operation quickly enough it becomes a long-running task. See Figure 3	Unconditional	2, Long Running Task Flowchart
10	Once the MC knows that the operation has been completed or killed, it checks the operation response to see if there is a response payload	A response payload exists	11
		No response payload	12
11	The MC retrieves BEJ-formatted response payload data from the device via the MultipartReceive command and decodes it into a JSON response for the client	If another chunk of data remains to be retrieved from the device	11
		Once all data has been transferred	12
12	If the operation was a read (HTTP/HTTPS GET), the MC compares the current ETag with the one it checked in Step 2.	If this wasn’t a read operation	14
		Read operation but tags match	14
		Read operation, tags do not match	13
13	A mismatch in the ETags indicates a potential consistency violation in the data read. The MC must discard the results and try again	If the MC has exceeded its read consistency failure retry threshold	14
		Otherwise	2
14	The MC sends a RedfishTaskComplete message to the device to inform it that the task may be finalized. Upon receipt, the device may discard any data it has stored for the task	n/a	-

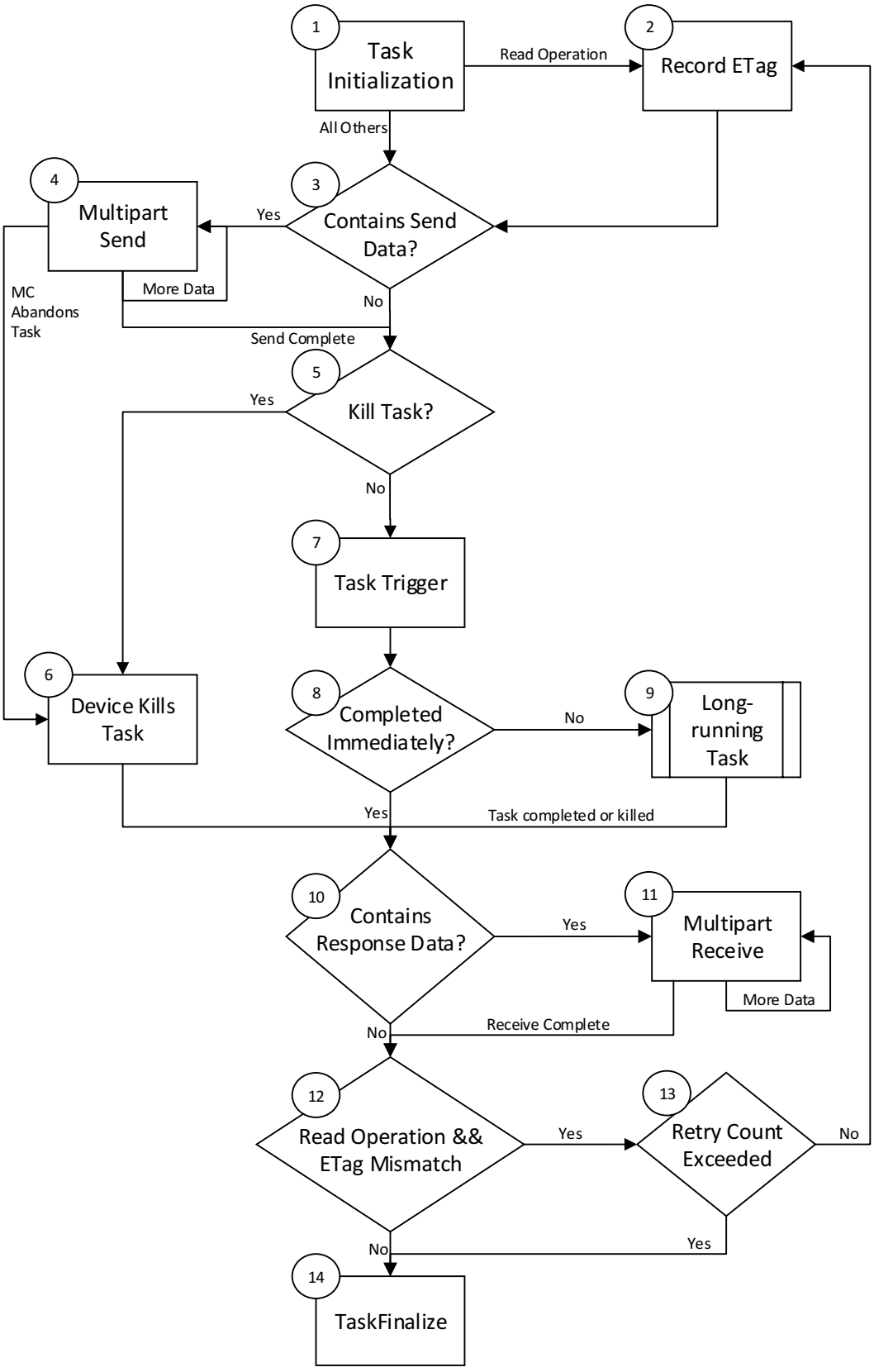
1473 **Table 35 – Long Running Task State Machine**

Current State	Description	Condition	Next State
1	The sublifecycle of a long-running Task begins when the MC sends a trigger to the device to effect the operation. (This corresponds to Step 7 in Figure 2.)	Unconditional	2
2	When the device receives the operation trigger, it is cleared to begin the operation. For long-running Tasks, the device will not be able to complete the operation quickly (and be able to respond within T_{resp_0}; see Section 7.5)	If the device is able to complete the Task quickly (not a long-running Task)	12
		Otherwise	3
3	The device runs the Task asynchronously	Unconditional	5

4	The MC may issue a RedfishTaskStatus command at any time to the device.		5
5	The device must be ready to respond to a RedfishTaskStatus command while running a Task asynchronously	Task status request received	6
		No Task status request received	8
6	The device sends a response to the RedfishTaskStatus command to provide a status update	Unconditional	3
7	The MC may issue a RedfishTaskKill command at any time to the device	Unconditional	8
8	The device must be ready to respond to a RedfishTaskKill command while running a Task asynchronously	Task kill request received	9
		No Task kill request received	10
9	If the device receives a Task kill request, it may or may not be able to abort the Task. This is a device-specific decision about whether the Task has crossed a critical boundary and must complete	Device cannot stop the Task	10
		Device can stop the Task	11
10	The device should eventually complete the Task	If the Task has been completed	12
		If the Task has not been completed	3
11	The device aborts the Task in response to a request from the MC	Unconditional	14
12	Once the Task is complete, the device generates a Task completion Event to announce this status update to the MC.	Unconditional	13
13	After receiving the Task completion Event, the MC uses the RedfishTaskFollowup command to retrieve the final response to the operation request	Unconditional	14
14	Once the MC knows that the operation has been completed or killed, it checks the operation response to see if there is a response payload (This corresponds to Step 10 in Table 36)	See Step 10 in Table 36	See Step 10 in Table 36

1474

1475



1476

1477 **Figure 2 -- Redfish Task Lifecycle Overview**

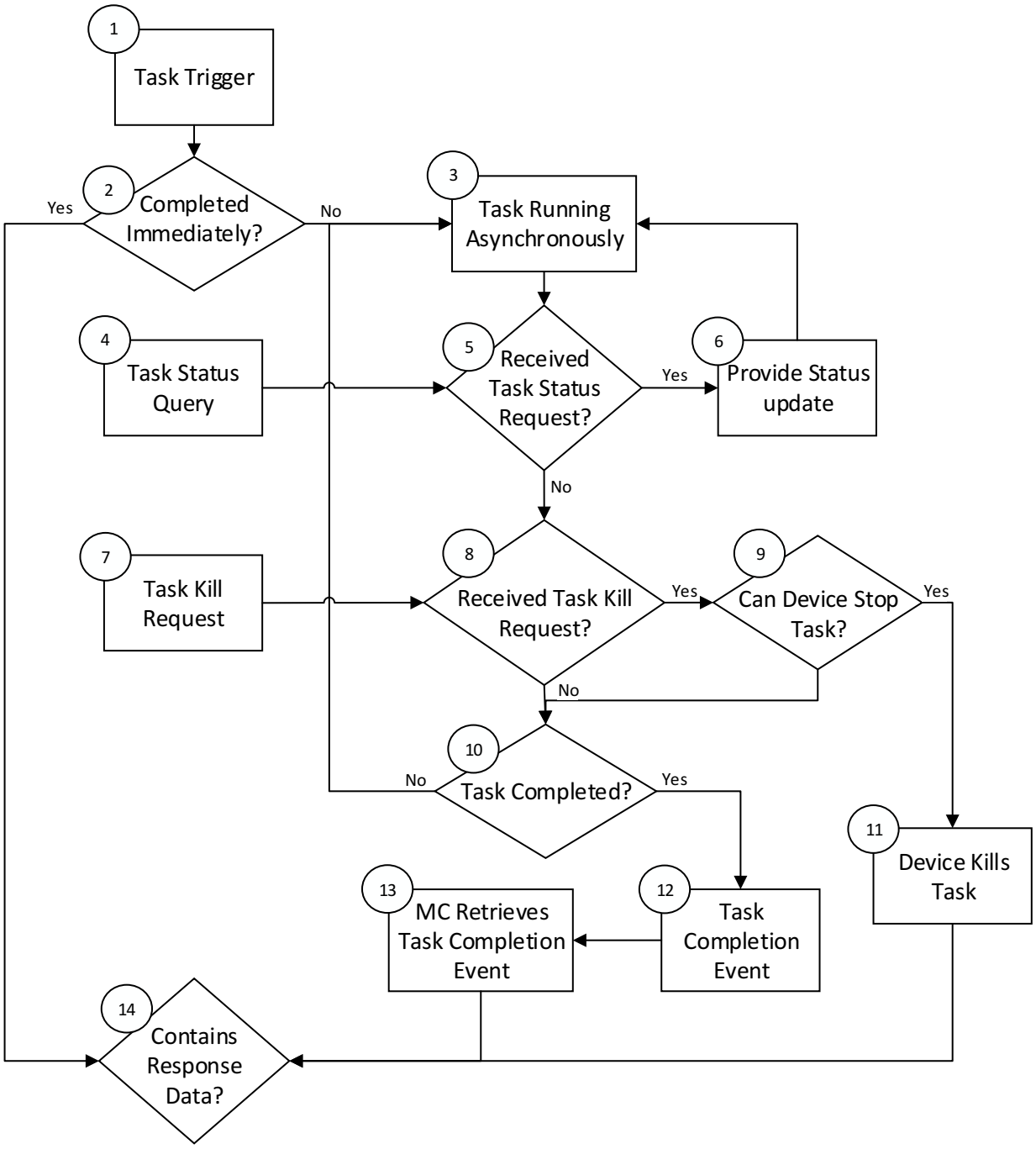


Figure 3 – Redfish Long Running Task Overview

9.1.2 [Dev] Device Perspective State Machine

The following sections describe the operating behavior for Devices over the lifecycle of Tasks from a Device-centric perspective. Table 36 details the information presented visually in Figure 4.

9.1.2.1 State Definitions

The following states shall be implemented by the device.

- INACTIVE
  - INACTIVE is the default state in which the device shall start after initialization. In this state, the device is not processing a Task as it has not received a RedfishTaskInit command from the MC
- INITIALIZED
  - After receiving the RedfishTaskInit command, the device moves to this state to prepare for a potential incoming payload and await additional Task-specific parameters
- NEEDING PAYLOAD
  - When the Task contains a BEJ-encoded JSON payload, the device remains in this state until it has received the entire payload from the MC via the MultipartSend command
- TRIGGERED
  - Once the device receives a Task-specific trigger command (one of RedfishCreate, RedfishRead, RedfishWrite, RedfishAction, RedfishDelete, or RedfishHead), and the corresponding Task-specific parameters, it is authorized to begin execution of the Task.
- LONG RUNNING
  - If the device cannot complete the Task within the timeframe needed for the response to the trigger command, the Task becomes a long-running Task that the device executes asynchronously
- EXECUTED
  - This state marks the point in time where the device has finished executing the Task
- HAVING RESPONSE
  - When execution of the Task produces a response, the device remains in this state until the MC has recovered the entire response via the MultipartReceive command
- COMPLETED
  - The MC acknowledges receipt of any Task response data by issuing the RedfishTaskComplete command. When the device receives this command, it may discard any internal records or state it has maintained for the Task
- ABANDONED
  - If MC fails to progress the Task through this state machine, the device may abort the Task and mark it as abandoned
- FAILED
  - The MC has explicitly killed the Task or an eror prevented execution of the Task

9.1.2.2 Task Lifecycle State Machine

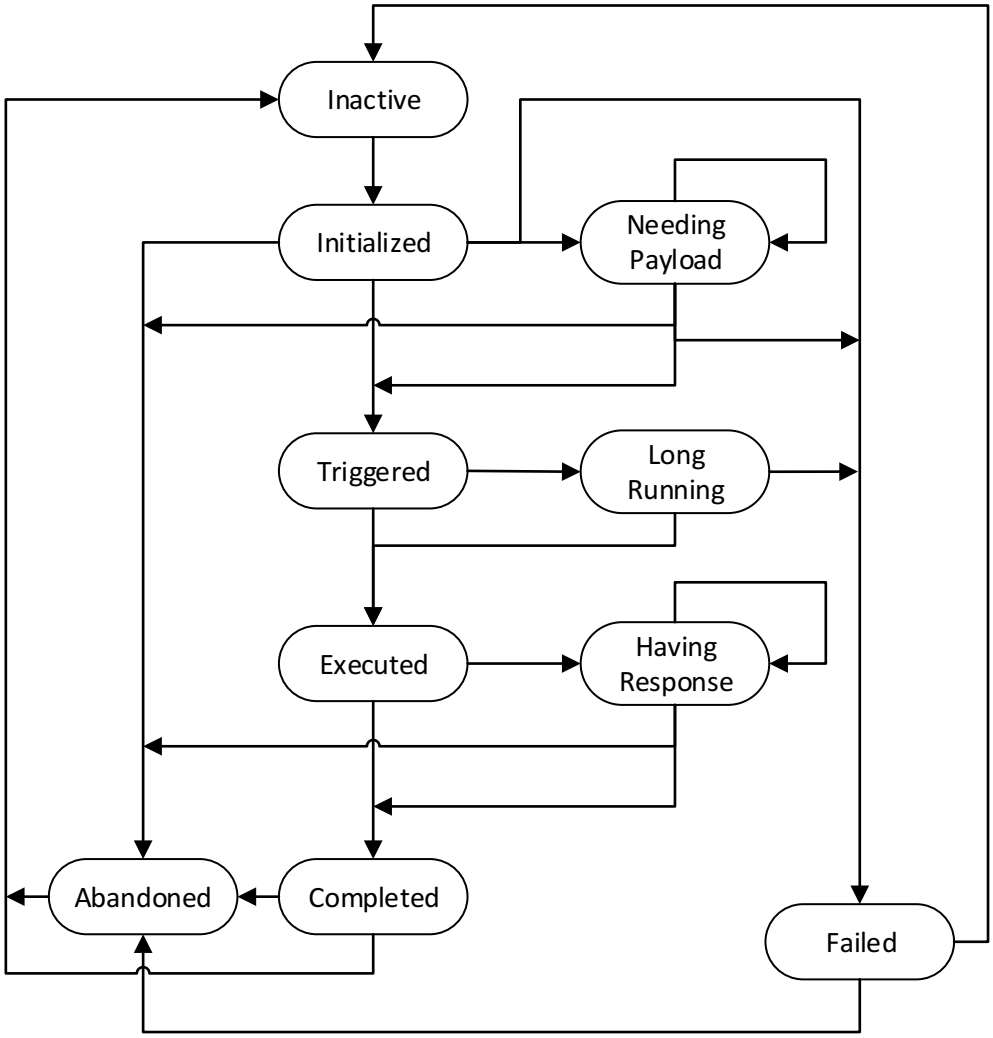
The below diagram illustrates the state transitions the device shall implement. Each bubble represents a particular state as defined in the previous section. Upon initialization, system reboot, or a device reset the provider device shall enter the INACTIVE state.

Table 36 – Task Lifecycle State Machine

Current State	Trigger	Response	Next State
INACTIVE	RedfishTaskInit, valid request	Success	INITIALIZED
	RedfishTaskInit, device not ready	ERROR_NOT_READY	INACTIVE
	RedfishTaskInit, request contains an error	Various, depending on the specific error encountered	INACTIVE
	RedfishTaskStatus	TASK_INACTIVE	INACTIVE
	Any other command	ERROR_NO_SUCH_TASK	INACTIVE
INITIALIZED	Task initialization indicates presence of payload	None	NEEDING PAYLOAD
	$T_{abandon}$ timeout waiting for trigger operation	None	ABANDONED

	RedfishTaskKill	Success	FAILED
	Trigger, Task initialization does not indicate presence of payload	Success	TRIGGERED
	Trigger, Task initialization indicates presence of payload, none received	ERROR_NO_DATA	KILLED
	RedfishTaskStatus	TASK_INITIALIZED	INITIALIZED
	Any other command	Error	INITIALIZED
NEEDING PAYLOAD	MultipartSend	Success	NEEDING PAYLOAD
	Trigger, last chunk of payload data received	Success	TRIGGERED
	Trigger, not all payload data received	ERROR_NO_DATA	FAILED
	$T_{abandon}$ timeout waiting for MutlipartSend or trigger	None	ABANDONED
	RedfishTaskKill	Success	FAILED
	RedfishTaskStatus	TASK_NEEDING_PAYLOAD	NEEDING PAYLOAD
	Any other command	Error	NEEDING PAYLOAD
TRIGGERED	Task execution completed and device can respond within T_{resp_0}	None	EXECUTED
	Otherwise	None	LONG RUNNING
LONG RUNNING	Execution finished	Generate Task completion Event	EXECUTED
	RedfishTaskKill, Task can be aborted	Success	FAILED
	RedfishTaskKill, Task cannot be aborted	ERROR_TASK_INEVITABLE	LONG RUNNING
	RedfishTaskStatus	TASK_LONG_RUNNING	LONG RUNNING
EXECUTED	Task contains response data	None	HAVING RESPONSE
	Task does not contain response data	None	COMPLETED
HAVING RESPONSE	MultipartReceive, more data to return	Success	HAVING RESPONSE
	MultipartReceive, no more data to return	Success	COMPLETED
	$T_{abandon}$ timeout waiting for MutlipartReceive, more data to return	None	ABANDONED
	RedfishTaskKill	ERROR_TASK_EXECUTED	HAVING RESPONSE
	RedfishTaskStatus	TASK_HAVING_RESPONSE	HAVING RESPONSE

	Any other command	Error	HAVING RESPONSE
COMPLETED	RedfishTaskComplete	Success	INACTIVE
	$T_{abandon}$ timeout waiting for RedfishTaskComplete	None	ABANDONED
	RedfishTaskKill	ERROR_TASK_EXECUTED	COMPLETED
	RedfishTaskStatus	TASK_COMPLETED	COMPLETED
	Any other command	Error	COMPLETED
ABANDONED	RedfishTaskComplete	Success	INACTIVE
	$T_{abandon}$ timeout waiting for MC to issue RedfishTaskComplete	None	INACTIVE
	RedfishTaskKill	ERROR_TASK_ABANDONED	ABANDONED
	RedfishTaskStatus	TASK_ABANDONED	ABANDONED
	Any other command	ERROR_TASK_ABANDONED	ABANDONED
FAILED	RedfishTaskComplete	Success	INACTIVE
	$T_{abandon}$ timeout waiting for RedfishTaskComplete	None	ABANDONED
	RedfishTaskKill	ERROR_TASK_FAILED	FAILED
	RedfishTaskStatus	TASK_FAILED	FAILED
	Any other command	ERROR_TASK_FAILED	FAILED



1522

1523 **Figure 4 – Task Lifecycle State Machine (Device Perspective)**

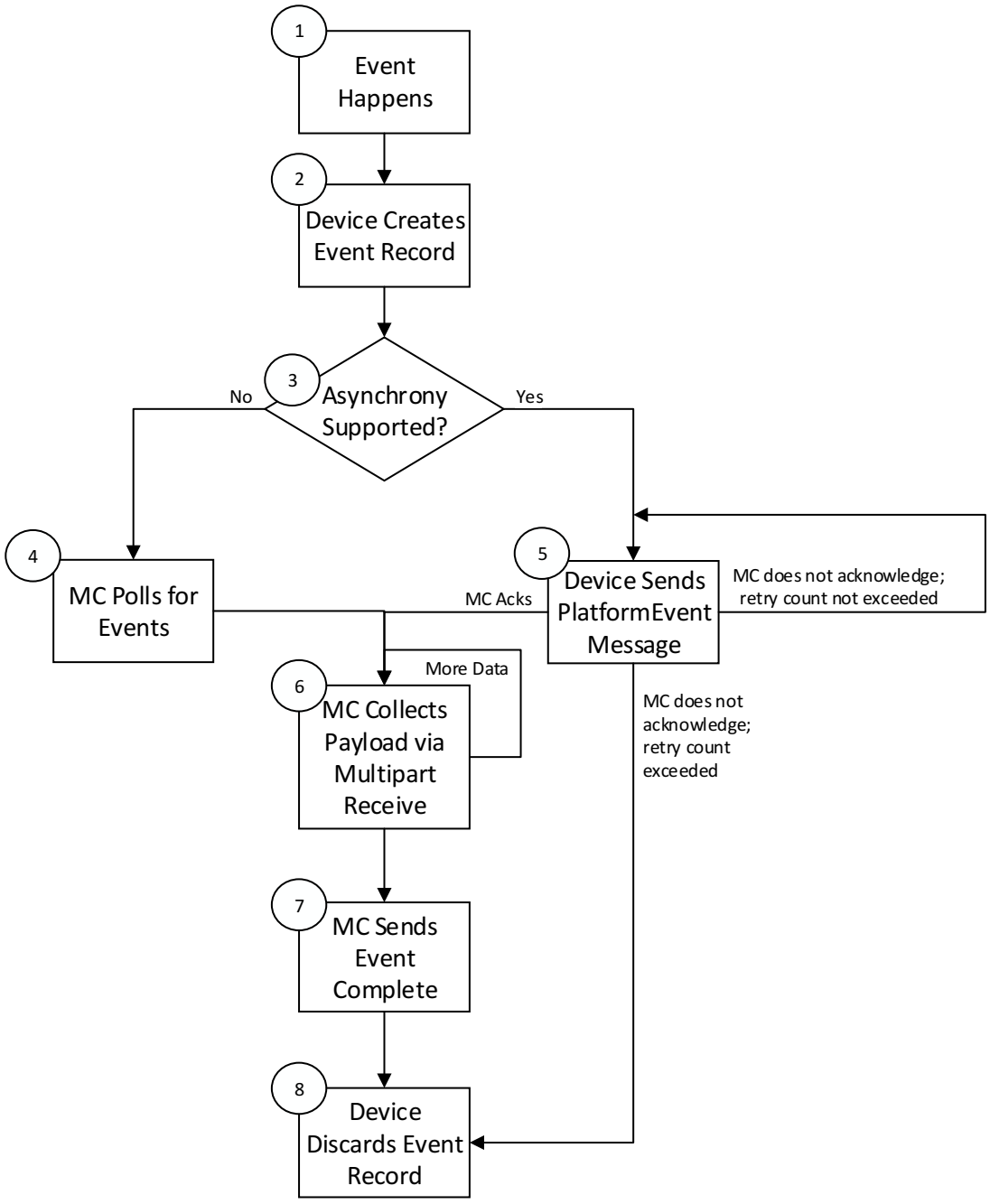
1524 ## 9.2 Event Lifecycle

1525 The below tables describe the operating behavior for MCs and Devices over the lifecycle of Events
1526 depicted visually in Figure 5. This sequence applies to both Task completion Events and schema-based
1527 Events. MC and device implementations of RDE shall comply to the sequences presented here.

1528 **Table 37 – Event Lifecycle Overview**

Current State	Description	Condition	Next State
1	The lifecycle of an Event begins when the Event occurs	Unconditional	2
2	The device creates an Event record	Unconditional	3
3	If the device and the MC both indicated support for asynchrony in the NegotiateRedfishParameters registration sequence	Asynchrony supported	5
		Asynchrony not supported	4

4	The MC polls for Events using the QueryRedfishEvents command and discovers the Event	Unconditional	6
5	The device issues a PlatformEventMessage command to the MC to notify it of the Event	MC acknowledges the Event	6
		MC does not acknowledge the Event and retry count not exceeded	5
		MC does not acknowledge the Event and retry count exceeded	8
6	The MC uses MutlipartReceive to retrieve the Event payload (only needed for Redfish Message Events)	More data to be transfered	6
		Data transfer complete	7
7	The MC sends the RedfishEventComplete command to inform the device that it may discard the Event record	Unconditional	8
8	The device discards its Event record	n/a	-



1529

1530 **Figure 5 – Redfish Event Lifecycle Overview**

1531 10 PLDM Commands for Redfish Device Enablement

1532 This section provides the list of command codes that are used by MCs and providers which implement
1533 PLDM Redfish Device Enablement as defined in this specification. The command codes for the PLDM
1534 messages are given in Table 38. Devices and MCs shall implement all commands where the entry in the
1535 “Command Requirement for Device” or “Command Requirement for MC”, respectively, is listed as
1536 Mandatory. Devices and MCs may optionally implement any commands where the entry in the
1537 “Command Requirement for Device” or “Command Requirement for MC”, respectively, is listed as
1538 Optional.

1539 **Table 38 – PLDM for Redfish Device Enablement Command Codes**

Command	Command Code	Command Requirement for Device	Command Requirement for MC	Command Requestor (Initiator)	Reference
Discovery and Schema Management Commands					
NegotiateRedfishParameters	0x01	Mandatory	Mandatory	MC	See 11.1
GetSchemaDictionary	0x02	Mandatory	Mandatory	MC	See 11.2
GetSchemaURI	0x03	Mandatory	Mandatory	MC	See 11.3
GetSchemaInstanceETag	0x04	Mandatory	Mandatory	MC	See 11.4
Reserved	0x04-0x06				
Event Commands					
QueryRedfishEvents	0x07	Mandatory	Mandatory	MC	See 11.4
RedfishEventComplete	0x08	Mandatory	Mandatory	MC	See 12.2
SetEventReceiver	See DSP0248				
PlatformEventMessage	See DSP0248				
Reserved	0x09-0x0F				
Redfish Operation and Task Commands					
RedfishTaskInit	0x10	Mandatory	Mandatory	MC	See 13.1
RedfishCreate	0x11	Mandatory	Mandatory	MC	See 13.2
RedfishRead	0x12	Mandatory	Mandatory	MC	See 13.3
RedfishWrite	0x13	Mandatory	Mandatory	MC	See 13.4
RedfishAction	0x14	Mandatory	Mandatory	MC	See 13.5
RedfishDelete	0x15	Mandatory	Mandatory	MC	See 13.6
RedfishHead	0x16	Optional	Optional	MC	See 13.7
SupplyCustomRequestParameters	0x17	Mandatory	Mandatory	MC	See 13.8
RetrieveCustomResponseParameters	0x18	Optional *	Optional *	MC	See 13.9
RedfishTaskComplete	0x19	Mandatory	Mandatory	MC	See 13.10
RedfishTaskStatus	0x1A	Optional	Optional	MC	See 13.11
RedfishTaskKill	0x1B	Optional	Optional	MC	See 13.12
RedfishTaskEnumerate	0x1C	Optional	Optional	MC	See 13.13
RedfishTaskFollowup	0x1D	Mandatory	Mandatory	MC	See 13.14
Reserved	0x1E-0x2F				
Multipart Transfer Commands					
MultipartSend	0x30	Mandatory	Mandatory	MC	See 14.1
MultipartReceive	0x31	Mandatory	Mandatory	MC	See 14.2
Reserved	0x32-0xFF				
PLDM for Monitoring and Control Commands					
GetPDRRepositoryInfo	See DSP0248				
GetPDR	See DSP0248				

SetEventReceiver	See DSP0248
PlatformEventMessage	See DSP0248

* The RetrieveCustomResponseParameters should be supported by the MC. However, the MC shall only send this command to the device if the device supports it.

11 PLDM for Redfish Device Enablement – Discovery and Schema Management Commands

This section describes the commands that are used by providers 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 38.

11.1 NegotiateRedfishParameters Command Format

This command enables the MC to negotiate general Redfish parameters with a device.

When the 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.

Table 39 – NegotiateRedfishParameters command format

Type	Request data
uint8	**MCConcurrencySupport** The maximum number of concurrent outstanding long-running Tasks the MC can support for this device. A value of 255 (0xFF) shall be interpreted to indicate that no such limit exists. Upon completion of this command, the device shall not initiate a long-running Task if **MCConcurrencySupport** (or **DeviceConcurrencySupport** whichever is lower) Tasks are already active.
enum8	**MCAsynchronySupport** An indication of whether the MC and platform support asynchronous notifications of Events and completion of Tasks on at least one communication channel with the device. Upon completion of this command, the device shall only issue asynchronous notifications (the RedfishEventsAvailable command, Section 12.2) of event availability if both the **MCAsynchronySupport** and **DeviceAsynchronySupport** are ASYNC_SUPPORTED. Value: { ASYNC_SUPPORTED = 0, ASYNC_NOT_SUPPORTED = 1 }
uint32	**MCMaximumTransferChunkSizeBytes** An indication of the maximum amount of data the MC can support for a single message transfer. For cases of larger blobs of data, the multipart transfer support described in Section 15.1 shall be utilized
Type	**Response data**
enum8	**CompletionCode** value: { PLDM_BASE_CODES }
uint8	**DeviceConcurrencySupport** The maximum number of concurrent outstanding long-running Tasks the device can support. A value of 255 (0xFF) shall be interpreted to indicate that no such limit exists. Regardless of the device's level of support for concurrency, it shall not initiate a task if **MCConcurrencySupport** tasks are already active.

enum8	**DeviceAsynchronySupport** An indication of whether the device supports asynchronous notifications of Events and completion of Tasks. If **MCAsynchronySupport** was set to ASYNC_NOT_SUPPORTED in the request portion of this command, this value shall be ignored; otherwise, this value enables the MC to know whether to expect asychronous notification of events. Value: { ASYNC_SUPPORTED = 0, ASYNC_NOT_SUPPORTED = 1 }
uint16	**DeviceMaximumTransferChunkSizeBytes** The maximum number of bytes that the device can support in a chunk for a single message transfer. If this value is greater than **MCMaximumTransferChunkSizeBytes**, the device shall "throttle down" to using the smaller value. If this value is smaller, the MC shall not attempt a transfer exceeding it.
varstring	**DeviceProviderName** An informal name for the device

11.2 GetSchemaDictionary Command Format

This command enables the MC to retrieve a dictionary (see Section 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 (Section 14.2) to transfer data for the dictionary from the device.

When the 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.

Table 40 – GetSchemaDictionary command format

Type	Request data
uint32	**ResourceID** The ResourceID of for the Redfish Resource PDR from which to retrieve the dictionary
schemaClass	**RequestedSchemaClass** The class of schema being requested
uint8	**RequestedSchemaIndex** The one-based index of the schema being requested. The device shall ignore this field if the RequestedSchemaClass is neither MAJOR_OEM_EXTENSION nor EVENT_OEM_EXTENSION.
Type	**Response data**
enum8	**CompletionCode** value: { PLDM_BASE_CODES, ERROR_BAD_RESOURCE_ID } If an out of range index is supplied for OEM extensions to a schema, the device shall provide completion code ERROR_INVALID_DATA. If an in-range index is supplied but the OEM does not extend the schema, the device shall provide completion code SUCCESS but indicate the lack of an extension through the **TransferHandle** field below.
uint32	**TransferHandle** A data transfer handle that the MC shall use to retrieve the dictionary data via one or more MultipartReceive commands (See Section 14.2). In conjunction with a non-failed **CompletionCode**, the device shall return a valid transfer handle. In the event that the operation fails (e.g. an invalid **ResourceID**), the device shall return a null transfer handle, 0x00000000. If the device does not support a schema of the type requested, it shall return the invalid transfer handle, 0xFFFFFFFF; this can occur when OEM does not extend a schema.

11.3 GetSchemaURI Command Format

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

When the 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.

Table 41 – GetSchemaURI command format

Type	Request data
uint32	**ResourceID** The ResourceID of for the Redfish Resource PDR for which to retrieve the URI
schemaClass	**RequestedSchemaClass** The class of schema being requested
uint8	**RequestedSchemaIndex** The one-based index of the schema being requested. The device shall ignore this field if the RequestedSchemaClass is neither MAJOR_OEM_EXTENSION nor EVENT_OEM_EXTENSION.
Type	**Response data**
enum8	**CompletionCode** value: { PLDM_BASE_CODES, ERROR_BAD_RESOURCE_ID }
varstring	**SchemaURI** URI for the schema

11.4 GetSchemaInstanceETag Command Format

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

When the 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.

Table 42 – GetSchemaInstanceEtag command format

Type	Request data
uint32	**ResourceID** The ResourceID of the Redfish Resource PDR for the instance in from which to get an ETag digest; or 0xFFFFFFFF to get a global digest of all resource-based data within the device
Type	**Response data**
enum8	**CompletionCode** value: { PLDM_BASE_CODES, ERROR_BAD_RESOURCE_ID }
varstring	**ETag** The ETag string data

12 PLDM for Redfish Device Enablement – Event Commands

This section describes the Event commands that are used by providers and MCs that implement Redfish Device Enablement as defined in this specification. The command numbers for the PLDM messages are given in Table 38. NOTE: The SetEventReceiver and PlatformEventMessage commands may be found in DSP0248.

12.1 QueryRedfishEvents Command Format

This command enables the MC to poll whether the device has any Events available for pickup.

When the 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.

Table 43 – QueryRedfishEvents command format

Type	Request data
--	No request data
Type	**Response data**
enum8	**CompletionCode** value: { PLDM_BASE_CODES }
enum8	**EventClass** See PlatformEventMessage in DSP0248; a value of 0xFF indicates that no Events are available (in which case, the size of **EventData** below will be zero).
var	**EventData** See PlatformEventMessage in DSP0248

12.2 RedfishEventComplete Command Format

This command enables the MC to inform a device that it has received an Event and that the device may safely discard any data structures it has constructed for the Event.

When the 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.

Table 44 – QueryRedfishEvents command format

Type	Request data
uint16	**EventID** Identifier for the Event to mark as complete
Type	**Response data**
enum8	**CompletionCode** value: { PLDM_BASE_CODES, ERROR_NO_SUCH_EVENT }

13 PLDM for Redfish Device Enablement – Redfish Task Commands

This section describes the Task commands that are used by providers and MCs that implement Redfish Device Enablement as defined in this specification. The command numbers for the PLDM messages are given in Table 38.

13.1 RedfishTaskInit Command Format

This command enables the MC to initiate a Redfish Task with a device on behalf of a client. After invoking the RedfishTaskInit command, the MC may, upon receipt of a successful completion code, invoke one or more MultipartSend commands (Section 14.1) to transfer data of type bejTuple to the device before invoking the appropriate trigger command. See Section 9 for more details on the Task lifecycle.

When the 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.

1599 **Table 45 – RedfishTaskInit command format**

Type	Request data
uint32	**ResourceID** The resourceID of the Redfish Resource PDR for the data that is the target of this operation
uint16	**TaskID** Sequence number for this Task; must match the one subsequently used for the Task trigger and for any multipart send or receive used to transfer BEJ data for it. Task ID 0x0000 is reserved to indicate that no Task is active.
enum8	**TaskType** The type of operation this Task represents. values: { TASK_HEAD = 0; TASK_READ = 1; TASK_CREATE = 2; TASK_DELETE = 3; TASK_UPDATE = 4; TASK_REPLACE = 5; TASK_ACTION = 6 }
uint32	**SendDataTransferHandle** handle to be used with a MultipartSend command to transfer BEJ formatted data for the operation; shall be 0x00000000 and the contains_send_data bit set to 0 in **TaskFlags** if no data is to be sent for this operation.
uint8	**TaskFlags** Flags associated with this task: bit 0: locator_valid; if 0, the locator in the **TaskLocator** field shall be ignored bit 1: contains_send_data; if 0, the operation does not require data to be sent bit 2..7: reserved for future use
bejLcoator	**TaskLocator** BEJ locator indicating where the new Task is to take place within the schema hierarchy.
Type	**Response data**
enum8	**CompletionCode** value: { PLDM_BASE_CODES, ERROR_BAD_RESOURCE_ID, ERROR_BAD_LOCATOR, ERROR_STRING_FORMAT, ERROR_CANNOT_CREATE_TASK }
uint32	**DeferralTimeframe** The expected length of time before the device will be able to respond to a request to create a Task, in the unit specified in **DeferralUnits** below. The MC shall ignore this field except when the completion code is ERROR_NOT_READY.
enum8	**DeferralUnits** values: { microseconds = 0, seconds = 1, minutes = 2, hours = 3, days = 4 }

1600 ## 13.2 RedfishCreate Command Format

1601 This command enables the MC to trigger a Redfish Create operation to a device on behalf of a client.

1602 When the device receives a request with data formatted per the Request Data section below, it shall
1603 respond with data formatted per the Response Data section if it supports the command.

1604 **Table 46 – RedfishCreate command format**

Type	Request data
uint32	**ResourceID** The resourceID of the Redfish Resource PDR for the instance in which to create data
uint16	**TaskID** Sequence number for this Task; must match the one previously used in the RedfishTaskInit request message before any transfer of BEJ data

Type	Response data
enum8	**CompletionCode** value: { PLDM_BASE_CODES, ERROR_BAD_RESOURCE_ID, ERROR_NO_SUCH_TASK, ERROR_TASK_ABANDONED, ERROR_TASK_FAILED, ERROR_NOT_COLLECTION, ERROR_NO_DATA, ERROR_NOT_ALLOWED, ERROR_UNSUPPORTED_PROPERTY }
bool8	**TaskExecutionFinished** Indicates whether the device has finished executing the task
uint32	**ResultTransferHandle** A data transfer handle that the MC may use to retrieve response data via one or more MultipartReceive commands (See Section 14.2). The device shall return a transfer handle of 0xFFFFFFFF if Task execution has not finished. In the event of a failed operation, or if there is no data to retrieve, the device shall return a null transfer handle, 0x00000000.
bool8	**HaveCustomResponseHeaders** Indicates that the device has custom response headers to return to the client
bitfield8	**PermissionFlags** Bit 0: read access Bit 1: write access Bit 2: create access Bit 3: delete access Bit 4: execute access (for actions) Bits 5-7: reserved for future use This field supports the Allow Response header as described in Section 7.2.4.2.8.
uint16	**NewSequenceNumber** Sequence number for the newly created collection entry; this value is invalid if the Task has not yet completed. This field supports the Location Response header as described in Section 7.2.4.2.6.
varstring	**ETag** String data for ETag This field supports the ETag Response header as described in Section 7.2.4.2.4

13.3 RedfishRead Command Format

This command enables the MC to trigger a Redfish Read operation to a device on behalf of a client.

When the 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.

1609 **Table 47 – RedfishRead command format**

Type	Request data
uint32	**ResourceID** The ResourceID of the Redfish Resource PDR for the instance from which to read data
uint16	**TaskID** Sequence number for this Task; must match the one previously used in the RedfishTaskInit request message before any transfer of BEJ data
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 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.
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 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.
Type	**Response data**
enum8	**CompletionCode** value: { PLDM_BASE_CODES, ERROR_BAD_RESOURCE_ID, ERROR_NO_SUCH_TASK, ERROR_TASK_ABANDONED, ERROR_TASK_FAILED, ERROR_NOT_ALLOWED }
bool8	**TaskExecutionFinished** Indicates whether the device has finished executing the task
uint32	**ResultTransferHandle** A data transfer handle that the MC may use to retrieve response data via one or more MultipartReceive commands (See Section 14.2). The device shall return a transfer handle of 0xFFFFFFFF if Task execution has not finished. In the event of a failed operation, or if there is no data to retrieve, the device shall return a null transfer handle, 0x00000000.
bool8	**HaveCustomResponseHeaders** Indicates that the device has custom response headers to return to the client
bitfield8	**PermissionFlags** Bit 0: read access Bit 1: write access Bit 2: create access Bit 3: delete access Bit 4: execute access (for actions) Bits 5-7: reserved for future use This field supports the Allow Response header as described in Section 7.2.4.2.8.
bool8	**CacheAllowed** True if the data read from this same location is cacheable This field supports the Cache-Control Response header as described in Section 7.2.4.2.7.
varstring	**ETag** String data for ETag This field supports the ETag Response header as described in Section 7.2.4.2.4

13.4 RedfishWrite Command Format

This command enables the MC to trigger a Redfish Update or Redfish Replace operation to a device on behalf of a client.

When the 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.

Table 48 – RedfishWrite command format

Type	Request data
uint32	**ResourceID** The resourceID of the Redfish Resource PDR for the instance in which to create data
uint16	**TaskID** Sequence number for this Task; must match the one previously used in the RedfishTaskInit request message before any transfer of BEJ data
Type	**Response data**
enum8	**CompletionCode** value: { PLDM_BASE_CODES, ERROR_BAD_RESOURCE_ID, ERROR_NOT_ALLOWED, ERROR_NO_SUCH_TASK, ERROR_TASK_ABANDONED, ERROR_TASK_FAILED, ERROR_NO_DATA, ERROR_WRITE_TO_READ_ONLY, ERROR_UNSUPPORTED_PROPERTY }
bool8	**TaskExecutionFinished** Indicates whether the device has finished executing the task
uint32	**ResultTransferHandle** A data transfer handle that the MC may use to retrieve response data via one or more MultipartReceive commands (See Section 14.2). The device shall return a transfer handle of 0xFFFFFFFF if Task execution has not finished. In the event of a failed operation, or if there is no data to retrieve, the device shall return a null transfer handle, 0x00000000.
bool8	**HaveCustomResponseHeaders** Indicates that the device has custom response headers to return to the client
bitfield8	**PermissionFlags** Bit 0: read access Bit 1: write access Bit 2: create access Bit 3: delete access Bit 4: execute access (for actions) Bits 5-7: reserved for future use This field supports the Allow Response header as described in Section 7.2.4.2.8.

13.5 RedfishAction Command Format

This command enables the MC to trigger execution of a Redfish schema-defined action on a device on behalf of a client.

When the 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.

1621 **Table 49 – RedfishAction command format**

Type	Request data
uint32	**ResourceID** The resourceID of the Redfish Resource PDR for the instance in which to perform the action
uint16	**TaskID** Sequence number for this Task; must match the one previously used in the RedfishTaskInit request message before any transfer of BEJ data
Type	**Response data**
enum8	**CompletionCode** value: { PLDM_BASE_CODES, ERROR_BAD_RESOURCE_ID, ERROR_NO_SUCH_TASK, ERROR_TASK_ABANDONED, ERROR_TASK_FAILED, ERROR_NOT_ACTION, ERROR_NO_DATA, ERROR_NOT_ALLOWED }
bool8	**TaskExecutionFinished** Indicates whether the device has finished executing the task
uint32	**ResultTransferHandle** A data transfer handle that the MC may use to retrieve response data via one or more MultipartReceive commands (See Section 14.2). The device shall return a transfer handle of 0xFFFFFFFF if Task execution has not finished. In the event of a failed operation, or if there is no data to retrieve, the device shall return a null transfer handle, 0x00000000.
bool8	**HaveCustomResponseHeaders** Indicates that the device has custom response headers to return to the client
bitfield8	**PermissionFlags** Bit 0: read access Bit 1: write access Bit 2: create access Bit 3: delete access Bit 4: execute access (for actions) Bits 5-7: reserved for future use This field supports the Allow Response header as described in Section 7.2.4.2.8.

1622 ## 13.6 RedfishDelete Command Format

1623 This command enables the MC to send a Redfish Delete operation to a device on behalf of a client.

1624 When the device receives a request with data formatted per the Request Data section below, it shall
1625 respond with data formatted per the Response Data section if it supports the command.

1626 **Table 50 – RedfishDelete command format**

Type	Request data
uint32	**ResourceID** The resourceID of the Redfish Resource PDR for the instance in to delete data
bejLocator	**DeleteLocator** BEJ locator targeting the record to be deleted. NOTE: the pointer must point inside a collection to an element that is to be deleted.
uint16	**TaskID** Sequence number for this Task; must match the one previously used in the RedfishTaskInit request message before any transfer of BEJ data

Type	**Response data**
enum8	**CompletionCode** value: { PLDM_BASE_CODES, ERROR_BAD_RESOURCE_ID, ERROR_NO_SUCH_TASK, ERROR_TASK_ABANDONED, ERROR_TASK_FAILED, ERROR_NOT_COLLECTION, ERROR_NOT_ALLOWED }
bool8	**TaskExecutionFinished** Indicates whether the device has finished executing the task
uint32	**ResultTransferHandle** A data transfer handle that the MC may use to retrieve response data via one or more MultipartReceive commands (See Section 14.2). The device shall return a transfer handle of 0xFFFFFFFF if Task execution has not finished. In the event of a failed operation, or if there is no data to retrieve, the device shall return a null transfer handle, 0x00000000.
bool8	**HaveCustomResponseHeaders** Indicates that the device has custom response headers to return to the client
bitfield8	**PermissionFlags** Bit 0: read access Bit 1: write access Bit 2: create access Bit 3: delete access Bit 4: execute access (for actions) Bits 5-7: reserved for future use This field supports the Allow Response header as described in Section 7.2.4.2.8.

13.7 RedfishHead Command Format

This command enables the MC to send a Redfish Head operation to a device on behalf of a client.

When the 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.

Table 51 – RedfishHead command format

Type	**Request data**
uint32	**ResourceID** The ResourceID of the Redfish Resource PDR for the instance from which to retrieve header data
uint16	**TaskID** Sequence number for this Task; must match the one previously used in the RedfishTaskInit request message before any transfer of BEJ data
Type	**Response data**
enum8	**CompletionCode** value: { PLDM_BASE_CODES, ERROR_BAD_RESOURCE_ID, ERROR_NO_SUCH_TASK, ERROR_TASK_ABANDONED, ERROR_TASK_FAILED, }

bool8	**TaskExecutionFinished** Indicates whether the device has finished executing the Task
uint32	**ResultTransferHandle** A data transfer handle that the MC may use to retrieve response data via one or more MultipartReceive commands (See Section 14.2). The device shall return a transfer handle of 0xFFFFFFFF if Task execution has not finished. In the event of a failed operation, or if there is no data to retrieve, the device shall return a null transfer handle, 0x00000000.
bool8	**HaveCustomResponseHeaders** Indicates that the device has custom response headers to return to the client
bitfield8	**PermissionFlags** Bit 0: read access Bit 1: write access Bit 2: create access Bit 3: delete access Bit 4: execute access (for actions) Bits 5-7: reserved for future use This field supports the Allow Response header as described in Section 7.2.4.2.8.
bool8	**CacheAllowed** True if the data read from this location would be cacheable This field supports the Cache-Control Response header as described in Section 7.2.4.2.7.
varstring	**ETag** String data for ETag; the string text format shall be UTF-8. This field supports the ETag Response header as described in Section 7.2.4.2.4

13.8 SupplyCustomRequestParameters

This command enables the MC to send custom HTTP/HTTPS X- headers and other uncommon request parameters to a device to be applied to an operation. The MC must not use this command to submit any headers for which a standard handling is defined in either this specification or DSP0266.

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 device. When sent, the **SupplyCustomRequestParameters** command shall be invoked after the MC sends the RedfishTaskInit command but before the MC sends the trigger command for the corresponding application.

When the 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.

Table 52 – SupplyCustomRequestHeaders command format

Type	Request data
uint32	**ResourceID** The resourceID of the Redfish Resource PDR for the instance to which custom headers should be supplied
uint16	**TaskID** Sequence number for the Task to which the headers shall be applied; must match the one previously used in the RedfishTaskInit request message before any transfer of BEJ data
enum8	**EtagOperation** values: { ETAG_IGNORE = 0; ETAG_IFMATCH = 1; ETAG_IF_NONE_MATCH = 2 }

uint8	**ETagCount** Number of Etags supplied in this message
varstring	**ETag [0]** String data for first ETag, if ETagCount > 0. This string shall be UTF-8 format
…	Additional ETags
uint8	**HeaderCount** The number of custom headers being supplied in this operation.
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_BAD_RESOURCE_ID, ERROR_NO_SUCH_TASK, ERROR_TASK_ABANDONED, ERROR_TASK_FAILED, ERROR_STRING_FORMAT }

13.9 RetrieveCustomResponseParameters

This command enables the MC to retrieve custom HTTP/HTTPS headers or other uncommon response parameters from a 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 Redfish command indicates that it is needed.

When the 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.

Table 53 – RetrieveCustomResponseParameters command format

Type	**Request data**
uint32	**ResourceID** The resourceID of the Redfish Resource PDR for the instance from which custom headers should be reported
uint16	**TaskID** Sequence number for the Task from which custom headers shall be reported; must match the one previously used in the RedfishTaskInit request message before any transfer of BEJ data
Type	**Response data**
enum8	**CompletionCode** value: { PLDM_BASE_CODES, ERROR_BAD_RESOURCE_ID, ERROR_NO_SUCH_TASK, ERROR_TASK_ABANDONED, ERROR_TASK_FAILED, ERROR_TASK_NOT_FINISHED }
uint8	**HeaderCount** The number of custom headers being supplied in this operation. These headers support the HTTP/HTTPS response with custom headers as described in Section 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
…	**…**

13.10 RedfishTaskComplete

This command enables the MC to inform a device that it considers a Task to be complete. The device in turn may discard any internal records for the Task.

When the 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.

Table 54 – RedfishTaskComplete command format

Type	Request data
uint32	**ResourceID** The resourceID of the Redfish Resource PDR to which the Task's operation was targeted
uint16	**TaskID** Sequence number for the Task; must match the one previously used in the RedfishTaskInit request message before any transfer of BEJ data
Type	**Response data**
enum8	**CompletionCode** value: { PLDM_BASE_CODES, ERROR_BAD_RESOURCE_ID, ERROR_NO_SUCH_TASK, ERROR_TASK_NOT_FINISHED }

13.11 RedfishTaskStatus

This command enables the MC to ask a device for status on a Task.

When the 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.

Table 55 – RedfishTaskStatus command format

Type	Request data
uint32	**ResourceID** The resourceID of the Redfish Resource PDR to which the Task's operation was targeted
uint16	**TaskID** Sequence number for the Task; must match the one previously used in the RedfishTaskInit request message before any transfer of BEJ data
Type	**Response data**
enum8	**CompletionCode** value: { PLDM_BASE_CODES PLDM_BASE_CODES, ERROR_BAD_RESOURCE_ID } The device shall not respond with any of the following codes as these statuses shall be reported in the **TaskStatus** field below: ERROR_NO_SUCH_TASK, ERROR_TASK_ABANDONED, ERROR_TASK_FAILED
enum8	**TaskStatus** values: { 0 = TASK_UNKNOWN; 1 = TASK_RUNNING; 2 = TASK_FINISHED; 3 = TASK_FAILED, 4 = TASK_ABANDONED }

uint8	**TaskCompletionPercentage** 0..100: percentage complete; 101-253: reserved for future use; 254:unable to estimate (but a valid Task) 255: invalid Task
uint32	**CompletionTimeSeconds** An estimate of the number of seconds remaining before the Task is completed, or 0xFFFF FFFF if such an estimate cannot be provided.

13.12 RedfishTaskKill

This command enables the MC to request that a device terminate a Task. The device should kill the Task if it is able to do so; however, the MC must be aware that not all Tasks may be terminated.

When the 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.

Table 56 – RedfishTaskKill command format

Type	Request data
uint32	**ResourceID** The resourceID of the Redfish Resource PDR to which the Task's operation was targeted
uint16	**TaskID** Sequence number for the Task; must match the one previously used in the RedfishTaskInit request message before any transfer of BEJ data
Type	**Response data**
enum8	**CompletionCode** value: { PLDM_BASE_CODES, ERROR_BAD_RESOURCE_ID, ERROR_NO_SUCH_TASK, ERROR_TASK_ABANDONED, ERROR_TASK_FAILED, ERROR_TASK_INEVITABLE, ERROR_TASK_EXECUTED }

13.13 RedfishTaskEnumerate

This command enables the MC to request that a device enumerate all Tasks that are currently active. It is expected that the MC will typically use this command during its initialization to discover any long-running Tasks that were active through shutdown.

When the 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.

Table 57 – RedfishTaskEnumerate command format

Type	Request data
n/a	This request contains no parameters
Type	**Response data**
enum8	**CompletionCode** value: { PLDM_BASE_CODES }
uint8	**TaskCount** The number of active Tasks described in the remainder of this message
uint32	**ResourceID [0]** The resourceID of the Redfish Resource PDR to which the Task's operation was targeted

uint16	**TaskID [0]** Sequence number assigned for the Task when the MC initialized the Task via the RedfishTaskInit command
enum8	**TaskType [0]** The type of operation the Task represents values: { TASK_HEAD = 0; TASK_READ = 1; TASK_CREATE = 2; TASK_DELETE = 3; TASK_UPDATE = 4; TASK_REPLACE = 5; TASK_ACTION = 6 }
…	**…**

13.14 RedfishTaskFollowup Command Format

This command enables the MC to request an update on the status of a Task that has become a long-running Task (i.e. could not be completed within the timeframe required for a response to the trigger request message).

When the 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.

Table 58 – RedfishTaskFollowup command format

Type	Request data
uint32	**ResourceID** The ResourceID of the Redfish Resource PDR for the instance from which to follow up on tasks
uint32	**TaskID** The sequence number for the Task; must match the one previously used in the RedfishTaskInit request message before any transfer of BEJ data
Type	**Response data**
enum8	**CompletionCode** value: { PLDM_BASE_CODES, ERROR_BAD_RESOURCE_ID, ERROR_NO_SUCH_TASK, ERROR_TASK_ABANDONED, ERROR_TASK_FAILED }
uint8	**CompletionPercentage** An estimation of the percentage of the task that has been completed (0..100), or 255 (0xFF) if an estimate cannot be provided. (Funneling this back to a client could ultimately allow display of a progress bar for long-running tasks.)
uint32	**CompletionTimeSeconds** An estimate of the number of seconds remaining before the task is completed, or 0xFFFF FFFF if such an estimate cannot be provided.
enum8	**TaskType** The type of operation this handle refers to. Values: { TASK_CREATE = 0, TASK_READ = 1, TASK_WRITE = 2, TASK_ACTION = 3, TASK_DELETE=4 }
variable	**TaskResponse** The full response packet for the Task, updated to its current status.

14 PLDM for Redfish Device Enablement – Utility Commands

14.1 MultipartSend Command Format

This command enables a participant (in this specification, the MC) to send a large blob of data to another (in this specification, the device).

When the 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.

Table 59 – MultipartSend command format

Type	Request data
uint32	**DataTransferHandle** A handle to uniquely identify the chunk of data to be retrieved
uint16	**TaskID** A handle to line up the transfer to the operation on behalf of which the transfer is occurring. 0x0000 if this transfer is not part of a Task
enum8	**TransferFlag** value: { START = 0, MIDDLE = 1, END = 2, START_AND_END = 3 }
uint32	**NextDataTransferHandle** The handle for the next chunk of data from this blob; zero (0x00000000) if no further data
uint32	**DataLengthBytes** The amount of data being sent in this chunk in bytes. This value shall not exceed the negotiated maximum transfer chunk size (Section 11.1).
bytestream	**Data** The current chunk of data
uint32	**DataIntegrityChecksum** 32-bit CRC for the entire blob of data (all parts concatenated together). May be set to zero (0x00000000) for non-final chunks (TransferFlag ≠ END or START_AND_END) in the transfer. The recipient shall ignore this field except from the final transfer. 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_NO_SUCH_TASK, ERROR_TASK_ABANDONED, ERROR_TASK_FAILED, ERROR_BAD_XFER_HANDLE, ERROR_BAD_CHECKSUM }
enum8	**TransferOperation** value: { XFER_FIRST_PART = 0, XFER_NEXT_PART = 1, XFER_ABORT = 2, XFER_COMPLETE = 3 }

14.2 MultipartReceive Command Format

This command enables a participant (in this specification, the MC) to receive a large blob of data from another (in this specification, the device). In the event of a data checksum error, the MC may reissue the first MultipartReceive command with the initial data transfer handle; the device shall recognize this to mean that the transfer failed and respond as if this were the first transfer attempt.

When the 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.

Table 60 – MultipartReceive command format

Type	Request data
uint32	**DataTransferHandle** A handle to uniquely identify the chunk of data to be retrieved
uint16	**TaskID** A handle to line up the transfer to the Task on behalf of which the transfer is occuring. 0x0000 if this transfer is not part of a Task
enum8	**TransferOperation** value: { XFER_FIRST_PART = 0, XFER_NEXT_PART = 1, XFER_ABORT = 2 }
Type	**Response data**
enum8	**CompletionCode** value: { PLDM_BASE_CODES, ERROR_NO_SUCH_TASK, ERROR_TASK_ABANDONED, ERROR_TASK_FAILED, ERROR_BAD_XFER_HANDLE }
enum8	**TransferFlag** value: { START = 0, MIDDLE = 1, END = 2, START_AND_END = 3 }
uint32	**DataLengthBytes** The amount of data being sent in this chunk in bytes. This value shall not exceed the negotiated maximum transfer chunk size (Section 11.1).
uint32	**NextDataTransferHandle** The handle for the next chunk of data from this blob; zero (0x00000000) if no further data
bytestream	**Data** The current chunk of data
uint32	**DataIntegrityChecksum** 32-bit CRC for the entire blob of data (all parts concatenated together). May be set to zero (0x00000000) for non-final chunks (TransferFlag ≠ END or START_AND_END) in the transfer. The recipient shall ignore this value except from the final transfer. 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.

15 Additional Information

15.1 Multipart Transfers

The GetSchemaDictionary, RedfishCreate, RedfishRead, RedfishUpdate, and RedfishAction commands defined in Sections 11 and 13 support multipart transfers via the MultipartSend and MultipartReceive commands defined in Section 13.8. 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. Following are some requirements for using TransferOperationFlag, TransferFlag, and DataTransferHandle for a given data transfer:

- To prepare a large data transfer for use in one of the GetSchemaDictionary, RedfishCreate, RedfishRead, RedfishUpdate, and RedfishAction commands, a DataTransferHandle shall be sent by the MC in the request message (RedfishCreate, RedfishUpdate, RedfishAction commands) or by the device in the response message (GetSchemaDictionary, RedfishRead commands).

- To initiate a data transfer with either a MultipartSent or MultipartReceive command, the TransferOperationFlag shall be set to XFER_FIRST_PART in the request message.

- For transferring a part other than the first part of data, the TransferOperationFlag 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 or in the request for the previous MultipartSend command for this data transfer.

- The TransferFlag specified in the response of a MultipartReceive command or 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

- For a MultipartReceive, the requester shall consider a data transfer complete when the TransferFlag in the response is set to End or StartAndEnd. 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.

The following example shows 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 device as part of a Redfish Update operation. Following the RedfishTaskInit command sequence, the MC effects the transfer via a series of MultipartSend commands. Figure 6 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 7 shows the flow of the data transfer.

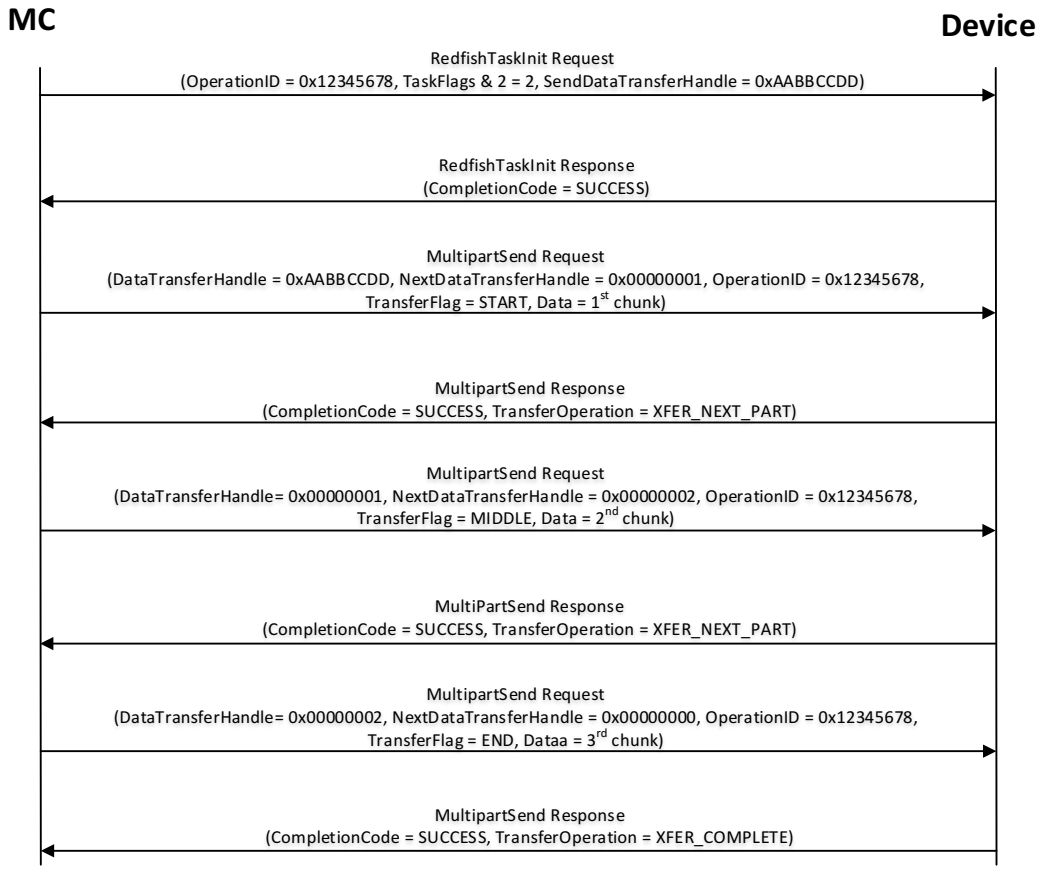


Figure 6 – MultipartSend Example

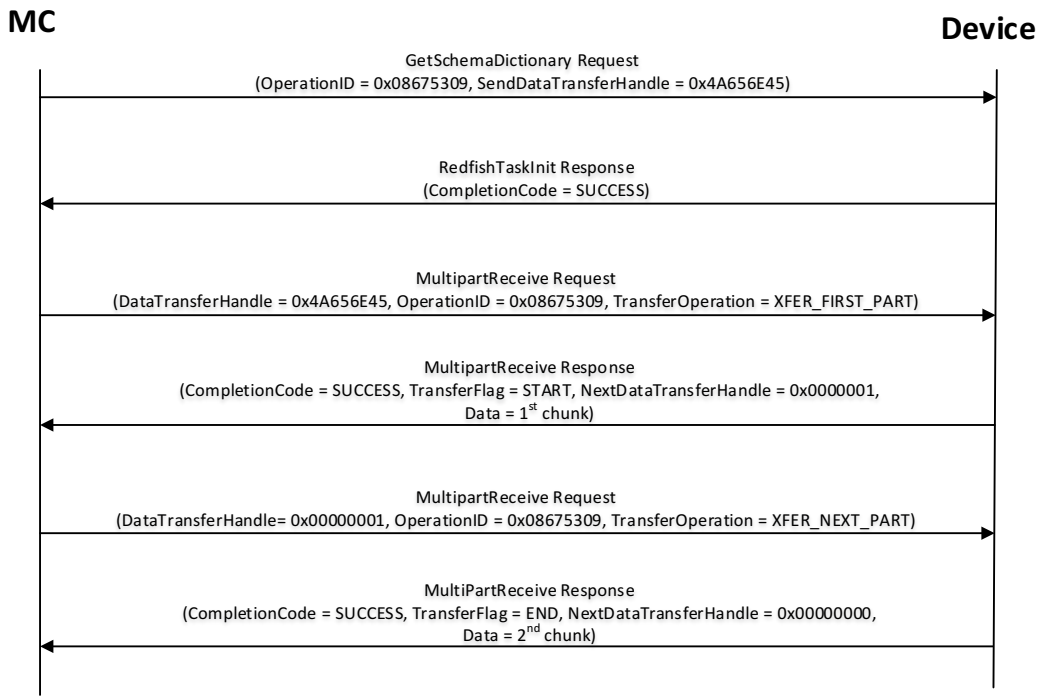


Figure 7 – MultipartReceive Example

15.2 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 38.

15.3 [Dev] Considerations for Provider Device Manufacturers

Several implementation notes apply to manufacturers of provider devices.

15.3.1 Schema Update

If one or more schemas for a device are updated, such as via a firmware update, the device may communicate this to the MC signaling an update to its PDRs via modifying the update timestamp in its response to the PLDM Monitoring and Control GetRepositoryInfo command (DSP0248).

15.3.2 Handling Multiple MCs

A single device may find that it is attached to multiple MCs. This can introduce several complications.

15.3.2.1 Concurrency

In order to provide a consistent view of device data to MCs, it is recommended that devices strictly serialize Tasks that write or modify data (such as creates, deletes, some actions). In the event that a device chooses to support multiple concurrent Tasks that affect data, the device shall ensure that the behavior is serializable; that is, the data must always be viewable as if each Task was executed start to finish. The order of apparent execution of Tasks may be chosen to be whatever the device chooses.

In many cases, a device may choose to only allow data to be affected by one MC and restrict others to read-only access. Further discussion of concurrency strategies for devices is beyond the scope of this specification.

15.3.2.2 Isolation of Tasks

Every Task shall be uniquely associated with a single MC. Commands such as RedfishTaskStatus or RedfishTaskKill shall not be able to view or modify a Task when issued from the “wrong” MC. RedfishTaskEnumerate shall not enable an MC to view the Tasks issued by another MC. Note that a single MC connected through multiple media substrates shall not count as multiple MCs for this purpose; such an MC may issue, view, and manage Tasks through the connection media of its choice.

15.3.2.3 Recognition of MCs

When a connection to an MC is lost and then reestablished to the same MC, the device shall treat the restored MC as being the same MC as the pre-interruption MC for purposes of Task access. The technique by which the device may recognize that it is the same MC that has reconnected to the device will typically vary by connection medium. For example, the UUID of an MCTP-connected device may be used to identify it.

15.4 Alignment of Redfish Resource PDRs

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

NOTE: For collections, the device shall offer one PDR for the collection as a whole and one PDR for each entry 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.

15.5 [MC] Considerations for MC Manufacturers

Several implementation notes apply to manufacturers of management controllers.

15.5.1 Rediscovery of Devices

It is entirely possible that circumstances may arise requiring rediscovery of the device. The MC shall monitor changes to the device PDRs in order to detect when rediscovery is necessary. The MC shall be aware that upon rediscovery it is entirely possible that the schemas a particular device supports may change, such as if the device received a firmware upgrade.

15.5.2 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 devices via the GetSchemaDictionary command (Section 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.

15.5.3 Schemas for Sibling Instances

MCs must not assume that sibling instances of Redfish Resource PDRs in a hierarchy use the same 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 same dictionary.

15.5.4 HTTP/HTTPS POST Operations

As specified in DSP0266, an HTTP/HTTPS POST operation in Redfish can represent either a Create operation or an Action. To distinguish between these cases, the MC may examine the URI target of 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_NOT_COLLECTION error code from the 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.) Implementors should be aware that such delays could cause a client-side timout.

15.5.5 Consistency Checking of Read Operations

Because the collection of data contained within a schema cannot generally be read atomically by devices, issues of consistency arise. In particular, if the device reads some of the data, performs an update, 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 GetSchemaETag command (Section 11.4). 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 N_{read} attempts, the MC cannot obtain a consistent read, it should report an error to the client.

15.5.6 Placement of Devices in the Outward-facing Redfish URI Hierarchy

In the Redfish Resource PDRs that a device presents, there will normally be one or a limited number that reflect EXTERNAL (0x0000) as their ContainingResourceID. These PDRs 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 device’s responsibility to make sure that they are accessible via some chain of Redfish Resource PDRs that ultimately link to EXTERNAL.

When retrieving these Redfish Resource PDRs for device components, the MC should read the MajorSchemaName field to identify the type of 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 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 Section 11.3, download the schema, and inspect the schema, namespace, or other content.)

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

15.6 [Cli] Considerations for Redfish Clients

No changes to behavior are required of Redfish clients in order to interact with BEJ-based 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 a device that operates in JSON over HTTP/HTTPS or BEJ over PLDM.

ANNEX A
(informative)

Change log

Version	Date	Author	Description
0.8.0	2018-02-20		Work in Progress

ANNEX B
(informative)

Tools in Support of PLDM for Redfish Device Enablement

[TBD]

ANNEX C
(temporary)

Materials for PLDM for Redfish Device Enablement that need to go into other specifications

Note: This Annex will go away in the final version of the specification. Its purpose is to collect those materials that will ultimately reside in other specifications.

The list of String formats (Table 20 from PLDM FW Update) needs to be moved to one of the base specs.

Additionally, the following data needs to be added:

- DSP0245 (IDs and Codes)
  - Table 1, PLDM types: add PLDM for Redfish Device Enablement as Type 6
- DSP0249 (State Sets)
  - Table 15, Entity ID codes: add Schema (and Schema reference?) entity types
- DSP0240 (PLDM base)
  - No updates required (transfer limits?)
- DSP0248 (Management and Control)
  - Add Redfish Resource PDR, below (highlighted fields are ones we don't need for this spec – do we need them for any other reason?

Type	Description
–	**CommonHeader** See [**Error! Reference source not found.**].
uint16	**PLDMTerminusHandle** A handle that identifies PDRs that belong to a particular PLDM terminus.
uint16	**EntityType** logical | schema
uint16	**EntityInstanceNumber**
uint16	**ContainerID** The containerID for the containing entity that holds this terminus. See [**Error! Reference source not found.**] for more information.
uint32	**ResourceID** The resourceID for this collection of data
uint32	**ContainingResourceID** value: 0x0001 to 0xFFFF = An opaque number that identifies a particular Redfish Resource PDR in the hierarchy of containment. See 11.1 for more information. special value: 0x0000 = " EXTERNAL ". This value is used to identify the logical root of a device component's management topology.
uint32	**MajorSchemaVersion** In standard PLDM version format
uint16	**MajorSchemaDictionaryLengthBytes** Length of dictionary data for the major schema
uint8	**MajorSchemaNameLength** Length of the name of the major schema, including null terminator

Type	Description
utf8string	**MajorSchemaName** Null-terminated UTF-8 string containing the name of the major schema
uint32	**EventSchemaVersion** Version of the Event (Message) schema used by this device, in standard PLDM version format
uint32	**EventSchemaDictionaryLengthBytes** Length of binary data for Event schema dictionary
uint8	**EventSchemaNameLength** Length of the name of the Event schema, including null terminator
utf8string	**EventSchemaName** Null-terminated UTF-8 string containing the name of the Event schema
uint32	**OEMDataSizeBytes** Length in bytes of the remainder of this PDR, beginning with the **OEMCount** field.
uint16	**OEMCount** Number of OEMs associated with this schema in the device
uint16	**OEMNameLengthBytes [1]** Null-terminated UTF-8 string containing the name of the OEM
utf8string	**OEMName [1]** Name of the OEM
uint32	**OEMSchemaVersion [1]** Version of OEM extension to the major schema in standard PLDM version format; 0xFFFFFFFF if this OEM does not extend the major schema
uint32	**OEMSchemaDictionaryLengthBytes [1]** Length of dictionary data for the OEM extension to the major schema
uint32	**OEMEventSchemaVersion [1]** Version of OEM extension to the Event schema in standard PLDM version format; 0xFFFFFFFF if this OEM does not extend the Event schema
uint32	**OEMEventSchemaDictionaryLengthBytes [1]** Length of dictionary data for the OEM extension to the Event schema
…	**…**

- Add Redfish Entity Association PDR, below

Type	Description
–	**CommonHeader** See [**Error! Reference source not found.**].
Container Entity Identification Information	
uint16	**ContainingResourceID** value: 0x0001 to 0xFFFF = An opaque number that identifies a particular container entity in the hierarchy of containment. See 11.1 for more information. special value: 0x0000 = "EXTERNAL". This value is used to identify the topmost containing entity for a device component in PLDM Entity Association containment hierarchies.
Contained Entity Identification Information	

Type	Description
uint8	**ContainedEntityCount** The number of contained entities (1 to N) listed in this particular PDR. This may not be the total number of contained entities because multiple containment association PDRs may exist for the same container entity. See 11.3 for more information.
uint16	**ContainedEntityResourceID[1]**
	…
uint16	**ContainedEntityResourceID[N]**

- Add Redfish Action PDR, below

Type	Description
–	**CommonHeader** See [**Error! Reference source not found.**].
Container Entity Identification Information	
uint16	**RelatedResourceID** value: 0x0001 to 0xFFFF = An opaque number that identifies a particular container entity in the hierarchy of containment. See 11.1 for more information. special value: 0x0000 = "EXTERNAL". This value may be used to define actions for the external to any schema.
Action Information	
schemaClass	**ActionSchemaClass** The type of schema for which this PDR names actions. Any value other than MAJOR and MAJOR_OEM_EXTENSION shall be considered erroneous
uint8	**ActionSchemaIndex** The specific schema for which this PDR names actions
uint8	**ActionCount** The number of Redfish Actions (0 to N – 1) associated with the host Redfish Resource PDR.
uint8	**ActionNameLengthBytes [1]** Including null terminator
utf8string	**ActionName [1]** The name of action, null terminated
	…
uint8	**ActionNameLengthBytes [N]** Including null terminator
utf8string	**ActionName [N]** The name of action, null terminated

- Add Event classes redfishTaskExecutedEvent and redfishMessageEvent to the **eventClass** enumeration in the PlatformEventMessage request message (Table 14)
- Add redfishTaskExecutedEvent class eventData format, below:

Type	Request data
uint16	**ResourceID** The ResourceID is the value that is used in PDRs and PLDM for Redfish Device Enablement commands to identify and access a particular collection of schema-based Redfish data

Type	Request data
uint16	**EventID** Identifier that the MC may use to reference this Event in a follow-up call to RedfishEventComplete
uint16	**TaskID** Identifier for a long-running Task that the device has finished executing

- Add redfishMessageEvent class eventData format, below:

Type	Request data
uint16	**HostResourceID** The ResourceID is the value that is used in PDRs and PLDM for Redfish Device Enablement commands to identify and access a particular collection of schema-based Redfish data
uint8	**EventSchemaIndex** The particular Event schema or Event schema extension that the Event refers to. A value of 0 refers to the Standard Redfish Event schema. Any other value refers to the 1-based list of OEM extensions to the standard Event schema
uint16	**EventID** Identifier that the MC may use to reference this Event in a follow-up call to RedfishEventComplete
uint16	**EventDataTransferHandle** A data transfer handle that the MC may use to retrieve Event data via one or more MultipartReceive commands (See DSP0218). The data blob received shall be of type bejTuple, encoded with the standard Redfish Event schema dictionary

ANNEX D
(temporary)

Known Issues

Note: This Annex will go away in the final version of the specification. Its purpose is to enumerate known open issues in the current draft of the specification.

The following issues are known to be outstanding in the current version of the spec:

- The definition of “schema” is less formal than it could be
- There are probably several more Device and MC considerations to be added
- The Redfish security model is not addressed in this specification
- Identification of tools in support of Redfish for Device Enablement (Annex B) remains outstanding