Document Identifier: DSP0240

Date: 2021-02-11

Version: 1.1.0

Platform Level Data Model (PLDM) Base Specification

Supersedes: 1.0.0

Document Class: Normative

Document Status: Published

Document Language: en-US

CONTENTS

Figures

Tables

Foreword

The *Platform Level Data Model (PLDM) Base Specification* (DSP0240) was prepared by the Platform Management Components Intercommunications (PMCI) Work Group.

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

Acknowledgments

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

- Hemal Shah – Broadcom, Inc.
- Bill Scherer – Hewlett Packard Enterprise
- Alan Berenbaum – SMSC
- Patrick Caporale – Lenovo
- Philip Chidester – Flex
- Andy Currid – NVIDIA Corporation
- Hoan Do – Broadcom Inc.
- Dov Goldstein – Intel Corp.
- Christoph Graham – Hewlett Packard Inc.
- Yuval Itkin – NVIDIA Corporation
- Janiszewski, Jacek – Intel Corporation
- Ira Kalman – Intel Corporation
- Edward Klodnicki – Lenovo
- Deepak Kodihalli – IBM
- Eliel Louzoun – Intel Corp.
- Rob Mapes – Marvell
- Balaji Natrajan – Microchip Technology Inc.
- Edward Newman – Hewlett Packard Enterprise
- Stephen Phong – Advanced Micro Devices
- Patrick Schoeller – Hewlett Packard Enterprise
- Tom Slaight – Intel Corporation
- Bob Stevens – Dell Inc.
- Abeye Teshome – Dell Inc.
- Richard Thomaiyar – Intel Corp.
- Paul Vancil – Advanced Micro Devices
- Supreeth Venkatesh – Advanced Micro Devices

Introduction

This document describes base protocol elements of the Platform Level Data Model (PLDM) for the purpose of supporting platform-level data models and platform functions in a platform management subsystem. PLDM is designed to be an effective interface and data model that provides efficient access to low-level platform inventory, monitoring, control, event, and data/parameters transfer functions. For example, temperature, voltage, or fan sensors can have a PLDM representation that can be used to monitor and control the platform using a set of PLDM messages. PLDM defines data representations and commands that abstract the platform management hardware.

Platform Level Data Model (PLDM) Base Specification

1 Scope

This specification describes base protocol elements of the Platform Level Data Model (PLDM) for the purpose of supporting platform-level data models and platform functions in a platform management subsystem. PLDM defines data representations and commands that abstract the platform management hardware.

This specification defines the following elements:

- the base Platform Level Data Model (PLDM) for various platform functions
- a common PLDM message format to support platform functions using PLDM

The PLDM message common fields support the identification of payload type, message, PLDM type, and PLDM command/completion codes.

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.

DMTF DSP0004, *CIM Infrastructure Specification 2.5,*
http://www.dmtf.org/standards/published_documents/DSP0004_2.5.0.pdf

DMTF DSP0223, *Generic Operations 1.0*,
http://www.dmtf.org/standards/published_documents/DSP0223_1.0.pdf

DMTF DSP0241, *Platform Level Data Model (PLDM) over MCTP Binding Specification,*
http://www.dmtf.org/standards/published_documents/DSP0241_1.0.0.pdf

DMTF DSP0245, *Platform Level Data Model (PLDM) IDs and Codes,*
http://www.dmtf.org/standards/published_documents/DSP0245_1.0.0.pdf

DMTF DSP1001, *Management Profile Specification Usage Guide 1.1*,
http://www.dmtf.org/standards/published_documents/DSP1001_1.1.pdf

DMTF DSP4004, *DMTF Release Process 2.3*,
https://www.dmtf.org/sites/default/files/standards/documents/DSP4004_2.3.0.pdf

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

ANSI/IEEE Standard 754, *Standard for Binary Floating Point Arithmetic*,
http://ieeexplore.ieee.org/xpl/tocresult.jsp?isNumber=1316

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

IETF RFC3629, *UTF-8, a transformation format of ISO 10646*, November 2003,
http://www.ietf.org/rfc/rfc3629.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

Hewlett-Packard, Intel, Microsoft, Phoenix, and Toshiba, *Advanced Configuration and Power Interface Specification 3.0*, ACPI, September 2, 2004, http://www.acpi.info/DOWNLOADS/ACPIspec30.zip

Intel, Hewlett-Packard, NEC, and Dell, *Intelligent Platform Management Interface Specification: Second Generation 2.0*, IPMI, 2004, ftp://download.intel.com/design/servers/ipmi/IPMIv2_0rev1_0markup.pdf

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

OMG, *Unified Modeling Language (UML) from the Open Management Group (OMG),* http://www.uml.org/

3 Terms and definitions

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

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

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

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

The terms defined in DSP0004, DSP0223, and DSP1001 apply to this document. The following additional terms are used in this document.

For the purposes of this document, the following terms and definitions apply.

3.1
baseboard management controller
BMC

A term coined by the IPMI specifications for the main management controller in an IPMI-based platform management subsystem. Also sometimes used as a generic name for a motherboard-resident management controller that provides motherboard-specific hardware monitoring and control functions for the platform management subsystem.

3.2
binary-coded decimal
BCD

Indicates a particular binary encoding for decimal numbers where each four bits (*nibble*) in a binary number is used to represent a single decimal digit, and with the least significant four bits of the binary number corresponding to the least significant decimal digit

The binary values `0000b` through `1001b` represent decimal values 0 through 9, respectively. For example, with BCD encoding a byte can represent a two-digit decimal number where the most significant nibble (bits 7:4) of the byte contains the encoding for the most significant decimal digit and the least

significant nibble (bits 3:0) contains the encoding for the least significant decimal digit (for example, 0010_1001b (0x29) in BCD encoding corresponds to the decimal number 29).

3.3
bridge

Generically, the circuitry and logic that connect one computer bus or interconnect to another, allowing an agent on one to access the other

3.4
bus

A physical addressing domain shared between one or more platform components that share a common physical layer address space

3.5
byte

An 8-bit quantity. Also referred to as an *octet*.

NOTE PLDM specifications shall use the term *byte*, not *octet*.

3.6
Common Information Model
CIM

The schema of the overall managed environment

It is divided into a *core, model*, *common model,* and *extended schemas*. For more information, see DSP0004.

3.7
endpoint

See *MCTP endpoint*

3.8
endpoint ID
EID

See *MCTP endpoint*

3.9
Globally Unique Identifier
GUID

See *UUID*

3.10
Initialization Agent function

A software or firmware component that configures PLDM and assigns Terminus IDs; typically a management controller

3.11
Inter-Integrated Circuit
I^2C

A multiple-master, two-wire, serial bus originally developed by Philips Semiconductor

3.12
idempotent command

A command that has the same effect for repeated applications of the same command

3.13

intelligent management device
IMD

A management device that is typically implemented using a microcontroller and accessed through a messaging protocol

Management parameter access provided by an IMD is typically accomplished using an abstracted interface and data model rather than through direct "register-level" access.

3.14

Intelligent Platform Management Interface
IPMI

A set of specifications defining interfaces and protocols originally developed for server platform management by the IPMI Promoters Group: Intel, Dell, HP, and NEC

3.15

Manageability Access Point
MAP

A collection of services of a system that provides management in accordance to CIM profiles and management protocol specifications published under the DMTF

3.16

managed entity

The physical or logical entity that is being managed through management parameters. Examples of *physical* entities include fans, processors, power supplies, circuit cards, chassis, and so on. Examples of *logical* entities include virtual processors, cooling domains, system security states, and so on.

3.17

Management Component Transport Protocol
MCTP

A media-independent transport protocol that was designed for intercommunication of low-level management messages within a platform management subsystem

3.18

management controller

A microcontroller or processor that aggregates management parameters from one or more management devices and makes access to those parameters available to local or remote software, or to other management controllers, through one or more management data models

Management controllers may also interpret and process management-related data, and initiate management-related actions on management devices. The microcontroller or processor that serves as a management controller can also incorporate the functions of a management device.

3.19

management device

Any physical device that provides protocol terminus for accessing one or more management parameters

A management device responds to management requests, but it does not initiate or aggregate management operations except in conjunction with a management controller (that is, it is a *satellite* device that is subsidiary to one or more management controllers). An example of a simple management device would be a temperature sensor chip. Another example would be a management controller that has I/O pins or built-in analog-to-digital converters that monitor state and voltages for a managed entity.

3.20
management parameter

A particular datum representing a characteristic, capability, status, or control point associated with a managed entity
Example management parameters include temperature, speed, volts, on/off, link state, uncorrectable error count, device power state, and so on.

3.21
MCTP bridge

An MCTP endpoint that can route MCTP messages (that are not destined for itself) that it receives on one interconnect to another without interpreting them

The ingress and egress media at the bridge may be either homogeneous or heterogeneous. Also referred to in this document as a "bridge".

3.22
MCTP bus owner

The entity that is responsible for MCTP EID assignment or translation on the buses of which it is a master

The MCTP bus owner may also be responsible for physical address assignment. For example, for SMBus bus segments, the MCTP bus owner is also the ARP master. This means the bus owner assigns dynamic SMBus addresses to devices that require it.

3.23
MCTP endpoint

A terminus or origin of an MCTP packet or message

The MCTP endpoint is identified by a value called the MCTP endpoint ID, or EID.

3.24
message

See *PLDM message*

3.25
message body

The portion of a PLDM message that carries the PLDM Type-specific data associated with the message

3.26
message originator

The original transmitter (source) of a message targeted to a particular PLDM terminus

3.27
most significant byte
MSB

The highest order byte in a number consisting of multiple bytes

3.28
non-idempotent command

A command that is not an idempotent command

3.29
nibble

The computer term for a four-bit aggregation, or half of a byte

3.30

payload

The information-bearing fields of a message

These fields are separate from the fields and elements (such as address fields, framing bits, checksums, and so on) that are used to transport the message from one point to another. In some instances, a given field may be both a payload field and a transport field.

3.31

physical transport binding

Refers to specifications that define how a base messaging protocol is implemented on a particular physical transport type and medium, such as SMBus/I^2C, PCI Express™ Vendor Defined Messaging, and so on

3.32

Platform Level Data Model
PLDM

An internal-facing low-level data model that is designed to be an effective data/control source for mapping under the Common Information Model (CIM)

PLDM defines data structures and commands that abstract platform management subsystem components. PLDM supports a Type field to distinguish various types of messages and group them together based on the functions.

3.33

PLDM command

A command defined under the PLDM Type that is used for PLDM communications (for example, commands to control BIOS configuration and attributes transfer, perform SMBIOS data transfer, and monitor and control sensors)

3.34

PLDM message

A unit of communication based on the PLDM Type that is used for PLDM communications

3.35

PLDM message payload

A portion of the message body of a PLDM message

This portion of the message is separate from those fields and elements that are used to identify the payload type, message, PLDM Type, and PLDM command/completion codes.

3.36

PLDM request

Same as *PLDM command*. See 3.33.

3.37

PLDM request message

A message that is sent to a PLDM terminus to request a specific PLDM operation

A PLDM request message is acknowledged with a corresponding response message.

3.38

PLDM response

A response to a specific PLDM request

388 **3.39**
389 **PLDM response message**
390 A message that is sent in response to a specific PLDM request message

391 This message includes a "Completion Code" field that indicates whether the response completed
392 normally.

393 **3.40**
394 **PLDM subsystem**

395 The collection of devices which are enumerated by the same PLDM initialization agent

396 **3.41**
397 **PLDM terminus**
398 Identifies a set of resources within the recipient endpoint that is handling a particular PLDM message

399 **3.42**
400 **Platform Management Component Intercommunications**
401 **PMCI**

402 The name of a working group under the Distributed Management Task Force that is chartered to define
403 standardized communication protocols, low-level data models, and transport definitions that support
404 communications with and between management controllers and management devices that form a
405 platform management subsystem within a managed computer system

406 **3.43**
407 **point-to-point**

408 Refers to the case where only two physical communication devices are interconnected through a physical
409 communication medium

410 The devices may be in a master and slave relationship, or the devices could be peers.

411 **3.44**
412 **Universally Unique Identifier**
413 **UUID**

414 An identifier originally standardized by the Open Software Foundation (OSF) as part of the Distributed
415 Computing Environment (DCE). UUIDs are created using a set of algorithms that enables them to be
416 independently generated by different parties without requiring that the parties coordinate to ensure that
417 generated IDs do not overlap

418 In this specification, RFC4122 is used as the base specification for describing the format and generation
419 of UUIDs. This identifier is also sometimes referred to as a globally unique identifier (GUID).

420 4 Symbols and abbreviated terms

421 The abbreviations defined in DSP0004, DSP0223, and DSP1001 apply to this document. The following
422 additional abbreviations are used in this document.

423 **4.1**
424 **ACPI**
425 Advanced Configuration and Power Interface

426 **4.2**
427 **ARP**
428 Address Resolution Protocol

4.3
CIM
Common Information Model

4.4
DCE
Distributed Computing Environment

4.5
GUID
Globally Unique Identifier

4.6
IMD
Intelligent management device

4.7
IPMI
Intelligent Platform Management Interface

4.8
ISO/IEC
International Organization for Standardization/International Engineering Consortium

4.9
MC
Management Controller

4.10
MCTP
Management Component Transport Protocol

4.11
MSB
Most significant byte

4.12
OSF
Open Software Foundation

4.13
PLDM
Platform Level Data Model

4.14
PMCI
Platform Management Component Intercommunications

4.15
TID
Terminus ID

4.16
UUID
Universally Unique Identifier

4.17
WBEM
Web-Based Enterprise Management

5 Conventions

The conventions described in the following clauses apply to all of the PLDM specifications.

5.1 Notations

PLDM specifications use the following notations:

Table 1 – PLDM notations

Notation	Interpretation
M:N	In field descriptions, this notation typically represents a range of byte offsets starting from byte M and continuing to and including byte N (M ≤ N). The lowest offset is on the left. The highest offset is on the right.
[4]	Square brackets around a number typically indicate a bit offset. Bit offsets are zero-based values. That is, the least significant bit.
[M:N]	A range of bit offsets where M is greater than or equal to N. The most significant bit is on the left, and the least significant bit is on the right
1b	A lowercase b after a number consisting of 0 s and 1 s indicates that the number is in binary format.
0x12A	The sequence “0x” preceding a number of decimal digits and the letters A..F indicates that the number is in hexadecimal format.

Numeric constants in specifications will generally be presented in decimal; however, two exceptions exist where non-decimal presentations may be used in addition or instead of decimal presentations:

- Numeric constants for fields of size less than one byte should be represented in binary
- Numeric constants that exceed 15 or in sets where at least one value exceeds 15 (such as the Generic PLDM completion codes of clause 7.2) should be presented in hexadecimal

5.2 Reserved and unassigned values

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

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

5.3 Byte ordering

Unless otherwise specified, for all PLDM specifications byte ordering of multibyte numeric fields or multibyte bit fields is "Little Endian" (that is, the lowest byte offset holds the least significant byte, and higher offsets hold the more significant bytes).

5.4 PLDM data types

Table 2 lists the abbreviations and descriptions for common data types that are used in PLDM message fields and data structure definitions.

Table 2 – PLDM data types

Data Type	Interpretation
uint8	Unsigned 8-bit binary integer
sint8	Signed 8-bit binary integer
uint16	Unsigned 16-bit binary integer
sint16	Signed 16-bit binary integer
uint32	Unsigned 32-bit binary integer
sint32	Signed 32-bit binary integer
uint40	Unsigned 40-bit binary integer
sint40	Signed 40-bit binary integer
uint64	Unsigned 64-bit binary integer
sint64	Signed 64-bit binary integer
string	UCS-2 string as defined in ISO/IEC 10646
bool8	Boolean value represented using an unsigned 8-bit binary integer where 0x00 means False, and any non-zero value means True
real32	Four-byte floating-point format, also known as "single precision", where: [31] – S (sign) bit (1 = negative, 0 = positive) [30:23] – exponent as a binary integer (8 bits) [22:0] – mantissa as a binary integer (23 bits) Per ANSI/IEEE Standard 754 convention, the value represented is determined as follows: If Exponent = 255 and Mantissa is nonzero, then Value = NaN ("Not a number"). If Exponent = 255 and Mantissa is zero and S is 1, then Value = -Infinity. If Exponent = 255 and Mantissa is zero and S is 0, then Value = Infinity. If 0<Exponent<255, then Value=(-1)**S * 2 ** (Exponent-127) * (1.Mantissa) where "1.Mantissa" is intended to represent the binary number created by prefixing Mantissa with an implicit leading 1 and a binary point. If Exponent = 0 and Mantissa is nonzero, then Value = (-1)**S * 2 ** (-126) * (0.Mantissa). These are "unnormalized" values. If Exponent = 0 and Mantissa is zero and S is 1, then Value = -0. * If Exponent = 0 and Mantissa is zero and S is 0, then Value = 0.

Data Type	Interpretation
real64	Eight-byte floating-point, also known as "double-precison", where: [63] – S (sign) bit (1 = negative, 0 = positive) [62:52] – exponent as a binary integer (11 bits) [51:0] – mantissa as a binary integer (52 bits) Per IEEE 754 convention, the value represented is determined as follows: If Exponent = 2047 and Mantissa is nonzero, then Value = NaN ("Not a number"). If Exponent = 2047 and Mantissa is zero and S is 1, then Value = -Infinity. If Exponent = 2047 and Mantissa is zero and S is 0, then Value = Infinity. If 0<Exponent<2047, then Value = (-1)**S * 2 ** (Exponent-1023) * (1.Mantissa) where "1.Mantissa" is intended to represent the binary number created by prefixing Mantissa with an implicit leading 1 and a binary point. If Exponent = 0 and Mantissa is nonzero, then Value = (-1)**S * 2 ** (-1022) * (0.Mantissa). These are "unnormalized" values. If Exponent = 0 and Mantissa is zero and S is 1, then Value = -0. * If Exponent = 0 and Mantissa is zero and S is 0, then Value = 0.
datetime	String containing a date-time per DSP0004
char16	Sixteen-bit UCS-2 character as defined in ISO/IEC 10646
enum4	Sequential enumeration, starting from 0 as the default, with optional numeric declarator. The number 4 indicates that the enum is encoded using an 4-bit binary number. Enum4 enumeration values shall be presented in decimal in PLDM specifications.
enum8	Sequential enumeration, starting from 0 as the default, with optional numeric declarator. The number 8 indicates that the enum is encoded using an 8-bit binary number. Enum8 enumeration values shall be presented primarily in decimal in PLDM specifications Example: enum8 { fred, mary, bob, george } has the value 0 correspond to fred, 1 for mary, 2 for bob, and 3 for george. A value may be explicitly declared such as: enum { fred, mary=2, bob, george }, in which case 0 corresponds to fred, 2 corresponds to mary, and 4 corresponds to george.

Data Type	Interpretation
timestamp104	Binary datetime type formatted as a series of 13 bytes, as follows: (Generally, this format can be mapped to a CIM Datetime timestamp value.) byte 12 UTC and Time resolution The CIM Datetime format allows a variable number of significant digits to be represented for the date/time and UTC fields using a '*' character in the string to indicate which contiguous digit positions should be ignored, starting from the least significant position. PLDM generally supports this format by using this byte to present an enumeration for the resolution. [7:4] UTC resolution = enum4 {UTCunspecified = 0, minute = 1, 10minute = 2, hour = 3 } [3:0] Time resolution = enum4 { microsecond = 0, 10microsecond = 1, 100microsecond = 2, millisecond = 3, 10millisecond = 4, 100millisecond = 5, second = 6, 10second = 7, minute = 8, 10minute = 9, hour = 10, day = 11, month = 12, year = 13, null (see below) = 15 } bytes 11:10 year as uint16 byte 9 month as uint8 (starting with 1) byte 8 day within the month as uint8 (starting with 1) byte 7 hour within the day as uint8 (24-hour representation starting with 0) byte 6 minute within the hour as uint8 (starting with 0) byte 5 seconds within the minute as uint8 (starting with 0) byte 4:2 microsecond within the second as a 24-bit binary integer (starting with 0) bytes 1:0 UTC offset in minutes as sint16 If the time resolution bits in byte 12 are set to enumeration value null (15), the timestamp value shall be interpreted as an unknown time.
interval72	Binary datetime interval formatted as a series of 9 bytes, as follows: (Generally, this format can be mapped to a CIM Datetime interval value.) byte 8 Time resolution [7:4] reserved [3:0] enum4 { microsecond = 0, 10microsecond = 1, 100microsecond = 2, 1millisecond = 3, 10millisecond = 4, 100millisecond = 5, second = 6, 10second = 7, minute = 8, hour = 9, day = 10, 10day = 11, 100day = 12} byte 7:6 number of days as uint16 (starting with 1) NOTE: CIM DateTime specifies this as six-digit field. byte 5 hour within the day as uint8 (24-hour representation starting with 0) byte 4 minute within the hour as uint8 (starting with 0) byte 3 seconds within the minute as uint8 (starting with 0) bytes 2:0 microsecond within the second as a 24-bit binary integer (starting with 0)
ver32	Thirty-two-bit encoding of a version number. The encoding of the version number and alpha fields is defined in 5.6. [31:24] = major version number [23:16] = minor version number [15:8] = update version number [7:0] = "alpha" byte
UUID	See 5.5.
bitfield8	Byte with 8 bit fields. Each of these bit fields can be defined separately.
bitfield16	Two-byte word with 16 bit fields. Each of these bit fields can be defined separately.

Data Type	Interpretation
strASCII	A null terminated 8-bit per character string. Unless otherwise specified, characters are encoded using the 8-bit ISO8859-1 "ASCII + Latin1" character set encoding. All strASCII strings shall have a single null (0x00) character as the last character in the string. Unless otherwise specified, strASCII strings are limited to a maximum of 256 bytes including the null terminator.
strUTF-8	A null terminated, UTF-8 encoded string per RFC3629. UTF-8 defines a variable length for Unicode encoded characters where each individual character may require one to four bytes. All strUTF-8 strings shall have a single null character as the last character in the string with encoding of the null character per RFC3629 Unless otherwise specified, strUTF-8 strings are limited to a maximum of 256 bytes including the null terminator.
strUTF-16	A null terminated, UTF-16 encoded string with Byte Order Mark (BOM) per RFC2781. All strUTF-16 strings shall have a single null (0x0000) character as the last character in the string. An empty string shall be represented using two bytes set to 0x0000, representing a single null (0x0000) character. Otherwise, the first two bytes shall be the BOM. Unless otherwise specified, strUTF-16 strings are limited to a maximum of 256 bytes including the BOM and null terminator.
strUTF16LE	A null terminated, UTF-16, "little endian" encoded string per RFC2781. All strUTF-16LE strings shall have a single null (0x0000) character as the last character in the string. Unless otherwise specified, strUTF16LE strings are limited to a maximum of 256 bytes including the null terminator.
strUTF-16BE	A null terminated, UTF-16, "big-endian" encoded string per RFC2781. All strUTF-16BE strings shall have a single null (0x0000) character as the last character in the string. Unless otherwise specified, strUTF16BE strings are limited to a maximum of 256 bytes including the null terminator.

5.5 UUID

The format of the ID follows the byte (octet) format specified in RFC4122. RFC4122 specifies the following four different versions of UUID formats and generation algorithms suitable for use with PLDM:

- version 1 (0001b) ("time based")
- version 3 (0011b) "MD5 hash" ("name-based")
- version 4 (0100b) "Pseudo-random" ("name-based")
- version 5 "SHA1 hash" ("name-based")

The version 1 format is recommended. A UUID value should never change over the lifetime of the device or software version associated with the UUID.

For PLDM, the individual fields within the UUID are transferred in network byte order (most-significant byte first) per the convention described in RFC4122. For example, Table 3 shows byte order for a UUID in version 1 format.

Table 3 – Example UUID format

Field	UUID Byte	MSB
time low	1	MSB
	2	
	3	
	4	
time mid	5	MSB
	6	
time high and version	7	MSB
	8	

Field	UUID Byte	MSB
clock seq high and reserved	9	
clock seq low	10	
Node	11	
	12	
	13	
	14	
	15	
	16	

5.6 Ver32 encoding

The version field is comprised of four bytes referred to as the "major," "minor," "update," and "alpha" bytes. These bytes shall be encoded as follows:

- The "major," "minor," and "update" bytes are BCD-encoded, and each byte holds two BCD digits.
- The "alpha" byte holds an optional alphanumeric character extension that is encoded using the ISO/IEC 8859-1 Character Set.
- The semantics of these fields follow those in DSP4004.
- The value `0x00` in the alpha field means that the alpha field is not used. Software or utilities that display the version number should not display any characters for this field.
- The value `0xF` in the most-significant nibble of a BCD-encoded value indicates that the most-significant nibble should be ignored and the overall field treated as a single-digit value. Software or utilities that display the number should display only a single digit and should not put in a leading "0" when displaying the number.
- A value of `0xFF` in the "update" field indicates that the entire field is not present. `0xFF` is not allowed as a value for the "major" or "minor" fields. Software or utilities that display the version number should not display any characters for this field.

EXAMPLE:

Version 3.7.10a → `0xF3F71061`

Version 10.01.7 → `0x1001F700`

Version 3.1 → `0xF3F1FF00`

Version 1.0a → `0xF1F0FF61`

5.7 Notations

The following notations are used for PLDM specifications:

- M:N — In field descriptions, this will typically be used to represent a range of byte offsets starting from byte M and continuing to and including byte N (M ≤ N). The lowest offset is on the left, and the highest is on the right.
- rsvd — Abbreviation for Reserved. Case insensitive.
- [4] — Square brackets around a number are typically used to indicate a bit offset. Bit offsets are given as zero-based values (that is, the least significant bit [LSb] offset = 0).

- [7:5] A range of bit offsets. The most-significant is on the left, and the least-significant is on the right.
- `1b` A lowercase "`b`" after a number consisting of `0`s and `1`s indicates that the number is in binary format.
- `0x12A` A leading "`0x`" indicates that the number is in hexadecimal format.

6 PLDM base version

The version of this Platform Level Data Model (PLDM) Base Specification shall be 1.1.0 (major version number 1, minor version number 1, update version number 0, and no alpha version).

In response to the GetPLDMVersion command in clause 8.2, the reported version for Type 0 (PLDM base, this specification) shall be encoded as 0xF1F1F000.

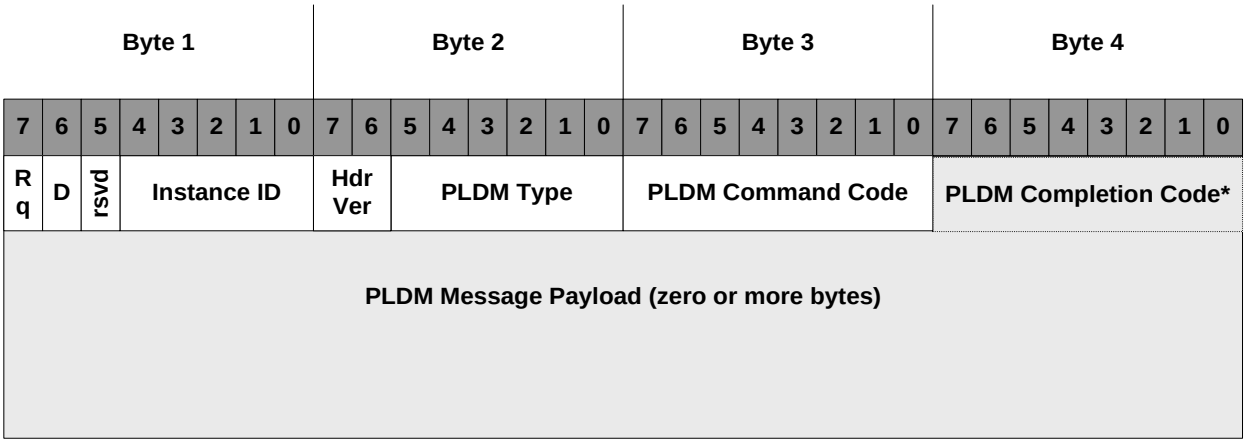
7 PLDM base protocol

The PLDM base protocol defines the common fields for PLDM messages and their usage.

Though there are command-specific PLDM header fields and trailer fields, the fields for the base protocol are common for all PLDM messages. These common fields support the identification of payload type, message, PLDM Type, and PLDM command/completion codes. The base protocol's common fields include a PLDM Type field that identifies the particular class of PLDM messages.

7.1 PLDM message fields

Figure 1 shows the fields that constitute a generic PLDM message. The fields within PLDM messages are transferred from the lowest offset first.



*****The PLDM Completion Code is present only in PLDM response messages.

Figure 1 – Generic PLDM message fields

563 Table 4 defines the common fields for PLDM messages.

564 **Table 4 – PLDM Message common fields**

<table>
<tr><th>Field Name</th><th>Field Size</th><th>Description</th></tr>
<tr><td>Rq</td><td>1 bit</td><td>Request bit, used to help differentiate between PLDM request messages and other PLDM messages.

This field is set to 1b for PLDM request messages and unacknowledged datagram request messages.

This field is set to 0b for PLDM response messages. See the following row of this table for valid combinations of Rq and D bits.</td></tr>
<tr><td>D</td><td>1 bit</td><td>Datagram bit, used to indicate whether the Instance ID field is being used for tracking and matching requests and responses, or just being used for asynchronous notifications.

This field is set to 1b for asynchronous notifications.

This field is set to 0b to indicate that the Instance ID field is being used for tracking and matching requests and responses.

Rq and D bit combinations:
<table>
<tr><th>Rq</th><th>D</th><th>Meaning</th></tr>
<tr><td>0b</td><td>0b</td><td>For PLDM response messages</td></tr>
<tr><td>0b</td><td>1b</td><td>Reserved</td></tr>
<tr><td>1b</td><td>0b</td><td>For PLDM request messages</td></tr>
<tr><td>1b</td><td>1b</td><td>For unacknowledged PLDM request messages or asynchronous notifications</td></tr>
</table></td></tr>
<tr><td>rsvd</td><td>1 bit</td><td>Reserved</td></tr>
<tr><td>Instance ID</td><td>5 bits</td><td>The Instance ID (Instance Identifier) field is used to identify a new instance of a PLDM request to differentiate new PLDM requests that are sent to the same PLDM terminus. The Instance ID field is used to match up a particular instance of a PLDM response message with the corresponding instance of PLDM request message.

If the requester issued a non-idempotent command, it shall complete any retries for that command before issuing a command with a new Instance ID.</td></tr>
<tr><td>Hdr Ver</td><td>2 bits</td><td>The Hdr Ver (Header Version) field identifies the header format. For this version of the specification, the value is set to 00b. This version applies to the PLDM message format.</td></tr>
<tr><td>PLDM Type</td><td>6 bits</td><td>The PLDM Type field identifies the type of PLDM that is being used in the control or data transfer carried out using this PLDM message. The PLDM Type field allows PLDM messages to be grouped together based on functions. See DSP0245 for the definitions of PLDM Type values.</td></tr>
<tr><td>PLDM Command Code</td><td>8 bits</td><td>For PLDM request messages, the PLDM Command Code field identifies the type of operation the message is requesting. The PLDM command code values are defined per PLDM Type. The PLDM Command Code that is sent in a PLDM request message shall be returned in the corresponding PLDM response message.</td></tr>
</table>

Field Name	Field Size	Description
PLDM Message Payload	Variable	The PLDM message payload is zero or more bytes that are specific to a particular PLDM Message. By convention, the PLDM Message formats are described using tables with the first byte of the payload identified as byte 0. NOTE: The baseline PLDM message payload size is PLDM Type-specific.
PLDM Completion Code	8 bits	The PLDM Completion Code field provides the status of the operation. This field is the first byte of the PLDM Message Payload for PLDM response messages and is not present in PLDM request messages. This field indicates whether the PLDM command completed normally. If the command did not complete normally, then the completion code provides additional information regarding the error condition. The PLDM Completion Code can be generic or PLDM Type-specific.

7.2 Generic PLDM completion codes (PLDM_BASE_CODES)

The command completion code fields are used to return PLDM operation results in the PLDM response messages. On a successful completion of a PLDM operation, the specified response parameters (if any) shall also be returned in the response message. For a PLDM operation resulting in an error, unless otherwise specified, the responder shall not return any additional parametric data and the requester shall ignore any additional parameter data provided in the response.

Table 5 defines the generic completion codes for the PLDM commands. PLDM Type-specific command completion codes are defined in the respective PLDM specification. Unless otherwise specified in a PLDM specification, specific error completion codes are optional. If a PLDM command completes with an error, the generic failure message (ERROR), an appropriate generic error completion code from Table 5, or a PLDM Type-specific error completion code shall be returned. For an unsupported PLDM command, the ERROR_UNSUPPORTED_PLDM_CMD completion code shall be returned unless the responder is in a transient state (not ready), in which it cannot process the PLDM command. If the responder is in a transient state, it may return the ERROR_NOT_READY completion code.

579 **Table 5 – Generic PLDM completion codes (PLDM_BASE_CODES)**

Value	Name	Description
0 (0x00)	SUCCESS	The PLDM command was accepted and completed normally.
1 (0x01)	ERROR	This is a generic failure message to indicate an error processing the corresponding request message. It should not be used when a more specific error code applies.
2 (0x02)	ERROR_INVALID_DATA	The PLDM request message payload contained invalid data or an illegal parameter value.
3 (0x03)	ERROR_INVALID_LENGTH	The PLDM request message length was invalid. (The PLDM request message body was larger or smaller than expected for the particular PLDM command.)
4 (0x04)	ERROR_NOT_READY	The Receiver is in a transient state where it is not ready to process the corresponding PLDM command.
5 (0x05)	ERROR_UNSUPPORTED_PLDM_CMD	The command field in the PLDM request message is unspecified or not supported for this PLDM Type. This completion code shall be returned for any unsupported command values received.
32 (0x20)	ERROR_INVALID_PLDM_TYPE	The PLDM Type field value in the PLDM request message is invalid or unsupported.
128-255 (0x80-0xFF)	COMMAND_SPECIFIC	This range of completion code values is reserved for values that are specific to a particular PLDM request message. The particular values (if any) and their definition is provided in the specification for the particular PLDM command.
All other	Reserved	Reserved

580 ## 7.3 Concurrent PLDM command processing

581 This clause describes the specifications and requirements for handling concurrent overlapping PLDM
582 requests.

583 ### 7.3.1 Requirements for responders

584 A PLDM terminus is not required to process more than one request at a time (that is, it can be "single
585 threaded" and does not have to accept and act on new requests until it has finished responding to any
586 previous request).

587 A responder that is not ready to accept a new request can either silently discard the request, or it can
588 respond with an `ERROR_NOT_READY` message completion code (preferred).

589 The PLDM does not restrict any specific model for the number of requesters or responders that can
590 communicate simultaneously. The PLDM specification allows an implementation to have a responder that
591 handles one request at a time and to not maintain contexts for multiple requests or multiple requesters.

592 If a PLDM terminus is working on a request from a requester, then the PLDM terminus shall be able to
593 process (or queue up processing) and send the response independently from sending its own request.

When a responder allows simultaneous communications with multiple requesters, the requirements on the responder are as follows:

- The responder shall use the following fields to track a PLDM request: the transport address (which is transport-binding specific, for example EID for MCTP transport) of the requester, PLDM Type, PLDM Command Code, and Instance ID of the PLDM request.
- If the responder runs out of internal resources, it may fail PLDM requests.

7.3.2 Requirements for requesters

A PLDM terminus that issues PLDM requests to another PLDM terminus shall wait until one of the following occurs before issuing a new PLDM request: it gets the response to a particular request, it times out waiting for the response, or it receives an indication that transmission of the particular request failed.

A PLDM terminus that issues PLDM requests is allowed to have multiple simultaneous requests outstanding to *different* responders.

A PLDM terminus that issues PLDM requests should be prepared to handle the order of responses that may not match the order in which the requests were sent (that is, it should not automatically assume that a response that it receives is in the order in which the request was sent). It should check to see that the PLDM Type, PLDM Command Code, and Instance ID values in the response match up with a corresponding outstanding command before acting on any parameters returned in the response.

The timing specifications shown in Table 6 are specific to PLDM request messages. The PLDM responses are not retried. A “try” or “retry” of a request is defined as a complete transmission of the PLDM request message.

Table 6 – Timing specifications for PLDM messages

Timing Specification	Symbol	Min	Max	Description
Number of request retries	PN1	2	See "Description"	The number of times a requester is obligated to retry a request. Total of three tries, minimum: the original try plus two retries. The maximum number of retries for a given request is limited by the requirment that all retries shall occur within $PT3_{Max}$ of the initial request.
Request-to-response time	PT1	–	100 msec	The amount of time a responder has to begin transmission of a response message. This interval is measured at the responder from the end of the reception of the PLDM request to the beginning of the transmission of the response. This requirement is tested under the condition where the responder can successfully transmit the response on the first try.

Timing Specification	Symbol	Min	Max	Description
Time-out waiting for a response	PT2	$PT1_{Max}$+ 2*$PT4_{Max}$	$PT3_{Min}$ – 2*$PT4_{Max}$	The amount of time a requester has to wait for a response message. This interval is measured at the requester from the end of the successful transmission of the PLDM request to the beginning of the reception of the corresponding PLDM response. This interval at the requester sets the minimum amount of time that a requester should wait before retrying a PLDM request. Note: This specification does not preclude an implementation from adjusting the minimum time-out waiting for a response to a smaller number than PT2 based on measured response times from responders. The mechanism for doing so is outside the scope of this specification.
Instance ID expiration interval	PT3	5 sec [1]	6 sec	This is the interval after which the Instance ID for a given response will expire and become reusable if a response has not been received for the request. This is also the maximum time that a responder tracks an Instance ID for a given request from a given requester.
Transmission Delay	PT4	–	100 ms	Time to take into account transmission delay of a PLDM Message. Measured as the time between the end of the transmission of a PLDM message at the transmitter to the beginning of the reception of the PLDM message at the receiver.
Not ready retry delay	PT5	250 ms	–	Time requestor must wait before retrying a command when receiving completion code ERROR_NOT_READY This interval is measured as the time between when the requester finishes receiving a response containing the ERROR_NOT_READY completion code and when the requeser begins retransmitting a retry. NOTE:There are no requirements for a retry delay when a request receives a response other than ERROR_NOT_READY, or when a request does not receive a response at all.

NOTE: [1] If a requester is reset, it may produce the same Instance ID for a request as one that was previously issued. To guard against this, it is recommended that Instance ID expiration be implemented. Any request from a given requester that is received more than PT3 seconds after a previous, matching request should be treated as a new request, not a retry.

8 PLDM messaging control and discovery commands

The PLDM base definition supports a PLDM Type field that allows the commands to be grouped using a PLDM Type. This section contains detailed descriptions for PLDM messages that are used for control and discovery operations. The PLDM commands for PLDM messaging control and discovery are also defined in this section.

620 Table 7 defines the PLDM command codes for PLDM messaging control and discovery.

621 **Table 7 – PLDM messaging control and discovery command codes**

Command	Code Value	Requirement	Section
SetTID	0x01	Optional	See 8.1.1.
GetTID	0x02	Mandatory	See 8.1.2.
GetPLDMVersion	0x03	Mandatory	See 8.2.
GetPLDMTypes	0x04	Mandatory	See 8.3.
GetPLDMCommands	0x05	Mandatory	See 8.4.
SelectPLDMVersion	0x06	Conditional [1]	See 8.5
NegotiateTransferParameters	0x07	Conditional [2]	See 8.6.3.
MultipartSend	0x08	Optional	See 8.6.4.
MultipartReceive	0x09	Optional	See 8.6.5.

622 Conditional requirements:

623 1. Implementing SelectPLDMVersion is mandatory only if a terminus advertises support for multiple
624 versions of any given PLDM type.

625 2. Implementing NegotiateTransferParameters is mandatory for PLDM Termini that support MultipartSend
626 or MultipartReceive for any version of any PLDM Type.

627 ## 8.1 PLDM Terminus

628 A PLDM Terminus is defined as the point of communication termination for PLDM messages and the
629 PLDM functions associated with those messages. Given a PLDM terminus, a mechanism is required that
630 can uniquely identify each terminus so that the semantic information can be bound to that identification.
631 The Terminus ID (TID) is a value that identifies a PLDM terminus. TIDs are used in PLDM messages
632 when it is necessary to identify the PLDM terminus that is the source of the PLDM Message. TIDs are
633 defined within the scope of PLDM Messaging.

634 ### 8.1.1 SetTID command (0x01)

635 The SetTID command is used to set the Terminus ID (TID) for a PLDM Terminus. This command is
636 typically only used by the PLDM Initialization Agent function. The command format is shown in Table 8.

637 **Table 8 – SetTID command format**

Byte	Type	Request Data
0	uint8	**TID** Special value: 0x00, 0xFF = reserved.
Byte	**Type**	**Response Data**
0	enum8	**completionCode** Possible values: { PLDM_BASE_CODES }

8.1.2 GetTID command (0x02)

The GetTID command is used to retrieve the present Terminus ID (TID) setting for a PLDM Terminus. The command format is shown in Table 9.

Table 9 – GetTID command format

Byte	Type	Request Data
0	–	No request data
Byte	**Type**	**Response Data**
0	enum8	**completionCode** possible value: { PLDM_BASE_CODES }
1	uint8	**TID** special value: 0x00 – Unassigned TID, 0xFF – reserved

8.2 GetPLDMVersion (0x03)

The GetPLDMVersion command can be used to retrieve the PLDM base specification versions that the PLDM terminus supports, as well as the PLDM Type specification versions supported for each PLDM Type. The format of the request and response message parameters for this command is shown in Table 10.

More than one version number can be returned for a given PLDM Type by the GetPLDMVersion command. This enables the command to be used for reporting different levels of compatibility and for backward compatibility with different specification versions. The individual specifications for the given PLDM Type define the requirements for which version number values should be used for that PLDM Type. Those documents define which earlier version numbers, if any, shall also be listed.

Generally, implementations that do not support SelectPLDMVersion should only report a single version for a PLDM Type as there is no way without this command to target a particular version of the Type. When interacting with a terminus that advertises support for more than one version of a PLDM Type but does not support the SelectPLDMVersion command, the requestor should assume that commands sent will be responded to following the highest version advertised as supported. Likewise, responders that advertise support for more than one version of a PLDM Type and do not implement the SelectPLDMVersion command should respond according to the highest version of the PLDM Type that they support.

The command returns a completion code that indicates whether the PLDM Type number passed in the request is supported. This enables the command to also be used to query the endpoint for whether it supports a given PLDM Type.

Table 10 – GetPLDMVersion request and response message format

Byte	Type	Request Data
0:3	uint32	**DataTransferHandle** This field is a handle that is used to identify PLDM version data transfer. This handle is ignored by the responder when the TransferOperationFlag is set to GetFirstPart.
4	enum8	**TransferOperationFlag** This field is an operation flag that indicates whether this is the start of the transfer. Value: {GetNextPart=0, GetFirstPart=1}

Byte	Type	Request Data
5	uint8	**PLDMType** This field identifies the PLDM Type whose version information is being requested. See DSP0245 for valid PLDMType values.
Byte	**Type**	**Response Data**

0	enum8	**CompletionCode**
		possible values:
		{
		PLDM_BASE_CODES,
		INVALID_DATA_TRANSFER_HANDLE=128 (0x80),
		INVALID_TRANSFER_OPERATION_FLAG=129 (0x81),
		INVALID_PLDM_TYPE_IN_REQUEST_DATA=131(0x83)
		}

1:4	uint32	**NextDataTransferHandle** This field is a handle that is used to identify the next portion of PLDM version data transfer.
5	enum8	**TransferFlag** This field is the transfer flag that indicates what part of the transfer this response represents. Possible values: {Start=1, Middle=2, End=4, StartAndEnd = 5}
Variable	–	**Portion of PLDMVersionData** (contains one or more version fields as described in Table 11) See Table 11 for the format.

664 **Table 11 – PLDM representation of PLDMVersionData**

Byte	Type	Field
0:3	ver32	**Version[0]** This field is the first entry of the version supported for the specified PLDM type.
…	…	…
4*(N-1):4*N-1	ver32	**Version[N-1]** This field is the N^{th} entry of the version supported for the specified PLDM type.
4*N:4*N+3	uint32	**PLDMVersionDataIntegrityChecksum** Integrity checksum on the PLDM version data. It is calculated starting at the first byte of the PLDM representation of PLDMVersionData. For this specification, 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.

665 This command is defined in such a manner that it allows the PLDM version data to be transferred using a
666 sequence of one or more command or response messages. When more than one command is used to
667 transfer the PLDM version data, the response messages contain the non-overlapping contiguous portions
668 of PLDM version data as defined in Table 11. By combining the portions of PLDM version data from the
669 response messages, the entire PLDM version data can be reconstructed.

8.3 GetPLDMTypes (0x04)

The GetPLDMTypes command can be used to discover the PLDM type capabilities supported by a PLDM terminus and to get a list of the PLDM types that are supported. The request and response parameters for this message are listed in Table 12.

The response to this command may be specific according to which transport endpoint over which the request was received (that is, a device that supports a given PLDM Type on a transport endpoint may not support that PLDM Type equally across all the transport endpoints that connect to the device).

Table 12 – GetPLDMTypes request and response message format

Byte	Type	Request Data
-	-	None
Byte	**Type**	**Response Data**
0	enum8	**CompletionCode** Possible values: { PLDM_BASE_CODES}
1:8	bitfield8[8]	**PLDMTypes** Each bit represents whether a given PLDM Type is supported: 1b = PLDM Type is supported. 0b = PLDM Type is not supported. For bitfield8[N], where N = 0 to 7 [7] – PLDM Type N*8+7 Supported [..] – … [1] – PLDM Type N*8+1 Supported [0] – PLDM Type N*8+0 Supported

8.4 GetPLDMCommands (0x05)

The GetPLDMCommands command can be used to discover the PLDM command capabilities supported by aPLDM terminus for a specific PLDM Type and version as a responder. The request and response parameters for this message are listed in Table 13.

The response to this command may be specific according to which transport endpoint over which the request was received (that is, a device that supports a given PLDM Type on a transport endpoint may not support that PLDM Type equally across all the transport endpoints that connect to the device).

Table 13 – GetPLDMCommands request and response message format

Byte	Type	Request Data
0	uint8	**PLDMType** This field identifies the PLDM Type for which command support information is being requested. See DSP0245 for valid PLDMType values.
1:4	ver32	**Version** This field identifies the version for the specified PLDM Type.

Byte	Type	Response Data

0	enum8	**CompletionCode**
		Possible values:
		{
		PLDM_BASE_CODES,
		INVALID_PLDM_TYPE_IN_REQUEST_DATA=131(0x83)
		INVALID_PLDM_VERSION_IN_REQUEST_DATA=132(0x84)
		}

1:32	bitfield8[32]	**PLDMCommands** (up to 256 commands supported for the specified PLDM Type) Each bit represents whether a given PLDM command is supported: 1b = PLDM command is supported. 0b = PLDM command is not supported. For bitfield8[N], where N = 0 to 31 [7] – PLDM Command N*8+7 Supported [..] – … [1] – PLDM Command N*8+1 Supported [0] – PLDM Command N*8 Supported

8.5 SelectPLDMVersion (0x06)

The SelectPLDMVersion command can be used to specify the version of a PLDM type that a PLDM endpoint shall use when interpreting request messages and providing response messages for PLDM commands. The request and response parameters for this message are listed in Table 14.

A PLDM endpoint that supports multiple versions of a PLDM Type but has not received a SelectPLDMVersion command for that Type shall interpret request messages and provide response messages according to the highest version of the Type it supports. Similarly, any time PLDM is reset for a terminus, it shall revert to the highest version of each Type it supports.

PLDM termini are not responsible for ensuring compatibility among the versions of PLDM types that they are asked to use via this command. Even if version selection results in mutually incompatible types, it is the requester's responsibility to manage the resultant behavior. Termini are not obligated to track compatibility between different versions of PLDM types.

Table 14 – SelectPLDMVersion request and response message format

Byte	Type	Request Data
0	uint8	**PLDMType** This field identifies the PLDM Type for which version selection is being performed. This command may not be used to select a version for PLDM type 0. See DSP0245 for valid PLDM type values.
1:4	ver32	**Version** This field identifies the version for the specified PLDM Type that the endpoint shall use for interpreting request messages and providing response messages.

Byte	Type	Response Data

0	enum8	**CompletionCode**
		Possible values:
		{
		PLDM_BASE_CODES,
		INVALID_PLDM_TYPE_IN_REQUEST_DATA=131 (0x83),
		INVALID_PLDM_VERSION_IN_REQUEST_DATA=132 (0x84)
		}
		Response codes INVALID_PLDM_TYPE_IN_REQUEST_DATA and INVALID_PLDM_VERSION_IN_REQUEST_DATA shall additionally be used to indicate that a particular PLDM Type or version for that type is not supported.

8.6 Multipart transfer commands

The various commands defined in this clause support bulk transfers via the MultipartSend and MultipartReceive commands. The MultipartSend and MultipartReceive commands use flags and data transfer handles to perform multipart transfers of data. The transfer can be contiguous (one section) or composed of multiple sections sent separately. Each section of a larger block of data that is transferred is identified by its offset within the larger block so that both sender and receiver know what is being transferred and how to process it. Each section in turn may comprise multiple parts of data (the amount that can be sent in a single message). In multipart transfers, a data transfer handle identifies each part of the transfer. Data transfer handle values are implementation specific. For example, an implementation can use memory offsets or sequence numbers as data transfer handles.

Each multipart transfer is flagged with an indication of the PLDM Type on behalf of which the transfer is occurring as well as a type-specific context field that differentiates different types of transfers within the type.

8.6.1 Flag usage for MultipartSend

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

- Defining the initial DataTransferHandle and TransferContext for a multipart transfer is out of scope for this specification. These fields are PLDM Type specific and should be documented with the PLDM Type for the transfer. (Similarly, definition of section offsets and byte counts are out of scope for this specification.)
- DataTransferHandles are opaque handles that are specific to PLDM Types and TransferContexts. Recipients shall make no inferences about the numeric values of valid DataTransferHandles.
- When sending the first part of data in the current section, the sender shall set the TransferFlag in the request message to START (for a multi-part section) or START_AND_END (for a single-part section).
- To request the next part in the current section, the responder shall set NextTransferOperation to XFER_NEXT_PART in the response message. This may only be done if the TransferFlag in the request message was set to START or MIDDLE.
- To request that transfer of the current section of data be restarted with a MultipartSend command, the responder shall set NextTransferOperation to XFER_FIRST_PART in the response message. NOTE: The ability of the responder to request that the transmission of a

section be restarted at any time means that the requester must retain data for the section until the transfer is complete.

- To request retransmission of a part of data, such as upon detection of a checksum error, the responder shall set NextTransferOperation to XFER_CURRENT_PART in the response message.
- To convey that a part being sent is the last part of the current section, the requester shall set the TransferFlag to END (for a multi-part section) or START_AND_END (for a single-part section).
- To acknowledge successful transfer of a section of data, the responder shall set NextTransferOperation to XFER_COMPLETE. The transfer of the current section is complete when the requester receives a response message with a success CompletionCode and NextTransferOperation set to XFER_COMPLETE.
- The TransferFlag specified in the request for a MultipartSend command request message has the following meanings:
  - START, which is the first part of the data transfer for the current section
  - MIDDLE, which is neither the first nor the last part of the data transfer for the current section
  - END, which is the last part of the data transfer for the current section
  - START_AND_END, which is the first and the last part of the data transfer. In this case, the transfer for the current section consists of a single part

8.6.2 Flag usage for MultipartReceive

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

- Defining the initial DataTransferHandle and TransferContext for a multipart transfer is out of scope for this specification. These fields are PLDM Type specific and should be documented with the PLDM Type for the transfer. (Similarly, definition of section offsets and byte counts are out of scope for this specification.)
- DataTransferHandles are opaque handles that are specific to PLDM Types and TransferContexts. Recipients shall make no inferences about the numeric values of valid DataTransferHandles.
- To initiate transfer of a section of data with a MultipartReceive command, the requester shall set TransferOperation to XFER_FIRST_PART in the request message. This flag may also be used to request restart of the transfer for the current section. NOTE: The ability of the requester to request that the transmission of a section be restarted at any time means that the responder must retain data for the section until the transfer is complete.
- To request retransmission of the current part of data (such as upon detection of a checksum error), the requestor shall set TransferOperation to XFER_CURRENT_PART in the request message and shall set DataTransferHandle to the handle that was previously used with this part.
- To request the next part of data, the requestor shall set TransferOperation to XFER_NEXT_PART and shall set DataTransferHandle to the NextDataTransferHandle that was obtained in the response to the previous MultipartReceive command for this data transfer. This may only be done if TransferFlag in the previous response message was set to START or MIDDLE.
- To acknowledge successful transfer of the current section of data, the requestor shall set TransferOperation to XFER_COMPLETE. The requestor may initiate transfer of another section at the same time by specifying RequestedSectionLengthBytes and RequestedSectionOffset for

the new section In this case, the NextDataTransferHandle in the response message becomes the initial handle for the new section. The transfer of the current section is complete when the requester receives a response to a message in which TransferOperation is set to XFER_COMPLETE.

- The TransferFlag specified in the response of a MultipartReceive command has the following meanings:
  - START, which is the first part of the data transfer for the current section
  - MIDDLE, which is neither the first nor the last part of the data transfer for the current section
  - END, which is the last part of the data transfer for the current section
  - START_AND_END, which is the first and the last part of the data transfer for the current section

8.6.3 NegotiateTransferParameters (0x07)

The NegotiateTransferParameters command is used to establish transfer parameters and support for multipart transfers. In the event that a MultipartSend or MultipartReceive command is issued before this negotiation completes for a PLDM Type, the command shall be rejected by the receiving terminus. An endpoint that does not support the NegotiateTransferParameters command shall be considered as not supporting the MultipartSend and MultipartReceive commands, even if it reflects support for them in the response to the GetPLDMCommands command.

This command is typically only used by the PLDM Initialization Agent function. Because responder's support for Multipart transfers for a PLDM Type may be limited to some versions of that Type, Initialization Agent functions shall invoke this command after using any SelectPLDMVersion commands to determine whether MultipartSend and MultipartReceive are to be used with that PLDM Type.

The PLDM Initialization Agent function may invoke this command multiple times in order to establish different part sizes with different PLDM Types. The transfer part sizes established with each such invocation only apply to the PLDM Types selected by the Initialization Agent function in RequestorProtocolSupport. The part size value represents the size of the PLDM header and PLDM payload; upper-level protocol and medium specific headers shall not be counted against this size.

The command format is shown in Table 15.

Table 15 – NegotiateTransferParameters command format

Byte	Type	Request Data
0..1	uint16	**RequestorPartSize** Requestor's maximum transfer part size for a single message in a multipart transfer. Upon successful completion of this command, the requestor and responder shall use the lesser of **Requestor**PartSize and ResponderPartSize as the part size for messages sent via the MultipartSend and MultipartReceive commands. The minimum part size is 256 bytes; a value of less than 256 bytes shall be interpreted as a lack of support for the MultipartSend and MultipartReceive commands with all PLDM types on the part of the requestor. Implementations that support multipart transfers are strongly recommended to support at least 512 bytes for the part size.

Byte	Type	Request Data (continued)
2..9	bitfield8[8]	**RequestorProtocolSupport** Each bit represents whether multipart transfer with a given PLDM Type is supported by the requestor and negotiation of transfer part size for that Type should proceed: 1b = multipart transfer is supported and negotiation of transfer part size should proceed. 0b = multipart transfer is not supported or negotiation of transfer part size should not proceed with this invocation of the NegotiateTransferParameters command. For bitfield8[N], where N = 0 to 7 [7] – Multipart transfer with PLDM Type N*8+7 supported and negotiation should proceed [..] – … [1] – Multipart transfer with PLDM Type N*8+1 supported and negotiation should proceed [0] – Multipart transfer with PLDM Type N*8+0 supported and negotiation should proceed Upon successful completion of this command, the requestor and responder shall use the PLDM base MultipartSend and MultipartReceive commands with PLDM Types where both the requestor and the responder have indicated support and negotiated a part size.
Byte	**Type**	**Response Data**
0	enum8	**completionCode** Possible values: { PLDM_BASE_CODES } ERROR_INVALID_DATA: RequestorPartSize in the request message was less than 256 bytes or not a power of two.
1..2	uint16	**ResponderPartSize** Responder's maximum transfer part size for a single message in a multipart transfer. Upon successful completion of this command, the requestor and responder shall use the lesser of RequestorPartSize and ResponderPartSize as the part size for messages sent via the MultipartSend and MultipartReceive commands. The minimum part size is 256 bytes; a value of less than 256 bytes shall be interpreted as a lack of support for the MultipartSend and MultipartReceive commands with all PLDM types on the part of the responder. This indication may also be expressed by denying support for the NegotiateTransferParameters command. In the event that a requestor queries support for multiple PLDM Types and the responder supports different maximum transfer part sizes for those Types, it shall respond with the minimum transfer part size it supports across the supported Types. Implementations that support multipart transfers are strongly recommended to support at least 512 bytes for the part size.

Byte	Type	Response Data (continued)
3..10	bitfield8[8]	**ResponderProtocolSupport** Each bit represents whether multipart transfer with a given PLDM Type queried by the requestor is supported by the responder: 1b = multipart transfer is supported. 0b = multipart transfer is not supported or the requestor did not query support for this Type. For bitfield8[N], where N = 0 to 7 [7] – Multipart transfer with PLDM Type N*8+7 supported [..] – … [1] – Multipart transfer with PLDM Type N*8+1 supported [0] – Multipart transfer with PLDM Type N*8+0 supported Upon successful completion of this command, the requestor and responder shall use the PLDM base MultipartSend and MultipartReceive commands with PLDM Types where both the requestor and the responder have indicated support. The manner in which these commands are to be used for a given PLDM type is documented in the specification corresponding to that type.

8.6.4 MultipartSend (0x08)

This command enables a requestor to transmit a large volume of data to a responder in one or more sections, each of which is broken into a series of single-message parts. In the event of a data checksum error, the responder may ask the requestor to resend the current part or restart the transfer. Responders may also abort transmissions or restart them entirely; motivations for and expected behavior with such actions are protocol-specific and should be documented in the corresponding PLDM Type specification.

Table 16 – MultipartSend command format

Byte	Type	Request Data
0	uint8	**PLDMType** The PLDM Type for the protocol under which this transfer is being performed.
1	enum8	**TransferFlag** An indication of current progress within the transfer. The value START_AND_END indicates that the entire transfer consists of a single part. value: { START = 1, MIDDLE = 2, END = 4, START_AND_END = 5 }
2..5	uint32	**TransferContext** A protocol-specific indication of the context in which this transfer is being performed. See the documentation for the corresponding PLDM type for details of the way in which this field is used.
6..9	uint32	**DataTransferHandle** A handle to uniquely identify the part of data to be sent. The DataTransferHandle supplied shall be one of: the initial handle, to begin or restart a transfer; the most recently sent handle, to resend the previous part; or the NextDataTransferHandle as specified in the previous part.

Byte	Type	Request Data (continued)
10..13	uint32	**NextDataTransferHandle** The handle for the next part of data for this transfer. Special value: zero (0x00000000) if no further data remains to be transferred in the current section
14..17	uint32	**SectionOffset** The start offset when initiating transfer of a new section. Special value: Zero (0x00000000) if TransferFlag is not one of START or START_AND_END.
18..21	uint32	**SectionLengthBytes** The size in bytes of data to be supplied when initiating transfer of a new section. Special value: Zero (0x00000000) if data size is unavailable or if TransferFlag is not one of START or START_AND_END.
22..25	uint32	**DataLengthBytes** The length in bytes N of data being sent in this part in the Data field. This value and the data bytes associated with it shall not cause this request message to exceed the negotiated maximum transfer part size (clause 8.6.3).
26..N+26	uint8[N]	**Data** The current part of data.
N+27.. N+30	uint32	**DataIntegityChecksum** 32-bit cumulative CRC for the entirety of data received so far for this section (all parts concatenated together, excluding checksums). Shall be included with all part transfers. When beginning a transfer of a new section of data within a single MultipartSend sequence, the data integrity checksum for the previous section is not included in the calculation. 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.
Byte	**Type**	**Response Data**
0	enum8	**completionCode** Possible values: { PLDM_BASE_CODES, NEGOTIATION_INCOMPLETE=131 (0x83) }

Byte	Type	Response Data (continued)
1	enum8	**NextTransferOperation** The follow-up action that the responder is requesting of the requestor. NextTransferOperation flags are described in more detail in clause 8.6.1. • XFER_FIRST_PART: resend the initial part (restarting transmission of the section, such as if the receiver had to reallocate a larger receive buffer and restart the transfer) • XFER_NEXT_PART: send the next part of data for the current section • XFER_ABORT: stop the transmission and do not retry. The expected behavior for an aborted transmission is PLDM type-specific. • XFER_COMPLETE: transmission of the section completed normally. If there is another section to transmit, the sender may begin sending it at this time. • XFER_CURRENT_PART: resend the last part, such as if the checksum of data cumulatively received did not match the DataIntegrityChecksum in the current part value: { XFER_FIRST_PART = 0, XFER_NEXT_PART = 1, XFER_ABORT = 2, XFER_COMPLETE = 3, XFER_CURRENT_PART = 4 }

8.6.5 MultipartReceive (0x09)

This command enables the requestor to receive a large volume of data from a responder in one or more sections, each of which is broken into a series of single-message parts. In the event of a data checksum error, the requestor may ask the responder to resend the current part or restart the transfer. Responders may also abort transmissions or restart them entirely; motivations for and expected behavior with such actions are protocol-specific and should be documented in the corresponding PLDM Type specification.

Table 17 – MultipartReceive command format

Byte	Type	Request Data
0	uint8	**PLDMType** The PLDM Type for the protocol under which this transfer is being performed.
1	enum8	**TransferOperation** The section of data requested for the transfer. TransferOperation flags are described in more detail in clause 8.6.2. • XFER_FIRST_PART: The requestor is asking that the section transfer begin or restart from the beginning • XFER_NEXT_PART: The requestor is asking for the next part of the current section • XFER_ABORT: The requestor is requesting that the transfer be discarded. Handling of aborted transfers is PLDM Type specific; see the documentation for the corresponding PLDM type. • XFER_COMPLETE: The requestor is acknowledging completion of transfer for the current section of data and may be requesting another section of data • XFER_CURRENT_PART: The requestor is asking for the part just sent to be retransmitted, such as if the checksum of data cumulatively received did not match the DataIntegrityChecksum in the current part value: { XFER_FIRST_PART = 0, XFER_NEXT_PART = 1, XFER_ABORT = 2, XFER_COMPLETE = 3, XFER_CURRENT_PART = 4 }

Byte	Type	Request Data (continued)
2..5	uint32	**TransferContext** A protocol-specific indication of the context in which this transfer is being performed. See the documentation the corresponding PLDM type for details of the way in which this field is used.
6..9	uint32	**DataTransferHandle** A handle to uniquely identify the part of data to be received. The DataTransferHandle supplied shall be one of: • the initial handle, to begin or restart a transfer for a section of data • the previous handle, when requesting retransmission with the XFER_CURRENT flag • the NextDataTransferHandle as specified in the previous part, to obtain the next part Special Value: Zero (0x00000000) when acknowledging completion of a section transfer with the XFER_COMPLETE flag
10..13	uint32	**RequestedSectionOffset** The requested start offset for a new section. Special Value: Zero (0x00000000) if TransferOperation is not one of XFER_FIRST_PART or XFER_COMPLETE. It may also be zero if TransferOperation is XFER_COMPLETE, the last part was successfully received, and no further sections are to be transferred,
14..17	uint32	**RequestedSectionLengthBytes** The size in bytes of data requested for a new section. Special value: Zero (0x00000000) if data size is unavailable or if TransferOperation is not one of XFER_FIRST_PART or XFER_COMPLETE. It may also be zero if TransferOperation is XFER_COMPLETE, the last part was successfully received, and no further sections are to be transferred,
Byte	**Type**	**Response Data**
0	enum8	**completionCode** Possible values: { PLDM_BASE_CODES, NEGOTIATION_INCOMPLETE=131 (0x83) }
1	enum8	**TransferFlag** value: { START = 1, MIDDLE = 2, END = 4, START_AND_END = 5, ACKNOWLEDGE_COMPLETION = 8 }
2..5	uint32	**NextDataTransferHandle** The handle for the next part of data for this section transfer. Special Value: Zero (0x00000000) if no further data remains for the transfer of this section. In response to a request message where TransferOperation was set to XFER_COMPLETE that requested transfer of another section of data, this shall be the initial DataTransferHandle for that section.
6..9	uint32	**DataLengthBytes** The length in bytes N of data being sent in this part in the Data field. This value and the data bytes associated with it shall not cause this response message to exceed the negotiated maximum transfer part size (clause 8.6.3). This value shall be zero in response to a request message where TransferOperation was set to XFER_COMPLETE.

Byte	Type	Response Data (continued)
10.. N+10	uint8[N]	**Data** The current part of data. This field shall be omitted in response to a request message where TransferOperation was set to XFER_COMPLETE.
N+11.. N+14	uint32	**DataIntegrityChecksum** 32-bit cumulative CRC for the entirety of data received so far for this section (all parts concatenated together, excluding checksums). Shall be included with all part transfers. When beginning a transfer of a new section of data within a single MultipartReceive sequence, the data integrity checksum for the previous section is not included in the calculation. 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. This field shall be omitted in response to a request message where TransferOperation was set to XFER_COMPLETE.

9 PLDM messaging control and discovery examples

The GetPLDMVersion command (see 8.2) for transferring PLDM version data supports multipart transfers. The GetPLDMVersion command uses flags and data transfer handles to perform multipart transfers. The following requirements apply to the usage of TransferOperationFlag, TransferFlag, and DataTransferHandle for a given data transfer:

1) For initiating a data transfer (or getting the first part of data) by using a Get* command, the TransferOperationFlag shall be set to GetFirstPart in the request of the Get* command.

   - For transferring any part of the data other than the first part by using a Get* command, the TransferOperationFlag shall be set to GetNextPart and the DataTransferHandle shall be set to the NextDataTransferHandle that was obtained in the response of the previous Get* command for this data transfer.
   - The TransferFlag specified in the response of a Get* 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.
     – StartAndEnd, which is the first and the last part of the data transfer.
   - The requester shall consider a data transfer complete when the TransferFlag in the response of a Get* command is set to End or StartAndEnd.

EXAMPLE 1: The example in Figure 2 shows how multipart transfers can be performed using the generic mechanism defined in the GetPLDMVersion command. In Figure 2, the PLDM version data is transferred in three parts. Figure 2 shows the flow of the data transfer.

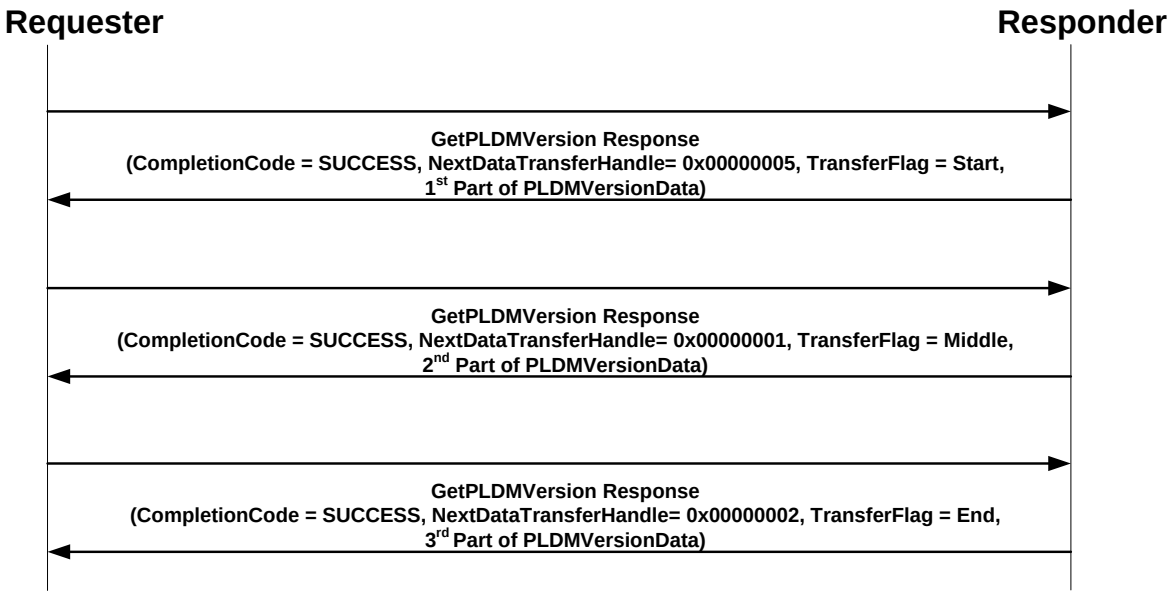


Figure 2 – Example of multipart PLDM version data transfer using the GetPLDMVersion command

EXAMPLE 2: Figure 3 shows an example sequence of steps performed by a requester to discover the PLDM versions and types supported by the responder as well as the commands supported for each PLDM type.

In the example, the following steps are performed by the requester:

1) The requester first uses the GetTID command to get the PLDM Terminus ID of the responder.

2) The requester then uses GetPLDMTypes to discover the PLDM types supported by the responder. (In the example shown in Figure 3, the responder supports two PLDM types, PLDM Type 0 and PLDM Type 1.)

3) For each PLDM type that is supported by the responder, the requester uses GetPLDMVersion and GetPLDMCommands to discover the supported versions of the specifications for the PLDM type and the supported PLDM commands for the specific PLDM version and type. In this example, the responder supports only one version of the specification (1.0.0) for each PLDM Type.

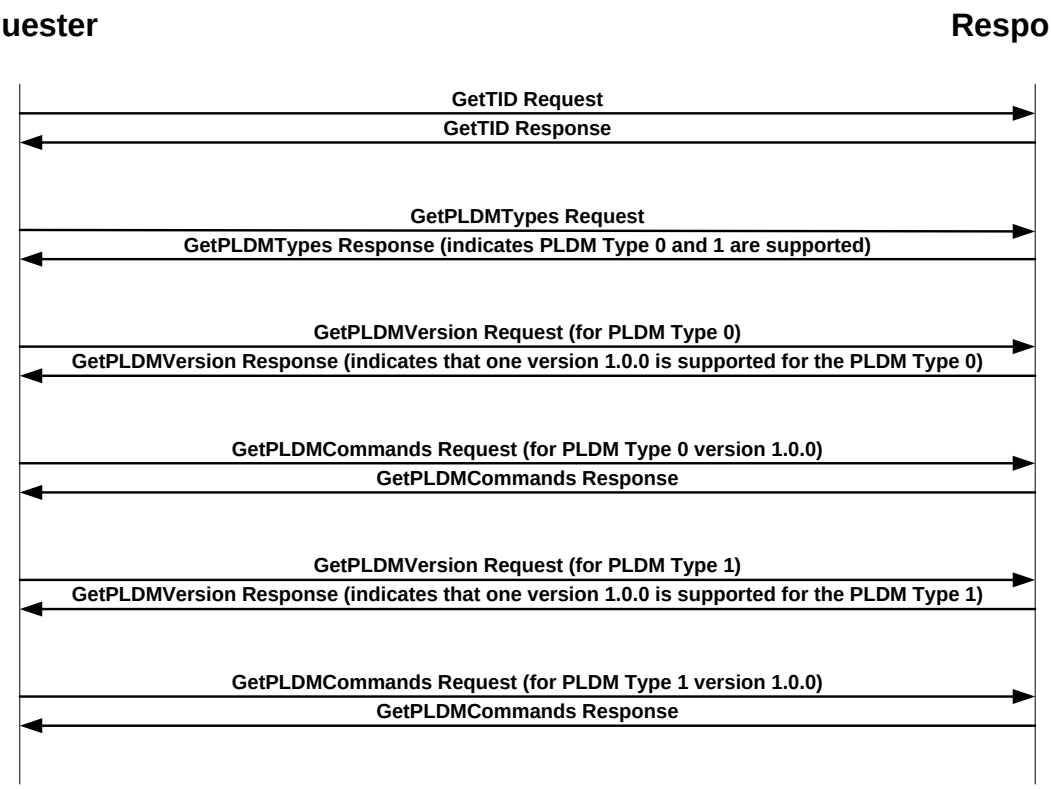


Figure 3 – PLDM discovery command example

ANNEX A
(Informative)

Change log

Version	Date	Description
1.0.0	2008-04-23	
1.1.0	2021-02-11	• Add standard string types to baseline list of PLDM data types • Add enum4 to baseline list of PLDM data types • Add contributors list • Add support for unknown timestamp value • Clarify use of D/Rq bits in PLDM message headers • Add timing parameter PT5 for time delay between retries • Add common multipart transfer commands • Add SelectPLDMVersion command to enable concurrent support of multiple versions of a PLDM type