1 **Document Identifier: DSP0274**

2 **Date: 2024-12-11**

3 **Version: 1.4.0WIP70**

4

Security Protocol and Data Model (SPDM) Specification

Information for Work-in-Progress version:

5 **IMPORTANT:** This document is not a standard. It does not necessarily reflect the views of 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 This document is available for public review and comment until superseded.

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

7 **Supersedes: 1.3.0**

8 **Document Class: Normative**

9 **Document Status: Work in Progress**

10 **Document Language: en-US**

CONTENTS

16 1 Foreword

17 The *Security Protocols and Data Models (SPDM)* Working Group of the DMTF prepared the *Security Protocol and Data Model (SPDM) Specification* (DSP0274). DMTF is a not-for-profit association of industry members that promotes enterprise and systems management and interoperability. For information about the DMTF, see DMTF.

18 This version supersedes version 1.2 and its errata versions. For a list of the changes, see ANNEX E (informative) change log.

19 1.1 Acknowledgments

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

21 **Contributors:**

- Richelle Ahlvers — Broadcom Inc.
- Jeff Andersen — Google
- Lee Ballard — Dell Technologies
- Steven Bellock — NVIDIA Corporation
- Heng Cai — Alibaba Group
- Patrick Caporale — Lenovo
- Yu-Yuan Chen — Intel Corporation
- Andrew Draper — Intel Corporation
- Nigel Edwards — Hewlett Packard Enterprise
- Daniil Egranov — Arm Limited
- Philip Hawkes — Qualcomm Inc.
- Brett Henning — Broadcom Inc.
- Jeff Hilland — Hewlett Packard Enterprise
- Yi Hou — Microchip
- Guerney Hunt — IBM
- Yuval Itkin — NVIDIA Corporation
- Theo Koulouris — Hewlett Packard Enterprise
- Raghupathy Krishnamurthy — NVIDIA Corporation
- Benjamin Lei — Lenovo
- Luis Luciani — Hewlett Packard Enterprise
- Masoud Manoo — Lenovo
- Donald Matthews — Advanced Micro Devices, Inc.
- Mahesh Natu — Intel Corporation
- Chandra Nelogal — Dell Technologies
- Edward Newman — Hewlett Packard Enterprise

- Alexander Novitskiy — Intel Corporation
- Jim Panian — Qualcomm Inc.
- Scott Phuong — Cisco Systems Inc., Axiado Corporation
- Jeffrey Plank — Microchip
- Viswanath Ponnuru — Dell Technologies
- Lohith Rangappa — Marvell Technology, Inc.
- Xiaoyu Ruan — Intel Corporation
- Nitin Sarangdhar — Intel Corporation
- Vidya Satyamsetti — Google
- Hemal Shah — Broadcom Inc.
- Yoni Shternhell — Western Digital Technologies, Inc.
- Srikanth Varadarajan — Intel Corporation
- Peng Xiao — Alibaba Group
- Qing Yang — Alibaba Group
- Jiewen Yao — Intel Corporation
- Wilson Young — Solidigm

22

2 Introduction

23 The *Security Protocol and Data Model (SPDM) Specification* defines *messages*, data objects, and sequences for performing message exchanges over a variety of transport and physical media. The description of message exchanges includes *authentication* and provisioning of hardware identities, measurement for firmware identities, session key exchange protocols to enable confidentiality with integrity-protected data communication, and other related capabilities. The SPDM enables efficient access to low-level security capabilities and operations. Other mechanisms, including non-DMTF-defined mechanisms, can use the SPDM.

24 To cope with the potential threats of emergent quantum computers, the SPDM supports both traditional and post-quantum cryptography (PQC) algorithms for the signature and authenticated key exchange protocols.

25

2.1 Advice

26 The authors recommend readers visit tutorial and education materials under Security Protocols and Data Models and Platform Management Communications Infrastructure (PMCI) on the DMTF website prior to or during the reading of this specification to help understand this specification.

27

2.2 Conventions

28 The following conventions apply to all SPDM specifications.

29

2.2.1 Document conventions

- Document titles appear in *italics*.
- The first occurrence of each important term appears in *italics* with a link to its definition.
- ABNF rules appear in a monospaced font.

30

2.2.2 Reserved and unassigned values

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

32 Unless otherwise specified, field values marked as Reserved shall be written as zero (`0`), ignored when read, not modified, and not interpreted as an error if not zero, and used in transcript hash calculations as is.

33

2.2.3 Byte ordering

34 Unless otherwise specified, for all SPDM specifications *byte* ordering of multi-byte numeric fields or multi-byte bit

fields is *little endian* (that is, the lowest byte offset holds the least significant byte, and higher offsets hold the more significant bytes).

35 #### 2.2.3.1 Hash byte order

36 For fields or values containing a digest or hash, SPDM preserves the byte order of the digest as the specification of a given hash algorithm defines. SPDM views these digests, simply, as a string of octets where the first byte is the leftmost byte of the digest, the second byte is the second leftmost byte, the third byte is the third leftmost byte, and this pattern continues until the last byte of the digest. Thus, the byte order for SPDM digests or hashes is: the first byte is placed at the lowest offset in the field or value, the second byte is placed at the second lowest offset, the third byte is placed at the third lowest offset in the field or value and this pattern continues until the last byte.

37 For example, in FIPS 180-4, a SHA 256 hash is the concatenation of eight 32-bit words where each word is in *big endian* order, but the order of words does not have any endianness associated with it. SPDM simply views this 256-bit digest as a string of octets that is 32 bytes in size where the first byte is the value at H_0[31:24] of the final digest, the second byte is the value at H_0[23:16], the third byte is the value at H_0[15:8], the fourth byte is the value at H_0[7:0], the fifth bytes is the value at H_1[31:24], and this pattern continues until the last byte, which is the value at H_7[7:0], where the FIPS 180-4 specification defines H_0, H_1, and H_7.

38 #### 2.2.3.2 Encoded ASN.1 byte order

39 For fields or values containing DER, CER, or BER encoded data, SPDM preserves the byte order as described in X.690 specification. SPDM views a DER, CER, or BER encoded data as simply a string of octets where the first byte is the leftmost byte of Figure 1 or Figure 2 in the X.690 specification, the second byte is the second leftmost byte, the third byte is the third leftmost byte, and this pattern continues until the last byte. The first byte is also called either the Identifier octet or the Leading identifier octet. The X.690 specification defines Figure 1, Figure 2, and identifier octets. When populating a DER, CER, or BER encoded data in SPDM fields, the first byte is placed in the lowest address, the second byte is placed in the second lowest offset, the third byte is placed in the third lowest offset in the field or value and this pattern continues until the last byte.

40 #### 2.2.3.3 Octet string byte order

41 A string of octets is conventionally written from left to right. Also by convention, byte zero of the octet string shall be the leftmost byte of the octet, byte 1 of the octet string shall be the second leftmost byte of the octet, and this pattern shall continue until the very last byte. When placing an octet string into an SPDM field, the i^{th} byte of the octet string shall be placed in the i^{th} offset of that field.

42 For example, if placing an octet stream consisting of "0xAA 0xCB 0x9F 0xD8" into `DMTFSpecMeasurementValue` field, then offset 0 (the lowest offset) of `DMTFSpecMeasurementValue` will contain 0xAA, offset 1 of `DMTFSpecMeasurementValue` will contain 0xCB, offset 2 of `DMTFSpecMeasurementValue` will contain 0x9F, and offset 3 of `DMTFSpecMeasurementValue` will contain 0xD8.

43 #### 2.2.3.4 Signature byte order

44 For fields or values containing a signature, SPDM attempts to preserve the byte order of the signature as the

specification of a given signature algorithm defines. Most signature specifications define a string of octets as the format of the signature, and others may explicitly state the endianness such as in the specification for Edwards-Curve Digital Signature Algorithm. Unless otherwise specified, the byte order of a signature for a given signature algorithm shall be octet string byte order.

45 #### 2.2.3.4.1 ECDSA signatures byte order

46 FIPS PUB 186-5 defines `r`, `s`, and ECDSA signature to be `(r, s)`, where `r` and `s` are just integers. For ECDSA signatures, excluding SM2, in SPDM, the signature shall be the concatenation of `r` and `s`. The size of `r` shall be the size of the selected curve. Likewise, the size of `s` shall be the size of the selected curve. See `BaseAsymAlgo` in `NEGOTIATE_ALGORITHMS` for the size of `r` and `s`. The byte order for `r` and `s` shall be big-endian order. When placing ECDSA signatures into an SPDM signature field, `r` shall come first, followed by `s`.

47 #### 2.2.3.4.2 SM2 signatures byte order

48 GB/T 32918.2-2016 defines `r` and `s` and SM2 signatures to be `(r, s)`, where `r` and `s` are just integers. The size of `r` and `s` shall each be 32 bytes. To form an SM2 signature, `r` and `s` shall be converted to an octet stream according to GB/T 32918.2-2016 and GB/T 32918.1-2016 with a target length of 32 bytes. Let the resulting octet string of `r` and `s` be called `SM2_R` and `SM2_S` respectively. The final SM2 signature shall be the concatenation of `SM2_R` and `SM2_S`. When placing SM2 signatures into an SPDM signature field, the SM2 signature byte order shall be octet string byte order.

49 ## 2.2.4 Sizes and lengths

50 Unless otherwise specified, all sizes and lengths are in units of bytes.

51 ## 2.2.5 SPDM data type conventions

52 ### 2.2.5.1 SPDM data types

53 Table 1 — SPDM data types lists the abbreviations and descriptions for common data types that SPDM message fields and data structure definitions use. These definitions follow DSP0240.

54 **Table 1 — SPDM data types**

Data type	Interpretation
`ver8`	Eight-bit encoding of the SPDM version number. Version encoding defines the encoding of the version number.
`bitfield8`	Byte with 8-bit fields.
`bitfield16`	Two-byte word with 16-bit fields.

55 #### 2.2.5.2 Integers

56 Unless noted otherwise, integers shall be unsigned.

57 ### 2.2.6 Version encoding

58 The `SPDMVersion` field represents the version of the specification through a combination of *Major* and *Minor* nibbles, encoded as follows:

Version	Matches	Incremented when
Major	Major version field in the `SPDMVersion` field in the SPDM message header.	Protocol modification breaks backward compatibility.
Minor	Minor version field in the `SPDMVersion` field in the SPDM message header.	Protocol modification maintains backward compatibility.

59 EXAMPLE:

60 Version 3.7 → `0x37`

61 Version 1.0 → `0x10`

62 Version 1.2 → `0x12`

63 An *endpoint* that supports Version 1.2 can interoperate with an older endpoint that supports Version 1.0 or other previous minor versions. Whether an endpoint supports inter-operation with previous minor versions of the SPDM specification is an implementation-specific decision.

64 An endpoint that supports Version 1.2 only and an endpoint that supports Version 3.7 only are not interoperable and shall not attempt to communicate beyond `GET_VERSION` .

65 This specification considers two minor versions to be interoperable when it is possible for an implementation that is conformant to a higher minor version number to also communicate with an implementation that is conformant to a lower minor version number with minimal differences in operation. In such a case, the following rules apply:

- Both endpoints shall use the same lower version number in the `SPDMVersion` field for all messages.
- Functionality shall be limited to what the lower minor version of the SPDM specification defines.
- Computations and other operations between different minor versions of the Secured Messages using SPDM specification should remain the same, unless security issues of lower minor versions are fixed in higher minor versions and the fixes require change in computations or other operations. These differences are dependent on the value in the `SPDMVersion` field in the message.
- In a newer minor version of the SPDM specification, a given message can be longer, bit fields and enumerations can contain new values, and reserved fields can gain functionality. Existing numeric and bit fields retain their existing definitions.
- Errata versions (indicated by a non-zero value in the `UpdateVersionNumber` field for the `GET_VERSION` request

and `VERSION` response messages) clarify existing behaviors in the SPDM specification. They maintain bitwise compatibility with the base version, except as required to fix security vulnerabilities or to correct mistakes from the base version.

66 For details on the version agreement process, see GET_VERSION request and VERSION response messages. The detailed version encoding that the `VERSION` response message returns contains an additional byte that indicates specification bug fixes or development versions. See Table 9 — Successful VERSION response message format.

67 2.2.7 Notations

68 SPDM specifications use the following notations:

Notation	Description
`Concatenate()`	The concatenation function `Concatenate(a, b, ..., z)` , where the first entry occupies the least-significant bits and the last entry occupies the most-significant bits.
`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 (`[LSb]`) offset = 0.
`[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`	Hexadecimal, as indicated by the leading `0x` .
`N+`	Variable-length byte range that starts at byte offset N.
`{ Payload }`	Used mostly in figures, this notation indicates that the payload specified in the enclosing curly brackets is encrypted and/or authenticated by the keys derived from one or more major secrets. The specific secret used is described throughout this specification. For example, `{ HEARTBEAT }` shows that the Heartbeat message is encrypted and/or authenticated by the keys derived from one or more major secrets.

Notation	Description
{ Payload }::[[S_X]]	Used mostly in figures, this notation indicates that the payload specified in the enclosing curly brackets is encrypted and/or authenticated by the keys derived from major Secret X. For example, { HEARTBEAT }::[[S_2]] shows that the Heartbeat message is encrypted and/or authenticated by the keys derived from major secret S_2 .
[${message_name}] . ${field_name}	Used to indicate a field in a message. • ${message_name} is the name of the request or response message. • ${field_name} is the name of the field in the request or response message. An asterisk (*) instead of a field name means all fields in that message except for any conditional fields that are empty (as for example KEY_EXCHANGE . OpaqueData).

69 **2.2.8 Text or string encoding**

70 When a value is indicated as a text or string data type, the encoding for the text or string shall be an array of contiguous *bytes* whose values are ordered. The first byte of the array resides at the lowest offset, and the last byte of the array is at the highest offset. The order of characters in the array shall be such that the leftmost character of the string is placed at the first byte in the array, the second leftmost character is placed in the second byte, and so forth until the last character is placed in the last byte.

71 Each byte in the array shall be the numeric value that represents that character, as ASCII — ISO/IEC 646:1991 defines.

72 Table 2 — "spdm" encoding example shows an encoding example of the string "spdm":

73 **Table 2 — "spdm" encoding example**

Offset	Character	Value
0	s	0x73
1	p	0x70
2	d	0x64
3	m	0x6D

74 **2.2.9 Deprecated material**

75 Deprecated material is not recommended for use in new development efforts. Existing and new implementations can use this material, but they shall move to the favored approach as soon as possible. Implementations can implement

any deprecated elements as required by this document to achieve backward compatibility. Although implementations can use deprecated elements, they are directed to use the favored elements instead.

76 The following typographical convention indicates deprecated material:

---

77 DEPRECATED

78 Deprecated material appears here.

79 DEPRECATED

---

80 In places where this typographical convention cannot be used (for example, in tables or figures), the "DEPRECATED" label is used alone.

81 2.2.10 Other conventions

82 Unless otherwise specified, all figures are informative.

83 3 Scope

84 This specification describes how to use messages, data objects, and sequences to exchange messages between two devices over a variety of transports and physical media. This specification contains the message exchanges, sequence diagrams, message formats, and other relevant semantics for such message exchanges, including authentication of hardware identities and firmware measurements.

85 Other specifications define the mapping of these messages to different transports and physical media. This specification provides information to enable security policy enforcement but does not specify individual policy decisions.

86 # 4 Normative references

87 The following documents are indispensable for the application of this specification. For dated or versioned references, only the edition cited, including any corrigenda or DMTF update versions, applies. For references without date or version, the latest published edition of the referenced document (including any corrigenda or DMTF update versions) applies.

- *ISO/IEC Directives, Part 2, Principles and rules for the structure and drafting of ISO and IEC documents - 2021 (9th edition)*
- DMTF DSP0004, *Common Information Model (CIM) Metamodel*, https://www.dmtf.org/sites/default/files/standards/documents/DSP0004_3.0.1.pdf
- DMTF DSP0223, *Generic Operations*, https://www.dmtf.org/sites/default/files/standards/documents/DSP0223_1.0.1.pdf
- DMTF DSP0236, *MCTP Base Specification 1.3.0*, https://dmtf.org/sites/default/files/standards/documents/DSP0236_1.3.0.pdf
- DMTF DSP0239, *MCTP IDs and Codes 1.6.0*, https://www.dmtf.org/sites/default/files/standards/documents/DSP0239_1.6.0.pdf
- DMTF DSP0240, *Platform Level Data Model (PLDM) Base Specification*, https://www.dmtf.org/sites/default/files/standards/documents/DSP0240_1.0.0.pdf
- DMTF DSP0275, *Security Protocol and Data Model (SPDM) over MCTP Binding Specification*, https://www.dmtf.org/dsp/DSP0275
- DMTF DSP1001, *Management Profile Usage Guide*, https://www.dmtf.org/sites/default/files/standards/documents/DSP1001_1.2.0.pdf
- IETF RFC 9147, *The Datagram Transport Layer Security (DTLS) Protocol Version 1.3*, April 2022
- IETF RFC 2986, *PKCS #10: Certification Request Syntax Specification*, November 2000
- IETF RFC 4716, *The Secure Shell (SSH) Public Key File Format*, November 2006
- IETF RFC 5234, *Augmented BNF for Syntax Specifications: ABNF*, January 2008
- IETF RFC 5280, *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*, May 2008
- IETF RFC 7250, *Using Raw Public Keys in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)*, June 2014
- IETF RFC 7919, *Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for Transport Layer Security (TLS)*, August 2016
- IETF RFC 8017, *PKCS #1: RSA Cryptography Specifications Version 2.2*, November, 2016
- IETF RFC 8032, *Edwards-Curve Digital Signature Algorithm (EdDSA)*, January 2017
- IETF RFC 8446, *The Transport Layer Security (TLS) Protocol Version 1.3*, August 2018
- *USB Authentication Specification Rev 1.0 with ECN and Errata through January 7, 2019*
- *TCG Algorithm Registry, Family "2.0", Level 00 Revision 01.32*, June 25, 2020
- NIST Special Publication 800-38D, *Recommendation for Block Cipher Modes of Operation: Galois/Counter*

Mode (GCM) and GMAC, November 2007

- IETF RFC 8439, *ChaCha20 and Poly1305 for IETF Protocols*, June 2018
- IETF RFC 8998, *ShangMi (SM) Cipher Suites for TLS 1.3*, March 2021
- GB/T 32918.1-2016, *Information security technology—Public key cryptographic algorithm SM2 based on elliptic curves—Part 1: General*, August 2016
- GB/T 32918.2-2016, *Information security technology—Public key cryptographic algorithm SM2 based on elliptic curves—Part 2: Digital signature algorithm*, August 2016
- GB/T 32918.3-2016, *Information security technology—Public key cryptographic algorithm SM2 based on elliptic curves—Part 3: Key exchange protocol*, August 2016
- GB/T 32918.4-2016, *Information security technology—Public key cryptographic algorithm SM2 based on elliptic curves—Part 4: Public key encryption algorithm*, August 2016
- GB/T 32918.5-2016, *Information security technology—Public key cryptographic algorithm SM2 based on elliptic curves—Part 5: Parameter definition*, August 2016
- GB/T 32905-2016, *Information security technology—SM3 cryptographic hash algorithm*, August 2016
- GB/T 32907-2016, *Information security technology—SM4 block cipher algorithm*, August 2016
- **ASN.1 — ISO-822-1-4, DER — ISO-8825-1**
  - ITU-T X.680, X.681, X.682, X.683, X.690, 08/2015
- **ASCII — ISO/IEC 646:1991**, 09/1991
- **ECDSA**
  - Section 6, The Elliptic Curve Digital Signature Algorithm (ECDSA) in FIPS PUB 186-5 Digital Signature Standard (DSS)
  - NIST SP 800-186 Recommendations for Discrete Logarithm-based Cryptography: Elliptic Curve Domain Parameters
  - IETF RFC 6979, Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA), August 2013
- **SHA2-256**, **SHA2-384**, and **SHA2-512**
  - FIPS PUB 180-4 Secure Hash Standard (SHS)
- **SHA3-256**, **SHA3-384**, and **SHA3-512**
  - FIPS PUB 202 SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions
- **ML-KEM**
  - FIPS PUB 203 Module-Lattice-Based Key-Encapsulation Mechanism Standard
- **ML-DSA**
  - FIPS PUB 204 Module-Lattice-Based Digital Signature Standard
- **SLH-DSA**
  - FIPS PUB 205 Stateless Hash-Based Digital Signature Standard

88
5 Terms and definitions

89 In this document, some terms have a specific meaning beyond the normal English meaning. This clause defines those terms.

90 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.

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

92 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, and annexes labeled "(informative)" do not contain normative content. Notes and examples are always informative elements.

93 The terms that DSP0004, DSP0223, DSP0236, DSP0239, DSP0275, and DSP1001 define also apply to this document.

94 This specification uses these terms:

Term	Definition
alias certificate	Certificate that is dynamically generated by the *component* or component firmware.
application data	Data that is specific to the application and whose definition and format is outside the scope of this specification. Application data usually exists at the application layer, which is, in general, the layer above SPDM and the transport layer. Examples of data that could be application data include: messages carried as DMTF MCTP payloads; Internet traffic; PCIe transaction layer packets (TLPs); camera images and video (MIPI CSI-2 packets); video display stream (MIPI DSI-2 packets); and touchscreen data (MIPI I3C Touch).
authentication initiator	Endpoint that initiates the authentication process by challenging another endpoint.
authentication	Process of determining whether an entity is who or what it claims to be.
byte	Eight-bit quantity. Also known as an *octet*.
certificate authority (CA)	Trusted entity that issues certificates.
certificate chain	Typically a series of two or more certificates. Each certificate is signed by the preceding certificate in the chain.
certificate	Digital form of identification that provides information about an entity and certifies ownership of a particular asymmetric key-pair.

Term	Definition
component	Physical device, contained in a single package. A "component" may also refer to a functional block implemented in hardware, firmware, and/or software.
device certificate	Certificate that contains information that identifies the component. Can be a leaf certificate or an *intermediate certificate*.
device	Physical entity such as a network controller or a fan.
DMTF	Formerly known as the Distributed Management Task Force, the DMTF creates open manageability standards that span diverse emerging and traditional information technology (IT) infrastructures, including cloud, virtualization, network, servers, and storage. Member companies and alliance partners worldwide collaborate on standards to improve the interoperable management of IT.
encapsulated request	A request embedded into an `ENCAPSULATED_REQUEST` or `ENCAPSULATED_RESPONSE_ACK` response message to allow the Responder to issue a request to a Requester. See GET_ENCAPSULATED_REQUEST request and ENCAPSULATED_REQUEST response messages.
generic certificate	A certificate, for use in certificate slots 1 or greater, that has minimal SPDM requirements to allow for numerous use cases that the vendor, standards body, or user defines.
endpoint	Logical entity that communicates with other endpoints over one or more transport protocols.
event notifier	An SPDM endpoint that is capable of sending asynchronous notifications using SPDM event mechanisms. See Event mechanism.
event recipient	An SPDM endpoint that is capable of receiving asynchronous notifications using SPDM event mechanisms. See Event mechanism.
hardware identity	A value that represents a unique instance of an endpoint. See Identity authentication.
intermediate certificate	Certificate that is neither a root certificate nor a leaf certificate.
invasive debug mode	A device mode that enables debug access that might expose or allow modification of firmware, hardware, or settings that can access (read or write) security keys, states, and contexts of the device. A device should not be trusted when it is operating in this mode.
large SPDM message	An SPDM message that is greater than the `DataTransferSize` of the receiving SPDM endpoint or greater than the transmit buffer size of the sending SPDM endpoint.
large SPDM request message	A large SPDM message that is an SPDM request.
large SPDM response message	A large SPDM message that is an SPDM response.
leaf certificate	Last certificate in a certificate chain. A leaf certificate is synonymous with an end entity certificate as RFC 5280 describes.
measurement	Representation of hardware/firmware/software or configuration data on an endpoint.
message	See SPDM message.
message body	Portion of an SPDM message that carries additional data.

Term	Definition
message transcript	The concatenation of a sequence of messages in the order in which they are sent and received by an endpoint. The final message included in the message transcript may be truncated to allow inclusion of a signature in that message which is computed over the message transcript. If an endpoint is communicating with multiple peer endpoints concurrently, the message transcripts for the peers are accumulated separately and independently.
monotonically increasing	This specification uses the term *monotonically increasing* to describe an integer field where the value of each instance of the field in a series increases from a lower starting point to a higher ending point without repeating values. For instance, a *monotonically increasing* field may contain the values 1, 3, 4, 7, and 9.
most significant byte (MSB)	Highest-order *byte* in a number consisting of multiple bytes.
Negotiated State	Set of parameters that represents the state of the communication between a corresponding pair of Requester and Responder at the successful completion of the `NEGOTIATE_ALGORITHMS` messages. These parameters may include values provided in `VERSION` , `CAPABILITIES` , and `ALGORITHMS` messages. Additionally, they may include parameters associated with the transport layer. They may include other values deemed necessary by the Requester or Responder to continue or preserve communication with each other.
nibble	Computer term for a four-bit aggregation, or half of a byte.
non-invasive debug mode	A device mode that enables debug access that does not expose or allow modification of security-critical firmware, hardware, or settings.
nonce	Number that is unpredictable to entities other than its generator. The probability of the same number occurring more than once is negligible. A nonce may be generated by combining a random number of at least 64 bits, optionally concatenated with a monotonically increasing counter of size suitable for the application.
opaque data	Opaque data fields transfer data that is outside the scope of this specification. The semantics and usage of this data are implementation specific and are also outside the scope of this specification.
payload	Information-bearing fields of a message. These fields are separate from the transport fields and elements, such as address fields, framing bits, and checksums, that transport the message from one point to another.
physical transport binding	Specifications that define how a base messaging protocol is implemented on a particular physical transport type and medium, such as SMBus/I^2C or PCI Express™ Vendor Defined Messaging.

Term	Definition
Platform Management Component Intercommunication (PMCI)	Working group under the DMTF that defines standardized communication protocols, low-level data models, and transport definitions that support communications with and between management controllers and management devices that form a platform management subsystem within a managed computer system.
record	A unit or chunk of data that is either encrypted and/or authenticated.
Requester	Original transmitter, or source, of an SPDM request message. It is also the ultimate receiver, or destination, of an SPDM response message. A Requester is the sender of the `GET_VERSION` request and remains the requester for the remainder of that connection.
Reset	This term is used to denote a Reset or restart of a device that runs the Requester or Responder code, which typically leads to the loss of all volatile state on the device.
Responder	Ultimate receiver, or destination, of an SPDM request message. It is also the original transmitter, or source of an SPDM response message.
root certificate	First certificate in a certificate chain, which acts as the trust anchor and is typically self-signed.
secure session	Provides either encryption or message authentication or both for communicating data over a transport.
Security Protocols and Data Models (SPDM) WG	Working group under the DMTF that defines standards to enable security for platforms, whether for the control plane, data plane, or other infrastructure.
sequentially decreasing	This specification uses the term *sequentially decreasing* to describe an integer field where the value of each instance of the field in a series decrements from a higher starting point to a lower ending point without skipping or repeating values. For instance, a *sequentially decreasing* field may contain the values 255, 254, 253, 252, and 251.
sequentially increasing	This specification uses the term *sequentially increasing* to describe an integer field where the value of each instance of the field in a series increments from a lower starting point to a higher ending point without skipping or repeating values. For instance, a *sequentially increasing* field may contain the values 1, 2, 3, 4, and 5.
session keys	Any secrets, derived cryptographic keys, or any cryptographic information bound to a session.
Session-Secrets-Exchange	Any SPDM request and their corresponding response that initiates a session and provides initial cryptographic exchange. Examples of such requests are `KEY_EXCHANGE` and `PSK_EXCHANGE`.
Session-Secrets-Finish	This term denotes any SPDM request and its corresponding response that finalizes a session setup and provides the final exchange of cryptographic or other information before application data can be securely transmitted. Examples of such requests are `FINISH` and `PSK_FINISH`.
SPDM message payload	Portion of the message body of an SPDM message. This portion of the message is separate from those fields and elements that identify the SPDM version, the SPDM request and response codes, and the two parameters.

Term	Definition
SPDM message	Unit of communication in SPDM communications. See Generic SPDM message format.
SPDM request message	Message that is sent to an endpoint to request a specific SPDM operation. A corresponding SPDM response message acknowledges receipt of an SPDM request message.
SPDM response message	Message that is sent in response to a specific SPDM request message. This message includes a `Response Code` field that indicates whether the request completed normally.
trusted computing base (TCB)	Set of all hardware, firmware, and/or software components that are critical to its security, in the sense that bugs or vulnerabilities occurring inside the TCB might jeopardize the security properties of the entire system. By contrast, parts of a computer system outside the TCB shall not be able to misbehave in a way that would leak any more privileges than are granted to them in accordance with the security policy. Reference: https://en.wikipedia.org/wiki/Trusted_computing_base
trusted environment	An environment where the operator is assured of no unauthorized interference in operations.

95
6 Symbols and abbreviated terms

96 The abbreviations that DSP0004, DSP0223, and DSP1001 define apply to this document.

97 The following additional abbreviations are used in this document.

Abbreviation	Definition
AEAD	Authenticated Encryption with Associated Data
CA	*certificate authority*
DMTF	Formerly the *Distributed Management Task Force*
ECC	Elliptic-curve cryptography
ECDSA	Elliptic-curve Digital Signature Algorithm
KDF	Key Derivation Function
MAC	Message Authentication Code
ML-DSA	*Module-Lattice-Based Digital Signature Standard*
ML-KEM	*Module-Lattice-Based Key Encapsulation Standard*
MSB	*most significant byte*
OID	Object identifier
PMCI	*Platform Management Component Intercommunication*
PQC	Post-Quantum Cryptography
RMA	Return Merchandise Authorization
RSA	Rivest–Shamir–Adleman
SLH-DSA	*Stateless Hash-Based Digital Signature Standard*
SPDM	Security Protocol and Data Model
SVH	Standards Body or Vendor Defined Header
TCB	*trusted computing base*
VCA	Version-Capabilities-Algorithms

98 7 SPDM message exchanges

99 The message exchanges that this specification defines are between two endpoints and are performed and exchanged through sending and receiving of *SPDM* messages that *SPDM messages* defines. The SPDM message exchanges are defined in a generic fashion that allows the messages to be communicated across different physical mediums and over different transport protocols.

100 The specification-defined message exchanges enable Requesters to:

- Discover and negotiate the security capabilities of a Responder.
- Authenticate or provision an identity of a Responder.
- Retrieve the measurements of a Responder.
- Securely establish cryptographic *session keys* to construct a secure communication channel for the transmission or reception of *application data*.
- Receive notifications of selectable events when certain scenarios transpire.

101 These message exchange capabilities are built on top of well-known and established security practices across the computing industry. The following clauses provide a brief overview of each message exchange capability. Some message exchange capabilities are based on the security model that the *USB Authentication Specification Rev 1.0 with ECN and Errata through January 7, 2019* defines.

102 7.1 Security capability discovery and negotiation

103 This specification defines a mechanism for a Requester to discover the security capabilities of a Responder. For example, an endpoint could support multiple cryptographic hash functions that this specification defines. Furthermore, the specification defines a mechanism for a Requester and Responder to select a common set of cryptographic algorithms to use for all subsequent message exchanges before another negotiation is initiated by the Requester, if an overlapping set of cryptographic algorithms exists that both endpoints support.

104 7.2 Identity authentication

105 In this specification, the authenticity of a Responder is determined by digital signatures using well-established techniques based on public key cryptography. A Responder proves its identity by generating digital signatures using a private key, and the signatures can be cryptographically verified by the Requester using the public key associated with that private key.

106 At a high level, the authentication of the identity of a Responder involves these processes:

- Identity provisioning
- Runtime authentication

107 ## 7.2.1 Identity provisioning

108 Identity provisioning is the process that device vendors follow during or after hardware manufacturing to equip a device with a secure identifier. In the context of this specification, this secure identifier consists of an asymmetric key pair and, optionally, a certificate to bind the key pair to a particular instance of a device and associate it with additional metadata. The specifics of key generation and provisioning are outside the scope of this specification. However, as the security of the SPDM protocol depends on device identities that cannot be easily modified, removed, or copied, it is strongly recommended that identity keys are unique per device and generated using cryptographically strong random seeds.

109 ### 7.2.1.1 Certificate models

110 If trust in a device public key is established through a certificate, the certificate is typically part of a *certificate chain*. The certificate chain has a *root certificate* (`RootCert`) as its root and a *leaf certificate* as the last certificate in it. The `RootCert` is generated by a trusted root *certificate authority (CA)* and certifies the certificate containing the device public key either directly or indirectly through a number of intermediate CAs. *Authentication initiators* use the `RootCert` to verify the validity of device certificate chains.

111 If the certificate chain uses the device certificate or alias certificate model, the certificate chain should contain at least one certificate that includes hardware identity information. One means of conveying hardware identity is by use of a public key. The Hardware identity OID should be used to indicate which certificate conveys the hardware identity. Though existing deployments might not include the Hardware identity OID in a certificate, it is strongly recommended that new deployments include this information. The public/private key pair associated with a hardware identity certificate is constant on the instance of the device, regardless of the version of firmware running on the device.

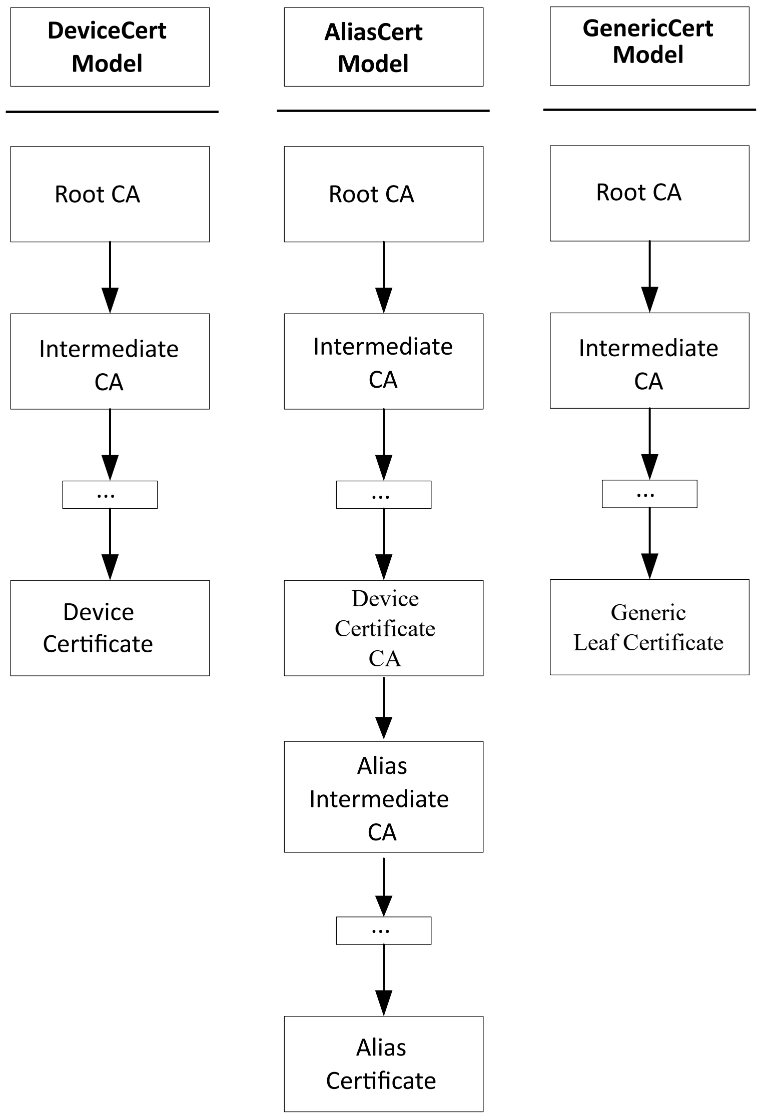
112 SPDM defines multiple overarching formats for certificate chains, referred to as certificate chain models. While the details of each certificate chain model vary, all of them follow the general format of connecting from a *root certificate* (`RootCert`) to a *leaf certificate*, possibly through one or more *intermediate certificates*.

113 A Responder can use one or more of the certificate chain models. A Requester should be capable of performing Runtime authentication on a certificate chain that conforms to any of the models.

114 Figure 1 — SPDM certificate chain models shows the SPDM certificate chain models:

115 **Figure 1 — SPDM certificate chain models**

116



117 **7.2.1.1.1 Device certificate model**

118 When the device certificate (`DeviceCert`) model is in use, the leaf certificate is a Device Certificate, which contains the public key that corresponds to the device private key. Through the certificate chain, the root CA indirectly endorses the device public key in the Device Certificate. In this model, the Device Certificate should contain the Hardware identity OID.

119 **7.2.1.1.2 Alias certificate model**

120 When the alias certificate (`AliasCert`) model is in use, the leaf certificate is an Alias Certificate, in which case there

may be one or more intermediate `AliasCert` certificates between the Device Certificate and the leaf Alias Certificate. In the `AliasCert` model, the device private key signs the next level Alias Certificate, and then the private key associated with the public key in each Alias Intermediate CA signs the Alias Certificate below it. When the `AliasCert` model is in use, the Device Certificate is referred to as a Device Certificate CA, indicating that the certificate both contains device hardware identity information and functions as a certificate authority to sign an additional certificate. In this model, the Device Certificate CA should contain the Hardware identity OID.

121 A device that implements the `AliasCert` model might factor some mutable information, such as the measurement of a firmware image, into the derivation of the public/private key pairs for the intermediate and leaf alias certificates. Therefore, the asymmetric public/private key pairs for each Alias Certificate should be treated as mutable.

122 Through the certificate chain, the root CA indirectly endorses the device public key in the Device Certificate. When the `AliasCert` model is in use, the Alias Certificates are endorsed by the device private key, meaning that the Alias Certificates are also indirectly endorsed by the root CA.

123 When the `AliasCert` model is used, the device creates and endorses one or more certificates. The certificates from the root certificate to the Device Certificate are considered immutable because the Responder cannot change them, as they can only be changed through the `SET_CERTIFICATE` command or an equivalent capability. The certificates below the Device Certificate can be created on the device and are mutable certificates in that they can change when the device state changes, such as a device *reset*. The Mutable certificate OID should be used to indicate mutable certificates.

124 In addition, when the `AliasCert` model is used, one or more Alias Certificates can contain firmware identity information. Other standards bodies might define the format of the firmware identity information. Such definitions are outside the scope of this specification.

125 Note that a signature algorithm used with a mutable alias certificate can insert random data during signing, which would cause the digest of the certificate chain to change each time it is regenerated. An implementer can use a mechanism that is outside the scope of this specification to ensure that such a signature does not change between instances of `DIGESTS` and `CERTIFICATE` responses.

126 7.2.1.1.3 Generic certificate model

127 With the support of multiple asymmetric keys, the need for another certificate model arises to accommodate varying use cases that `DeviceCert` and `AliasCert` models cannot fulfill. Thus, the generic certificate model offers the greatest flexibility to the device manufacturer, a manufacturer in the supply chain, and the users of the SPDM endpoint.

128 As Figure 1 — SPDM certificate chain models illustrates, much like the other certificate models, the generic certificate model, too, is composed of a chain of certificates starting with the root and ending with the leaf. The root CA, too, either directly certifies the leaf certificate or indirectly certifies the leaf certificate (`GenericCert`) through one or more intermediate certificate authorities. In other words, this model is the most flexible (or least restrictive) of the certificate models in this specification. The main difference between this model and the other models is that SPDM shall not impose any requirements on the contents of each certificate in the chain in a generic certificate model other than the key pair and related information associated in the leaf certificate.

129 For example, in a device certificate model, the leaf certificate can contain elements that specifically identify the

device and device manufacturer, whereas the generic certificate model has no such requirement nor any concept of a device certificate.

130 As such, the generic certificate model applies to certificates in slots greater than slot 0. A model in a certificate slot in this specification is either a `DeviceCert`, `AliasCert`, or `GenericCert` model.

131 The contents and use cases for the certificates of a generic certificate model, other than the associated key pair and related information in the leaf certificate, are outside the scope of this specification. Typically, the users of the SPDM endpoint, the device manufacturer, or standards define the contents and use cases of a generic certificate model.

132 7.2.2 Raw public keys

133 Instead of using certificate chains, the vendor can provision the raw public key of the Responder to the Requester in a trusted environment; for example, during the secure manufacturing process. In this case, trust of the public key of the Responder is established without the need for a certificate-based public key infrastructure.

134 The format of the provisioned public key is outside the scope of this specification. Vendors can use proprietary formats or public key formats that other standards define, such as RFC 7250 and RFC 4716.

135 7.2.3 Runtime authentication

136 Runtime authentication is the process by which an authentication initiator, or Requester, interacts with a Responder in a running system. The authentication initiator can retrieve the certificate chains from the Responder and send a unique challenge to the Responder. The Responder uses the private key associated with the leaf certificate to sign the challenge. The authentication initiator verifies the signature by using the public key associated with the leaf certificate of the Responder and any intermediate public keys within the certificate chain by using the root certificate as the trusted anchor.

137 If the public key of the Responder was provisioned to the Requester in a trusted environment, the authentication initiator sends a unique challenge to the Responder. The Responder signs the challenge with the private key. The authentication initiator verifies the signature by using the public key of the Responder. Device identification can be handled using the GET_ENDPOINT_INFO request and ENDPOINT_INFO response messages or the transport layer (which is outside the scope of this specification).

138 7.3 Firmware and configuration measurement

139 A measurement is a representation of firmware/software or configuration data on an endpoint. A measurement is typically either a cryptographic hash value of the data or the raw data itself. The endpoint optionally binds a measurement with the endpoint identity through the use of digital signatures. This binding enables an authentication initiator to establish the identity and measurement of the firmware/software or configuration running on the endpoint.

140 7.4 Secure sessions

141 Many devices exchange data that might require protection with other devices. In this specification, this data that is

being exchanged is generically referred to as application data. The protocol of the application data usually exists at a higher layer, and as such it is outside the scope of this specification. The protocol of the application data usually allows for encrypted and/or authenticated data transfer.

142 This specification provides a method to perform a cryptographic key exchange such that the protocol of the application data can use the exchanged keys to provide a secure channel of communication by using encryption and message authentication. This cryptographic key exchange provides either Responder-only authentication or mutual authentication, both of which can be considered equivalent to Runtime authentication. For more details, see the Session clause.

143 Finally, many SPDM requests and their corresponding responses can also be afforded the same protection. For more details, see Table 6 — SPDM request and response messages validity and the SPDM request and response code issuance allowance clause.

144 Figure 2 — SPDM messaging protocol flow gives a very high-level view of when the *secure session* starts.

145 7.5 Mutual authentication overview

146 The ability of a Responder to verify the authenticity of the Requester is called mutual authentication. Several mechanisms in this specification are detailed to provide mutual authentication capabilities. The cryptographic means to verify the identity of the Requester is the same as verifying the identity of the Responder. The Identity provisioning clause discusses identity in regards to the Responder but the details also apply to the Requester.

147 In general, when this specification states requirements or recommendations for Responders in the context of identity, those same rules also apply to Requesters in the context of mutual authentication. The various clauses in this specification provide more details.

148 7.6 Multiple asymmetric key support

149 An SPDM endpoint can use more than one asymmetric key pair for a negotiated asymmetric algorithm. This enables cryptographic isolation between different use cases which potentially increases the security posture of the SPDM endpoint and its corresponding SPDM connections. For example, an SPDM Responder can choose which key-pairs to use in a `CHALLENGE` request and which key pairs to use in a `GET_MEASUREMENTS` request. The SPDM Responder permits the `CHALLENGE` and `GET_MEASUREMENTS` requests to use the same key-pair for signing operations.

150 Additionally, a Responder can allow the Requester to select the use cases to associate with each asymmetric key pair. The Responder can, also, allow the Requester to request the generation of a new key pair.

151 To facilitate the use of multiple asymmetric keys, the ability to uniquely identify each key pair is essential. To achieve this, a unique key pair number, called `KeyPairID`, identifies each asymmetric key pair. Additionally, one or more leaf certificates can bind to the same asymmetric key pair.

152 7.7 Custom environments

153 A fixed or predetermined environment is an environment where certain characteristics of the environment are fixed or

known before the SPDM endpoints communicate with each other. In many cases, these characteristics are determined even before the environment can operate such as during the design phase. An example of a such an environment is when two specific endpoints can only communicate with each other. These environments may forfeit certain SPDM features such as interoperability. However, the security posture and guarantees of these environments are outside the scope of this specification.

154 7.8 Notification overview

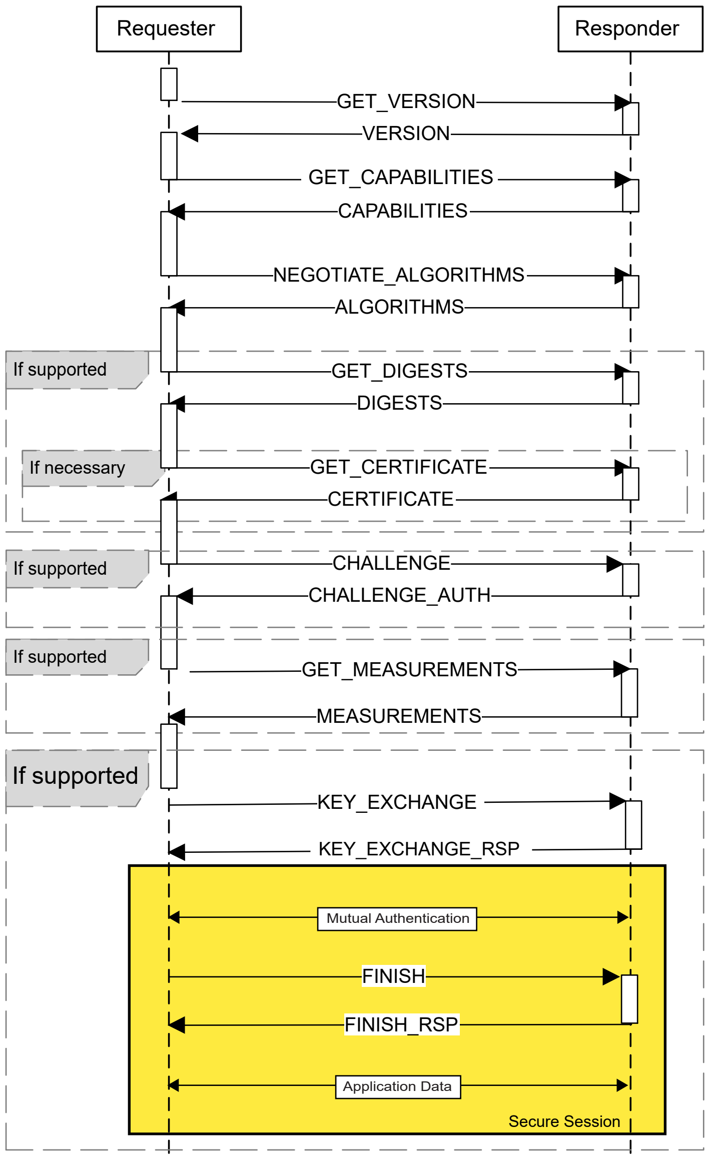
155 To aid an SPDM endpoint in enforcing its security policy requirements in an efficient, reliable, and timely manner, the SPDM event mechanism provides a method to asynchronously deliver a notification to or receive a notification from the interested SPDM endpoint. This mechanism allows an interested SPDM endpoint to choose only the event types it wants to receive. For more details, see Event mechanism.

156 8 SPDM messaging protocol

157 The SPDM messaging protocol defines a request-response messaging model between two endpoints to perform the message exchanges outlined in SPDM message exchanges. Each *SPDM request message* shall be responded to with an SPDM response message as this specification defines unless this specification states otherwise.

158 Figure 2 — SPDM messaging protocol flow is an example of a high-level request-response flow diagram for SPDM. An endpoint that acts as the *Requester* sends an SPDM request message to another endpoint that acts as the *Responder*, and the Responder returns an SPDM response message to the Requester.

159 **Figure 2 — SPDM messaging protocol flow**

160 

161 All SPDM request-response messages share a common data format that consists of a four-*byte* message header and zero or more bytes message *payload* that is message-dependent. The following clauses describe the common message format and SPDM messages' details for each of the request and response messages.

162 The Requester shall issue `GET_VERSION`, `GET_CAPABILITIES`, and `NEGOTIATE_ALGORITHMS` request messages before issuing any other request messages. The responses to `GET_VERSION`, `GET_CAPABILITIES`, and `NEGOTIATE_ALGORITHMS` can be saved by the Requester so that after Reset the Requester can skip these requests.

163 ## 8.1 SPDM connection model

164 In SPDM, communication between a pair of SPDM endpoints starts when one endpoint sends a `GET_VERSION` request to another SPDM endpoint. The SPDM endpoint that starts the communication is called the Requester. The endpoint receiving the `GET_VERSION` and providing the `VERSION` response is called a Responder. The communication between a pair of Requester and Responder is called a connection. One or more connections can exist between a Requester and Responder. Different connections might exist over the same transport or over different transports. When there are multiple connections over the same transport, the transport is responsible for providing mechanisms for SPDM endpoints to distinguish between one or more connections. When the transport does not provide such a mechanism, there shall be one connection between the Requester and Responder over that connection.

165 SPDM endpoints can be both a Requester and Responder. As a Requester, an SPDM endpoint can communicate with one or more Responders. Likewise, as a Responder, an SPDM endpoint can respond to multiple Requesters. The SPDM connection model considers each of these communications to be a separate connection. For example, a pair of SPDM endpoints can be both Requester and Responder to each other. Thus, the SPDM connection model considers this to be two separate connections.

166 Within a connection, the Requester remains the Requester for the remainder of the connection. Likewise, the Responder remains the Responder for the remainder of the connection. However, under certain scenarios allowed by SPDM, a Responder can send a request to a Requester and, likewise, a Requester might provide a response to a Responder. These cases are limited and this specification explicitly defines these cases. In such scenarios, when a Requester provides a response, the Requester shall abide by all requirements in this specification as if they are a Responder for that request. Similarly, when a Responder sends a request, the Responder shall abide by all requirements in this specification as if they are a Requester for that request.

167 Within a connection, the Requester can establish one or more secure sessions. These secure sessions are considered to be part of the same connection. Secure sessions can terminate and additional sessions can be established at any time. A `GET_VERSION` can reset the connection and all context associated with that connection including, but not limited to, information such as session keys and session IDs. However, this is not considered a termination of the connection. A connection can terminate due to external events such as a device reset or an error-handling strategy implemented on an SPDM endpoint, but such scenarios are outside the scope of this specification. Connections can be terminated using mechanisms outside the scope of this specification.

168 ## 8.2 SPDM bits-to-bytes mapping

169 All SPDM fields, regardless of size or endianness, map the highest numeric bits to the highest numerically assigned byte in sequentially decreasing order down to and including the least numerically assigned byte of that field. The following two figures illustrate this mapping.

170 Figure 3 — One-byte field bit map shows the one-byte field bit map:

171 **Figure 3 — One-byte field bit map**

172 Example:
A One-Byte Field Starting at Byte Offset 3

Byte Offset 3							
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

173 Figure 4 — Two-byte field bit map shows the two-byte field bit map:

174 **Figure 4 — Two-byte field bit map**

175 Example:
A Two-Byte Field Starting at Byte Offset 5

Byte Offset 6								Byte Offset 5							
Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

176 8.3 Generic SPDM message format

177 Table 3 — Generic SPDM message field definitions defines the fields that constitute a generic SPDM message, including the message header and payload:

178 **Table 3 — Generic SPDM message field definitions**

Byte offset	Bit offset	Size (bits)	Field	Description
0	[7:4]	4	SPDM Major Version	Shall be the major version of the SPDM Specification. An endpoint shall not communicate by using an incompatible SPDM version value. See Version encoding.
0	[3:0]	4	SPDM Minor Version	Shall be the minor version of the SPDM Specification. A specification with a given minor version extends a specification with a lower minor version as long as they share the major version. See Version encoding.

Byte offset	Bit offset	Size (bits)	Field	Description
1	[7:0]	8	Request Response Code	Shall be the request message code or response code, which Table 4 — SPDM request codes and Table 5 — SPDM response codes enumerate. `0x00` through `0x7F` represent response codes and `0x80` through `0xFF` represent request codes. In request messages, this field is considered the request code. In response messages, this field is considered the response code.
2	[7:0]	8	Param1	Shall be the first one-byte parameter. The contents of the parameter are specific to the `Request Response Code`.
3	[7:0]	8	Param2	Shall be the second one-byte parameter. The contents of the parameter are specific to the `Request Response Code`.
4	See the description.	Variable	*SPDM message payload*	Shall be zero or more bytes that are specific to the `Request Response Code`.

179 **8.3.1 SPDM version**

180 The `SPDMVersion` field, present as the first field in all SPDM messages, indicates the version of the SPDM specification that the format of an SPDM message adheres to. The format of this field shall be the same as byte 0 in Table 3 — Generic SPDM message field definitions. The value of this field shall be the same value as the version of this specification except for `GET_VERSION` and `VERSION` messages.

181 For example, if the version of this specification is 1.2, the value of `SPDMVersion` is `0x12`, which also corresponds to an `SPDM Major Version` of 1 and an `SPDM Minor Version` of 2. See Version encoding for more examples.

182 The version of this specification can be found on the title page and in the footer of the other pages in this document.

183 The `SPDMVersion` for the version of this specification shall be `0x13`.

184 The `SPDMVersionString` shall be a string formed by concatenating the major version, a period ("."), and the minor version. For example, if the version of this specification is 1.2.3, then `SPDMVersionString` is `"1.2"`.

185 ## 8.4 SPDM request codes

186 Table 4 — SPDM request codes defines the SPDM request codes. The **Implementation requirement** column indicates requirements on the Requester.

187 All SPDM-compatible implementations shall use SPDM request codes.

188 If an `ERROR` response is sent for unsupported request codes, the `ErrorCode` shall be `UnsupportedRequest`.

189 **Table 4 — SPDM request codes**

Request	Code value	Implementation requirement	Message format
GET_DIGESTS	0x81	Optional	Table 38 — GET_DIGESTS request message format
GET_CERTIFICATE	0x82	Optional	Table 42 — GET_CERTIFICATE request message format
CHALLENGE	0x83	Optional	Table 48 — CHALLENGE request message format
GET_VERSION	0x84	Required	Table 8 — GET_VERSION request message format
CHUNK_SEND	0x85	Optional	Table 101 — CHUNK_SEND request format
CHUNK_GET	0x86	Optional	Table 105 — CHUNK_GET request format
GET_ENDPOINT_INFO	0x87	Optional	Table 125 — GET_ENDPOINT_INFO request format
GET_MEASUREMENTS	0xE0	Optional	Table 49 — GET_MEASUREMENTS request message format
GET_CAPABILITIES	0xE1	Required	Table 11 — GET_CAPABILITIES request message format
GET_SUPPORTED_EVENT_TYPES	0xE2	Optional	Table 115 — GET_SUPPORTED_EVENT_TYPES request message format
NEGOTIATE_ALGORITHMS	0xE3	Required	Table 15 — NEGOTIATE_ALGORITHMS request message format
KEY_EXCHANGE	0xE4	Optional	Table 74 — KEY_EXCHANGE request message format
FINISH	0xE5	Optional	Table 77 — FINISH request message format

Request	Code value	Implementation requirement	Message format
PSK_EXCHANGE	0xE6	Optional	Table 79 — PSK_EXCHANGE request message format
PSK_FINISH	0xE7	Optional	Table 81 — PSK_FINISH request message format
HEARTBEAT	0xE8	Optional	Table 83 — HEARTBEAT request message format
KEY_UPDATE	0xE9	Optional	Table 85 — KEY_UPDATE request message format
GET_ENCAPSULATED_REQUEST	0xEA	Optional	Table 88 — GET_ENCAPSULATED_REQUEST request message format
DELIVER_ENCAPSULATED_RESPONSE	0xEB	Optional	Table 90 — DELIVER_ENCAPSULATED_RESPONSE request message format
END_SESSION	0xEC	Optional	Table 92 — END_SESSION request message format
GET_CSR	0xED	Optional	Table 95 — GET_CSR request message format
SET_CERTIFICATE	0xEE	Optional	Table 98 — SET_CERTIFICATE request message format
GET_MEASUREMENT_EXTENSION_LOG	0xEF	Optional	Table 132 — GET_MEASUREMENT_EXTENSION_LOG message format
SUBSCRIBE_EVENT_TYPES	0xF0	Optional	Table 119 — SUBSCRIBE_EVENT_TYPES request message format
SEND_EVENT	0xF1	Optional	Table 122 — SEND_EVENT request message format
GET_KEY_PAIR_INFO	0xFC	Optional	Table 107 — GET_KEY_PAIR_INFO request message format
SET_KEY_PAIR_INFO	0xFD	Optional	Table 112 — SET_KEY_PAIR_INFO request message format
VENDOR_DEFINED_REQUEST	0xFE	Optional	Table 71 — VENDOR_DEFINED_REQUEST request message format
RESPOND_IF_READY	0xFF	Required	Table 70 — RESPOND_IF_READY request message format

Request	Code value	Implementation requirement	Message format
Reserved	All other values		SPDM implementations compatible with this version shall not use the reserved request codes.

190 ## 8.5 SPDM response codes

191 The `Request Response Code` field in the SPDM response message shall specify the appropriate response code for a request. All SPDM-compatible implementations shall use Table 5 — SPDM response codes.

192 On a successful completion of an SPDM operation, the specified response message shall be returned. Upon an unsuccessful completion of an SPDM operation, the `ERROR` response message should be returned.

193 Table 5 — SPDM response codes defines the response codes for SPDM. The **Implementation requirement** column indicates requirements on the Responder.

194 **Table 5 — SPDM response codes**

Response	Value	Implementation requirement	Message format
DIGESTS	0x01	Optional	Table 39 — Successful DIGESTS response message format
CERTIFICATE	0x02	Optional	Table 44 — Successful CERTIFICATE response message format
CHALLENGE_AUTH	0x03	Optional	Table 49 — Successful CHALLENGE_AUTH response message format
VERSION	0x04	Required	Table 9 — Successful VERSION response message format
CHUNK_SEND_ACK	0x05	Optional	Table 105 — CHUNK_SEND_ACK response message format
CHUNK_RESPONSE	0x06	Optional	Table 106 — CHUNK_RESPONSE response format
ENDPOINT_INFO	0x07	Optional	Table 128 — ENDPOINT_INFO response format
MEASUREMENTS	0x60	Optional	Table 56 — Successful MEASUREMENTS response message format
CAPABILITIES	0x61	Required	Table 12 — Successful CAPABILITIES response message format

Response	Value	Implementation requirement	Message format
SUPPORTED_EVENT_TYPES	0x62	Optional	Table 116 — SUPPORTED_EVENT_TYPES response message format
ALGORITHMS	0x63	Required	Table 23 — Successful ALGORITHMS response message format
KEY_EXCHANGE_RSP	0x64	Optional	Table 76 — Successful KEY_EXCHANGE_RSP response message format
FINISH_RSP	0x65	Optional	Table 78 — Successful FINISH_RSP response message format
PSK_EXCHANGE_RSP	0x66	Optional	Table 80 — PSK_EXCHANGE_RSP response message format
PSK_FINISH_RSP	0x67	Optional	Table 82 — Successful PSK_FINISH_RSP response message format
HEARTBEAT_ACK	0x68	Optional	Table 84 — HEARTBEAT_ACK response message format
KEY_UPDATE_ACK	0x69	Optional	Table 86 — KEY_UPDATE_ACK response message format
ENCAPSULATED_REQUEST	0x6A	Optional	Table 89 — ENCAPSULATED_REQUEST response message format
ENCAPSULATED_RESPONSE_ACK	0x6B	Optional	Table 91 — ENCAPSULATED_RESPONSE_ACK response message format
END_SESSION_ACK	0x6C	Optional	Table 94 — END_SESSION_ACK response message format
CSR	0x6D	Optional	Table 97 — CSR response message format
SET_CERTIFICATE_RSP	0x6E	Optional	Table 100 — Successful SET_CERTIFICATE_RSP response message format
MEASUREMENT_EXTENSION_LOG	0x6F	Optional	Table 133 — Successful MEASUREMENT_EXTENSION_LOG message format

Response	Value	Implementation requirement	Message format
SUBSCRIBE_EVENT_TYPES_ACK	0x70	Optional	Table 120 — SUBSCRIBE_EVENT_TYPES_ACK response message format
EVENT_ACK	0x71	Optional	Table 124 — EVENT_ACK response message format
KEY_PAIR_INFO	0x7C	Optional	Table 108 — KEY_PAIR_INFO response message format
SET_KEY_PAIR_INFO_ACK	0x7D	Optional	Table 114 — SET_KEY_PAIR_INFO_ACK response message format
VENDOR_DEFINED_RESPONSE	0x7E	Optional	Table 72 — VENDOR_DEFINED_RESPONSE response message format
ERROR	0x7F	Required	Table 62 — ERROR response message format
Reserved	All other values		SPDM implementations compatible with this version shall not use the reserved response codes.

195

8.6 SPDM request and response code issuance allowance

196 Table 6 — SPDM request and response messages validity describes the conditions under which a request and response can be issued.

197 The **Session** column describes whether the respective request and response can be sent in a session. If the value is *"Allowed"*, the issuer of the request and response shall be able to send it in a secure session, thereby affording them the protection of a secure session. If the **Session** column value is *"Prohibited"*, the issuer shall be prohibited from sending the respective request and response in a secure session.

198 The **Outside of a session** column indicates which requests and responses are allowed to be sent free and independent of a session, thereby lacking the protection of a secure session. An *"Allowed"* in this column indicates that the respective request and response shall be able to be sent outside the context of a secure session. Likewise, a *"Prohibited"* in this column shall prohibit the issuer from sending the respective request or response outside the context of a session.

199 A request and its corresponding response can have an *"Allowed"* value in both the **Session** and **Outside of a session** columns, in which case they can be sent and received in both scenarios but might have additional restrictions. For details, see the respective request and response clauses.

200 A request and its corresponding response that has an *"Allowed"* value in the **Session** and *"Prohibited"* in the **Outside of a session** columns are commands used by the session. These commands only operate on the session

that they were sent under, which is outside the scope of this specification. The session ID is implicit from the session used to transmit the commands.

201 Finally, the **Session phases** column describes which phases of a session the respective request and response shall be issued when they are allowed to be issued in a session.

202 If, during the session handshake phase, an unexpected request is received using a valid session ID, the Responder shall either send an `ERROR` message in the session with `ErrorCode=UnexpectedRequest` or silently discard the request.

203 `Vendor-defined` shall indicate whether a VENDOR_DEFINED_REQUEST and VENDOR_DEFINED_RESPONSE is *"Allowed"* or *"Prohibited"* for use in the **Session**, **Outside of a session**, and the applicable **Session phases**.

204 For details, see the Session clause.

205 **Table 6 — SPDM request and response messages validity**

Request	Response	Outside of a session	Session	Session phases
GET_MEASUREMENTS	MEASUREMENTS	Allowed	Allowed	Application Phase
FINISH	FINISH_RSP	Conditional (**)	Allowed	Session Handshake
PSK_FINISH	PSK_FINISH_RSP	Prohibited	Allowed	Session Handshake
HEARTBEAT	HEARTBEAT_ACK	Prohibited	Allowed	Application Phase
KEY_UPDATE	KEY_UPDATE_ACK	Prohibited	Allowed	Application Phase
END_SESSION	END_SESSION_ACK	Prohibited	Allowed	Application Phase
Not Applicable	ERROR	Allowed	Allowed	All Phases
GET_ENCAPSULATED_REQUEST	ENCAPSULATED_REQUEST	Allowed	Allowed	All Phases
DELIVER_ENCAPSULATED_RESPONSE	ENCAPSULATED_RESPONSE_ACK	Allowed	Allowed	All Phases
VENDOR_DEFINED_REQUEST	VENDOR_DEFINED_RESPONSE	Vendor-defined	Vendor-defined	Vendor-defined
CHUNK_SEND	CHUNK_SEND_ACK	Allowed	Allowed	All Phases
CHUNK_GET	CHUNK_RESPONSE	Allowed	Allowed	All Phases
GET_ENDPOINT_INFO	ENDPOINT_INFO	Allowed	Allowed	Application Phase
GET_CSR	CSR	Allowed	Allowed	Application Phase
SET_CERTIFICATE	SET_CERTIFICATE_RSP	Allowed	Allowed	Application Phase
GET_DIGESTS	DIGESTS	Allowed	Allowed	Application Phase
GET_CERTIFICATE	CERTIFICATE	Allowed	Allowed	Application Phase

Request	Response	Outside of a session	Session	Session phases
GET_KEY_PAIR_INFO	KEY_PAIR_INFO	Allowed	Allowed	Application Phase
SET_KEY_PAIR_INFO	SET_KEY_PAIR_INFO_ACK	Allowed	Allowed	Application Phase
GET_MEASUREMENT_EXTENSION_LOG	MEASUREMENT_EXTENSION_LOG	Allowed	Allowed	Application Phase
GET_SUPPORTED_EVENT_TYPES	SUPPORTED_EVENT_TYPES	Prohibited	Allowed	Application Phase
SUBSCRIBE_EVENT_TYPES	SUBSCRIBE_EVENT_TYPES_ACK	Prohibited	Allowed	Application Phase
SEND_EVENT	EVENT_ACK	Prohibited	Allowed	Application Phase
RESPOND_IF_READY	Response to Original Request (*)	Allowed (*)	Allowed (*)	All Phases (*)
All others	All others	Allowed	Prohibited	Not Applicable

206 (*) See RESPOND_IF_READY request description for details (**) Prohibited when `HANDSHAKE_IN_THE_CLEAR_CAP` = `0`, Allowed when `HANDSHAKE_IN_THE_CLEAR_CAP` = `1`.

207 8.7 Concurrent SPDM message processing

208 This clause describes the specifications and requirements for handling concurrent overlapping SPDM request messages.

209 If an endpoint can act as both a Responder and Requester, it shall be able to send request messages and response messages independently.

210 8.8 Requirements for Requesters

211 A Requester shall not have multiple outstanding requests to the same Responder within a connection, with the following exceptions:

- As the GET_VERSION request and VERSION response messages clause describes, a Requester can issue a `GET_VERSION` to a Responder to reset the connection at any time, even if the Requester has existing outstanding requests to the same Responder.
- In the large SPDM message transfer mechanism, a single large SPDM request message and a single `CHUNK_SEND` request can be outstanding at the same time.

212 An outstanding request is a request where the request message has begun transmission, the corresponding response has not been fully received, and the request is not a retry as described in Timing Requirements.

213 If the Requester has sent a request to a Responder and wants to send a subsequent request to the same Responder, then the Requester shall wait to send the subsequent request until after the Requester completes one of the following actions:

- Receives the response from the Responder for the outstanding request.

- Times out waiting for a response.
- Receives an indication from the transport layer that transmission of the request message failed.
- The Requester encounters an internal error or Reset.
- The Requester sends a `GET_VERSION` to reinitialize the session.

214 A Requester might send simultaneous request messages to different Responders.

215 8.9 Requirements for Responders

216 A Responder is not required to process more than one request message at a time, even across connections, with the following exceptions:

- As the GET_VERSION request and VERSION response messages clause describes, a Requester can issue a `GET_VERSION` to a Responder to reset a connection at any time, even if the Requester has existing outstanding requests to the same Responder.
- In the large SPDM message transfer mechanism, a single large SPDM request message and a single `CHUNK_SEND` request can be outstanding at the same time.
- Retries can be issued multiple times to the same Responder, as Timing requirements defines.

217 A Responder that is not ready to accept a new request message or process more than one outstanding request at a time from the same Requester shall either respond with an `ERROR` message of `ErrorCode=Busy` or silently discard the request message.

218 If a Responder is working on a request message from a Requester, the Responder can respond with an `ERROR` message of `ErrorCode=Busy` .

219 If a Responder enables simultaneous communications with multiple Requesters, the Responder is expected to distinguish the Requesters by using mechanisms that are outside the scope of this specification.

220 8.10 Transcript and transcript hash calculation rules

221 The transcript is a concatenation of the prescribed full messages or message fields in order. In the case where a message is transferred in chunks, only the complete message that is built by the concatenation of chunk payloads shall be added to the transcript. Consequently, the transcript hash is the hash of the transcript using the negotiated hash algorithm (`BaseHashSel` or `ExtHashSel` of `ALGORITHMS`). For messages that are encrypted, the plaintext messages are used in the transcript. Where a transcript indicates that the hash of the specified certificate chain is used, the hash of the certificate chain is calculated over the specified certificate chain, as Table 37 — Certificate chain format describes. Messages that contribute to a transcript may be optional and/or conditional and will only contribute to a transcript if issued. Such messages are identified by the text "if issued" in the transcript definition. For a given message, if it does not have the "if issued" text in the transcript definition, then it is required to be present in the transcript. When an endpoint calculates the transcript hash over a series of messages, the endpoint shall ensure both the existence and the order of the messages as specified by each transcript hash calculation rule.

222 9 Timing requirements

223 Table 7 — Timing specification for SPDM messages shows the timing specifications for Requesters and Responders.

224 If the Requester does not receive a response within **T1** or **T2** time accordingly, the Requester can retry a request message. A retry of a request message shall be a complete retransmission of the original SPDM request message. From the perspective of a Requester, a retry of a request message is the retransmission of the original SPDM request one or more times in succession directly following the transmission of the original SPDM request. From the perspective of a Responder, a retry of a request message is the reception of the same SPDM request one or more times in succession, assuming that the transport receives messages in order. Successive SPDM requests are different if the values of any bits differ between them, in which case the Responder will process them differently.

225 If the transport is not reliable, then the Responder should support retry by identifying whether a received request is a retried one or a new one. If the Responder supports retry, then the response to a retried request shall be identical to the original response. If the transport is reliable, then the Responder may support retry.

226 The Responder shall not retry SPDM response messages. It is understood that the transport protocol(s) can retry, but this is outside the scope of this specification.

227 9.1 Timing measurements

228 Unless otherwise stated, a Requester shall measure timing parameters applicable to it from the end of a successful transmission of an SPDM request to the beginning of the reception of the corresponding SPDM response. With the exception of `RDT`, a Responder shall measure timing parameters applicable to it from the end of the reception of the SPDM request to the beginning of transmission of the response. The requirement assumes that the Responder has immediate access to the transport.

229 9.2 Timing parameters

230 In Table 7 — Timing specification for SPDM messages, timing parameters are differentiated into two categories: the timing parameters for non-cryptographic operations (`T1`) and the timing parameters for cryptographic operations (`T2`). The timing parameters are differentiated in this manner to allow a Responder to request additional time for cryptographic operations. The timing parameters apply to normal conditions, and some operations may take additional time in some situations. For instance, a Responder may need additional time to process a non-cryptographic operation because of another operation in progress or some other condition. In this case, the Responder shall respond with an `ERROR` message of `ErrorCode=ResponseNotReady` to indicate that it needs more time.

231 The Responder can request time beyond `ST1` for any non-cryptographic operation other than `GET_VERSION` . Since `GET_VERSION` serves as a reset to the connection, a Requester might send `GET_VERSION` requests as quickly as allowed by `T1` until it receives a response. The Responder shall not respond to `GET_VERSION` with an `ERROR` message of `ErrorCode=ResponseNotReady` .

232 ## 9.3 Timing specification table

233 The **Ownership** column of Table 7 — Timing specification for SPDM messages specifies whether the timing parameter applies to the Responder or Requester. For *encapsulated requests*, the Requester shall comply with the timing parameters where the **Ownership** indicates a Responder.

234 **Table 7 — Timing specification for SPDM messages**

Timing parameter	Ownership	Value	Units	Description
RTT	Requester	See the description.	μs	The value shall be the worst-case total time for the complete transmission and delivery of an SPDM message round trip at the transport layer(s). The actual value for this parameter is transport- or media-specific. Both the actual value and how an endpoint obtains this value are outside the scope of this specification. A Requester shall measure this timing parameter from the end of a successful transmission of an SPDM request to the beginning of the reception of the corresponding SPDM response less `ST1` or `CT`, depending on the Request.

Timing parameter	Ownership	Value	Units	Description
ST1	Responder	100,000	µs	This value shall be the maximum amount of time the Responder has to provide a response under normal conditions to requests that do not require cryptographic processing, such as the `GET_CAPABILITIES`, `GET_VERSION`, or `NEGOTIATE_ALGORITHMS` request messages. See Table 11 — GET_CAPABILITIES request message format, Table 8 — GET_VERSION request message format, and Table 15 — NEGOTIATE_ALGORITHMS request message format.
T1	Requester	`RTT` + `ST1`	µs	This value shall be the minimum amount of time the Requester shall wait before issuing a retry for requests that do not require cryptographic processing. For details, see the `ST1` timing parameter.

Timing parameter	Ownership	Value	Units	Description
CT	Requester and Responder	$2^{\texttt{CTExponent}}$	μs	CTExponent is reported in the `GET_CAPABILITIES` request message and `CAPABILITIES` response message. This parameter is applicable to both a Responder and Requester as the **Ownership** columns shows. Specifically for a Requester, this field is applicable when the Requester provides a response that requires cryptographic processing such as in the mutual authentication portion of a `KEY_EXCHANGE` flow. When the Requester provides a response that requires cryptographic processing, the Requester shall measure timing just as a Responder would. This timing parameter shall be the maximum amount of time the endpoint has to provide any response requiring cryptographic processing under normal conditions, such as the `GET_MEASUREMENTS` or `CHALLENGE` request messages. If the Responder cannot respond within `CT` , the Responder shall respond with an `ERROR` message of `ErrorCode=ResponseNotReady` to indicate that it needs more time. See Table 11 — GET_CAPABILITIES request message format, Table 12 — Successful CAPABILITIES response message format, Table 49 — GET_MEASUREMENTS request message format, and

Timing parameter	Ownership	Value	Units	Description
				Table 48 — CHALLENGE request message format.
T2	Requester	`RTT` + `CT`	µs	This value shall be the minimum amount of time the Requester shall wait before issuing a retry for requests that require cryptographic processing. For details, see the `CT` timing parameter.
RDT	Responder	$2^{\texttt{RDTExponent}}$	µs	This value shall be the recommended additional amount of time in microseconds that the Responder needs to complete the requested cryptographic operation. When the Responder cannot complete cryptographic processing response within the `CT` time, it shall provide `RDTExponent` as part of the `ERROR` response as Table 62 — ERROR response message format shows. For details, see `ErrorCode=ResponseNotReady` in Table 64 — ResponseNotReady extended error data for the `RDTExponent` value. An SPDM Responder measures the `RDT` value from the end of the transmission of the `ERROR` message of `ErrorCode=ResponseNotReady`, to the beginning of the reception of the next `RESPOND_IF_READY` request message.

Timing parameter	Ownership	Value	Units	Description
WT	Requester	`RDT`	μs	This value shall be the amount of time that the Requester should wait before issuing the `RESPOND_IF_READY` request message as Table 70 — RESPOND_IF_READY request message format shows. The Requester shall measure this time parameter from the reception of the `ERROR` response to the transmission of the `RESPOND_IF_READY` request. The Requester can include the transmission time of the `ERROR` from the Responder to Requester as time spent waiting for `WT` to expire. For example, if a Responder indicates `WT` is two seconds and the `ERROR` response takes one second to transport to the Requester, the Requester only needs to wait an additional one second upon reception of the `ERROR` response. For details, see the `RDT` timing parameter.

Timing parameter	Ownership	Value	Units	Description
WT_{Max}	Requester	(`RDT` * `RDTM`) - `RTT`	µs	This value shall be the maximum wait time the Requester has to issue the `RESPOND_IF_READY` request message, as Table 70 — RESPOND_IF_READY request message format shows, unless the Requester issued a successful `RESPOND_IF_READY` request message, as Table 70 — RESPOND_IF_READY request message format shows, earlier. The `RESPOND_IF_READY` message follows the most recently received `ERROR` message with `ErrorCode = ResponseNotReady`, which shall specify the wait time for that cycle. The Requester shall start measuring time from the reception of the first `ERROR` message of `ErrorCode=ResponseNotReady` with the same `Token` until `WT Max` µs elapses or the corresponding Response is successfully received. After this time has passed, the Responder is allowed to drop the response. The Requester shall take into account the transmission time of the `ERROR` response, as Table 62 — ERROR response message format shows, from the Responder to Requester when calculating `WT Max`. The `RDTM` value appears in Table 64 — ResponseNotReady extended error data. The Responder should ensure that `WT Max` does not result in less than `WT` in

Timing parameter	Ownership	Value	Units	Description
				determination of `RDTM`. See `ErrorCode=ResponseNotReady` in Table 64 — ResponseNotReady extended error data.
HeartbeatPeriod	Requester and Responder	Variable	s	See the HEARTBEAT request and HEARTBEAT_ACK response clause.

235
10 SPDM messages

236 SPDM messages can be divided into the following categories that support different aspects of security exchanges between a Requester and Responder:

- Capability discovery and negotiation
- Responder identity authentication
- Measurement
- Key agreement for secure-channel establishment

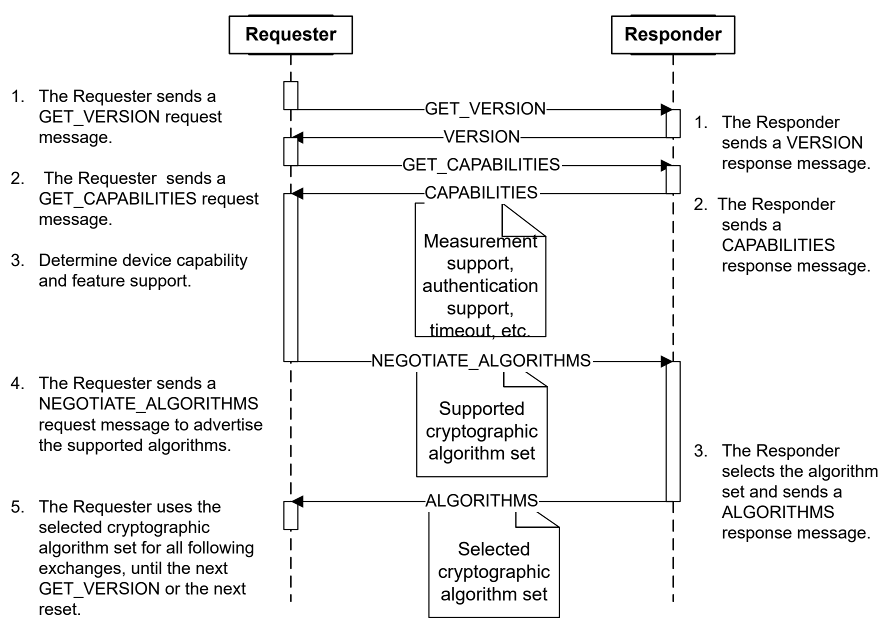
237
10.1 Capability discovery and negotiation

238 All Requesters and Responders shall support `GET_VERSION`, `GET_CAPABILITIES`, and `NEGOTIATE_ALGORITHMS`.

239 Figure 5 — Capability discovery and negotiation flow shows the high-level request-response flow and sequence for the capability discovery and negotiation:

240 **Figure 5 — Capability discovery and negotiation flow**

241



242
10.1.1 Negotiated state preamble

243 The `VCA` *(Version-Capabilities-Algorithms)* shall be the concatenation of messages `GET_VERSION`, `VERSION`,

`GET_CAPABILITIES` , `CAPABILITIES` , `NEGOTIATE_ALGORITHMS` , and `ALGORITHMS` last exchanged between the Requester and the Responder.

244 If the two endpoints do not support session key establishment with the PSK (Pre-Shared Key) option, or if the two endpoints support PSK but the negotiated capabilities and algorithms are not provisioned to both endpoints alongside the PSK, then the Requester shall issue `GET_VERSION` , `GET_CAPABILITIES` , and `NEGOTIATE_ALGORITHMS` to construct `VCA` .

245 If the Responder supports caching the negotiated state (`CACHE_CAP=1`), the Requester might not issue `GET_VERSION` , `GET_CAPABILITIES` , and `NEGOTIATE_ALGORITHMS` . In this case, the Requester and the Responder shall store the most recent `VCA` as part of the Negotiated State.

246 If the two endpoints support session key establishment with the PSK and if the negotiated capabilities and algorithms (the `C` and `A` of `VCA`) are provisioned to both endpoints alongside the PSK, then the Requester shall not issue `GET_CAPABILITIES` and `NEGOTIATE_ALGORITHMS` .

247 10.2 GET_VERSION request and VERSION response messages

248 This request message shall retrieve the SPDM version of an endpoint. Table 8 — GET_VERSION request message format shows the `GET_VERSION` request message format and Table 9 — Successful VERSION response message format shows the `VERSION` response message format.

249 In all future SPDM versions, the `GET_VERSION` and `VERSION` response messages will be backward compatible with all earlier versions.

250 The Requester shall begin the discovery process by sending a `GET_VERSION` request message with the value of the `SPDMVersion` field set to `0x10` . All Responders shall always support the `GET_VERSION` request message with major version `0x1` and provide a `VERSION` response containing all supported versions, as Table 8 — GET_VERSION request message format describes.

251 The Requester shall consult the `VERSION` response to select a common supported version, which should be the latest supported common version. The Requester shall use the selected version in all future communication of other requests. A Requester shall not issue other requests until it receives a successful `VERSION` response and identifies a common version that both sides support. A Responder shall not respond to the `GET_VERSION` request message with an `ERROR` message except for `ErrorCode` s specified in this clause. The selected version shall be the version in the `SPDMVersion` field of the Request (other than `GET_VERSION`) immediately following the `GET_VERSION` request. If the Requester uses a version other than the selected version in a Request, the Responder should either return an `ERROR` message of `ErrorCode=VersionMismatch` or silently discard the Request.

252 A Requester can issue a `GET_VERSION` request message to a Responder at any time, which serves as an exception to Requirements for Requesters to allow for scenarios where a Requester is required to restart the protocol due to an internal error or Reset.

253 After receiving a valid `GET_VERSION` request, the Responder shall invalidate state and data associated with all previous requests from the same Requester. All active sessions between the Requester and the Responder are terminated, and information (such as session keys and session IDs) for those sessions should not be used anymore. Additionally, this message shall clear the previously *Negotiated State*, if any, in both the Requester and its

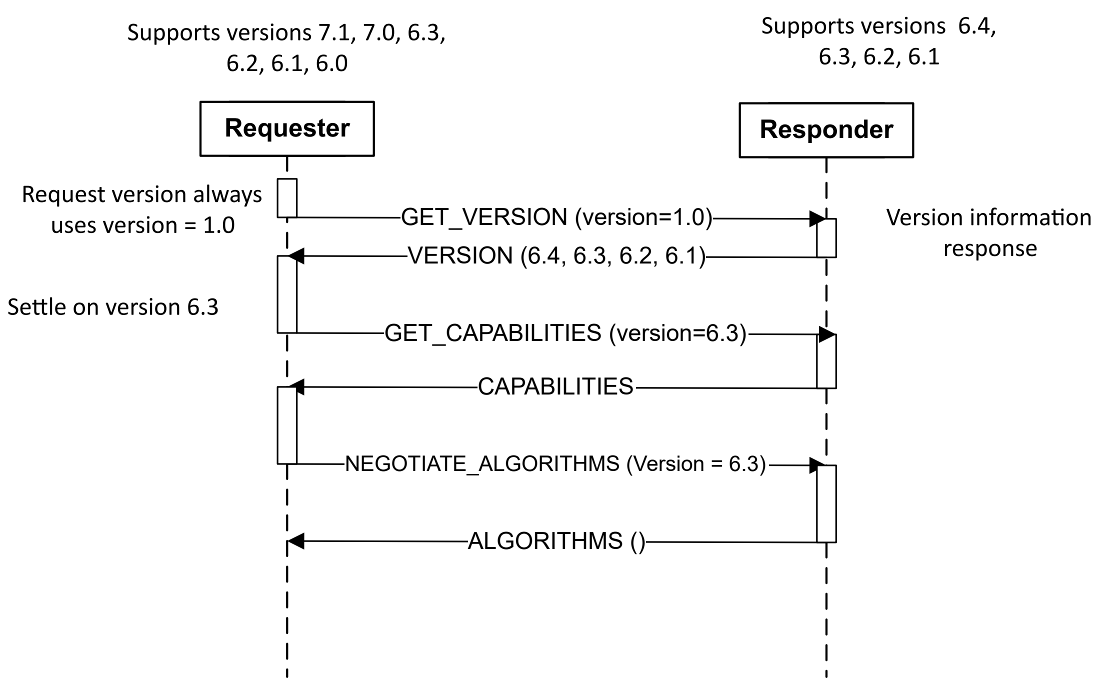
corresponding Responder. An invalid `GET_VERSION` request that results in the Responder returning an error to the Requester shall not affect the connection state. The `ERROR` message resulting from an invalid `GET_VERSION` request shall have the value of the `SPDMVersion` field set to `0x10`.

254 After sending the `VERSION` response for a `GET_VERSION` request, if the Responder completes a runtime code or configuration change for its hardware or firmware measurement and the change has taken effect, then the Responder shall either silently discard any request received outside of a session or respond with an `ERROR` message of `ErrorCode=RequestResynch` to any request received outside of a session, until a `GET_VERSION` request is received. For requests received within a session, the Responder shall follow the selected session policy that the Requester selects in Table 75 — Session policy at the time of session establishment.

255 Figure 6 — Discovering the common major version shows the process:

256 **Figure 6 — Discovering the common major version**

257



258 Table 8 — GET_VERSION request message format shows the `GET_VERSION` request message format:

259 **Table 8 — GET_VERSION request message format**

Byte offset	Field	Size (bytes)	Description
0	SPDMVersion	1	Shall be `0x10` (V1.0).
1	RequestResponseCode	1	Shall be `0x84` = `GET_VERSION`. See Table 4 — SPDM request codes.

Byte offset	Field	Size (bytes)	Description
2	Param1	1	Reserved.
3	Param2	1	Reserved.

260 Table 9 — Successful VERSION response message format shows the successful `VERSION` response message format:

261 **Table 9 — Successful VERSION response message format**

Byte offset	Field	Size (bytes)	Description
0	SPDMVersion	1	Shall be `0x10` (V1.0).
1	RequestResponseCode	1	Shall be `0x04` = `VERSION`. See Table 5 — SPDM response codes.
2	Param1	1	Reserved.
3	Param2	1	Reserved.
4	Reserved	1	Reserved.
5	VersionNumberEntryCount	1	Number of version entries present in this table (=n).
6	VersionNumberEntry1:n	2 * n	16-bit version entry. See Table 10 — VersionNumberEntry definition. Each entry should be unique.

262 Table 10 — VersionNumberEntry definition shows the `VersionNumberEntry` definition. See Version encoding for more details.

263 **Table 10 — VersionNumberEntry definition**

Bit offset	Field	Description
[15:12]	MajorVersion	Shall be the version of the specification having changes that are incompatible with one or more functions in earlier major versions of the specification.
[11:8]	MinorVersion	Shall be the version of the specification having changes that are compatible with functions in earlier minor versions of this major version specification.
[7:4]	UpdateVersionNumber	Shall be the version of the specification with editorial updates and errata fixes. Informational; ignore when checking versions for interoperability.
[3:0]	Alpha	Shall be the pre-release work-in-progress version of the specification. Because the `Alpha` value represents an in-development version of the specification, versions that share the same major and minor version numbers but have different `Alpha` versions might not be fully interoperable. Released versions shall have an `Alpha` value of zero (`0`).

264 10.3 GET_CAPABILITIES request and CAPABILITIES response messages

265 This request message shall retrieve the SPDM capabilities of an endpoint.

266 Table 11 — GET_CAPABILITIES request message format shows the `GET_CAPABILITIES` request message format.

267 Table 12 — Successful CAPABILITIES response message format shows the `CAPABILITIES` response message format.

268 Table 13 — Flag fields definitions for the Requester shows the flag fields definitions for the Requester.

269 Likewise, Table 14 — Flag fields definitions for the Responder shows the flag fields definitions for the Responder.

270 To properly support transferring of SPDM messages, the Requester and Responder shall indicate two buffer sizes:

- One for receiving a single SPDM transfer called `DataTransferSize`
- One for indicating their maximum internal buffer size for processing a single assembled received SPDM message called `MaxSPDMmsgSize`

271 Additionally, the Requester and Responder can have a transmit buffer. The transmit buffer size is not communicated to the other SPDM endpoint, but it can be less than the `DataTransferSize` of the receiving SPDM endpoint.

272 Both the Requester and Responder shall support a minimum size for both the transmit and receive buffer to successfully transfer SPDM messages. The minimum size is referred to as `MinDataTransferSize.` For this version of the specification, the `MinDataTransferSize` shall be 42. This value is the size, in bytes, of the SPDM message with the largest size from this list, assuming all fields are present:

- `GET_VERSION`
- `VERSION` assuming no versions returned contain `Alpha` versions in `VersionNumberEntry` and version entries are not duplicated.
- `GET_CAPABILITIES`
- `CAPABILITIES` with `Param1` in the `GET_CAPABILITIES` request set to 0.
- `CHUNK_SEND` using the size of the SPDM Header for the size of the `SPDMchunk` field.
- `CHUNK_SEND_ACK` using the maximum size of `ERROR` message for the size of the `ResponseToLargeRequest` field.
- `CHUNK_GET`
- `CHUNK_RESPONSE` using the size of SPDM Header for the size of the `SPDMchunk` field.
- `ERROR` using the maximum size for the `ExtendedErrorData`

273 The `GET_CAPABILITIES` request with Extended capabilities (Bit 0 of `Param1` set to a value of 1) is only allowed if both the Requester and Responder support the Large SPDM message transfer mechanism (`CHUNK_CAP=1`). If the `GET_CAPABILITIES` request sets Bit 0 of `Param1` to a value of 1, then the Responder shall use the value for `DataTransferSize` and `MaxSPDMmsgSize` from the request for the transmission of the `CAPABILITIES` response. A Responder can report that it needs to transmit the response in smaller transfers by sending an `ERROR` message of `ErrorCode=LargeResponse` . If the `GET_CAPABILITIES` request sets Bit 0 of `Param1` to a value of 1 and the Responder does not support the Large SPDM message transfer mechanism (`CHUNK_CAP=0`), the Responder shall send an `ERROR` message of `ErrorCode=InvalidRequest` .

274 **Table 11 — GET_CAPABILITIES request message format**

Byte offset	Field	Size (bytes)	Description
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	RequestResponseCode	1	Shall be `0xE1` = `GET_CAPABILITIES`. See Table 4 — SPDM request codes.
2	Param1	1	Shall be the extended capabilities to include in the response. • Bit 0. If set in the requests, the Responder shall include the Supported Algorithms Block in its `CAPABILITIES` response if it supports this extended capability. If the Requester does not support the Large SPDM message transfer mechanism (`CHUNK_CAP=0`), this bit shall be 0. • All other values reserved.
3	Param2	1	Reserved.
4	Reserved	1	Reserved.
5	CTExponent	1	Shall be exponent of base 2, which is used to calculate `CT`. See Table 7 — Timing specification for SPDM messages. The equation for `CT` shall be $2^{CTExponent}$ microseconds (µs). For example, if `CTExponent` is 10, `CT` is 2^{10} = 1024 µs.
6	Reserved	2	Reserved.
8	Flags	4	See Table 13 — Flag fields definitions for the Requester.
12	DataTransferSize	4	This field shall indicate the maximum buffer size, in bytes, of the Requester for receiving a single and complete SPDM message whose message size is less than or equal to the value in this field. The value of this field shall be equal to or greater than `MinDataTransferSize`. The `DataTransferSize` shall exclude transport headers, encryption headers, and MAC. This field helps the sender of the SPDM message know whether or not it needs to utilize the Large SPDM message transfer mechanism.

Byte offset	Field	Size (bytes)	Description
16	MaxSPDMmsgSize	4	If the Requester supports the Large SPDM message transfer mechanism, this field shall indicate the maximum size, in bytes, of the internal buffer of a Requester used to reassemble a single and complete Large SPDM message. This field shall be greater than or equal to `DataTransferSize`. This buffer size is most helpful when transferring a Large SPDM message in multiple chunks because it tells the sender whether or not there is enough memory for the fully reassembled SPDM message. If the Requester does not support the Large SPDM message transfer mechanism, this field shall be equal to the `DataTransferSize` of the Requester.

275 **Table 12 — Successful CAPABILITIES response message format**

Byte offset	Field	Size (bytes)	Description
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	RequestResponseCode	1	Shall be `0x61` = `CAPABILITIES`. See Table 5 — SPDM response codes.
2	Param1	1	Shall be the extended capabilities included in the response. • Bit 0. If the request message sets the Supported Algorithms extended capability bit and the Responder supports this extended capability, then the Responder shall set this bit in the response and shall include the Supported Algorithms Block in its `CAPABILITIES` response. If the Responder does not support this extended capability or does not support the Large SPDM message transfer mechanism (`CHUNK_CAP=0`), this bit shall be 0. • All other values reserved.
3	Param2	1	Reserved.
4	Reserved	1	Reserved.

Byte offset	Field	Size (bytes)	Description
5	CTExponent	1	Shall be the exponent of base 2, which used to calculate `CT`. See Table 7 — Timing specification for SPDM messages. The equation for `CT` shall be 2 `CTExponent` microseconds (µs). For example, if `CTExponent` is 10, `CT` is 2^{10} = 1024 µs.
6	Reserved	2	Reserved.
8	Flags	4	See Table 14 — Flag fields definitions for the Responder.
12	DataTransferSize	4	This field shall indicate the maximum buffer size, in bytes, of the Responder for receiving a single and complete SPDM message whose message size is less than or equal to the value in this field. The value of this field shall be equal to or greater than `MinDataTransferSize`. The `DataTransferSize` shall exclude transport headers, encryption headers, and MAC. This field helps the sender of the SPDM message know whether or not it needs to utilize the Large SPDM message transfer mechanism.
16	MaxSPDMmsgSize	4	If the Responder supports the Large SPDM message transfer mechanism, this field shall indicate the maximum size, in bytes, of the internal buffer of a Responder used to reassemble a single and complete Large SPDM message. This field shall be greater than or equal to `DataTransferSize`. This buffer size is most helpful when transferring a Large SPDM message in multiple chunks because it tells the sender whether or not there is enough memory for the fully reassembled SPDM message. If the Responder does not support the Large SPDM message transfer mechanism, this field shall be equal to the `DataTransferSize` of the Responder.
20	SupportedAlgorithms	AlgSize or 0	If present, this field shall be `AlgSize` in size and the format of the field shall be as described in Supported algorithms block. If Bit 0 of `Param1` does not indicate that the Supported Algorithm extended capability is included in this response, then this field shall be absent.

276 As described in other parts of this specification, a Requester or Responder can reverse roles or take on both roles for

certain SPDM messages and flows. Thus, an SPDM endpoint cannot send a Large SPDM message that exceeds the `MaxSPDMmsgSize` of the receiving SPDM endpoint. Specifically, a requesting SPDM endpoint shall not send a request that exceeds the size of `MaxSPDMmsgSize` of the responding SPDM endpoint. Likewise, a responding SPDM endpoint shall not send a response that exceeds the size of `MaxSPDMmsgSize` of the requesting SPDM endpoint. If the size of a response message exceeds the size of the `MaxSPDMmsgSize` of the requesting SPDM endpoint, the responding SPDM endpoint shall respond with an `ERROR` message of `ErrorCode=ResponseTooLarge`. If the size of a request message exceeds the size of the `MaxSPDMmsgSize` of the responding SPDM endpoint, the responding SPDM endpoint shall either respond with an `ERROR` message of `ErrorCode=RequestTooLarge` or silently discard the request. Additionally, an SPDM endpoint is expected to provide graceful error handling (for example, buffer overflow/underflow protection) in the event that it receives an SPDM message that exceeds its `MaxSPDMmsgSize`.

277 Table 13 — Flag fields definitions for the Requester shows the flag fields definitions for the Requester.

278 Unless otherwise stated, if a Requester indicates support for a capability associated with an SPDM request or response message, it means the Requester can receive the corresponding request and produce a successful response. In other words, the Requester is acting as a Responder to that SPDM request associated with that capability. For example, if a Requester sets the `CERT_CAP` bit to `1`, the Requester can receive a `GET_CERTIFICATE` request and send back a successful `CERTIFICATE` response message.

279 `AlgSize` is the size of the Supported algorithms block. If the Supported Algorithms Block is not included in the response, then the `SupportedAlgorithms` field shall be absent.

280 **Table 13 — Flag fields definitions for the Requester**

Byte offset	Bit offset	Field	Description
0	0	Reserved	Reserved.
0	1	CERT_CAP	If set, Requester shall support `DIGESTS` and `CERTIFICATE` response messages. Shall be `0b` if the Requester does not support asymmetric algorithms.
0	2	CHAL_CAP	**DEPRECATED:** If set, Requester shall support `CHALLENGE_AUTH` response message.
0	[5:3]	Reserved	Reserved.
0	6	ENCRYPT_CAP	If set, Requester shall support message encryption in a secure session. If set, when the Requester chooses to start a secure session, the Requester shall send one of the Session-Secrets-Exchange request messages supported by the Responder.

Byte offset	Bit offset	Field	Description
0	7	MAC_CAP	If set, Requester shall support message authentication in a secure session. If set, when the Requester chooses to start a secure session, the Requester shall send one of the Session-Secrets-Exchange request messages supported by the Responder. `MAC_CAP` is not the same as the `HMAC` in the `RequesterVerifyData` or `ResponderVerifyData` fields of Session-Secrets-Exchange and Session-Secrets-Finish messages.
1	0	MUT_AUTH_CAP	If set, Requester shall support mutual authentication.
1	1	KEY_EX_CAP	If set, Requester shall support `KEY_EXCHANGE` request message. If set, `ENCRYPT_CAP` or `MAC_CAP` shall be set.
1	[3:2]	PSK_CAP	Pre-Shared Key capabilities of the Requester. • `00b` . Requester shall not support Pre-Shared Key capabilities. • `01b` . Requester shall support Pre-Shared Key • `10b` and `11b` . Reserved. If supported, `ENCRYPT_CAP` or `MAC_CAP` shall be set.
1	4	ENCAP_CAP	If set, Requester shall support `GET_ENCAPSULATED_REQUEST` , `ENCAPSULATED_REQUEST` , `DELIVER_ENCAPSULATED_RESPONSE` , and `ENCAPSULATED_RESPONSE_ACK` messages. Additionally, the transport may require the Requester to support these messages. `ENCAP_CAP` was previously deprecated because Basic mutual authentication is deprecated. Deprecation is removed since some messages, such as `KEY_UPDATE` , do not require mutual authentication but still require `ENCAP_CAP` .
1	5	HBEAT_CAP	If set, Requester shall support `HEARTBEAT` messages.
1	6	KEY_UPD_CAP	If set, Requester shall support `KEY_UPDATE` messages.

Byte offset	Bit offset	Field	Description
1	7	HANDSHAKE_IN_THE_CLEAR_CAP	If set, the Requester can support a Responder that can only send and receive all SPDM messages exchanged during the Session Handshake Phase in the clear (such as without encryption and message authentication). Application data is encrypted and/or authenticated using the negotiated cryptographic algorithms as normal. Setting this bit leads to changes in the contents of certain SPDM messages, as discussed in other parts of this specification. If this bit is cleared, the Requester signals that it requires encryption and/or message authentication of SPDM messages exchanged during the Session Handshake Phase. If the Requester supports Pre-Shared Keys (`PSK_CAP` is `01b`) and does not support asymmetric key exchange (`KEY_EX_CAP` is `0b`), then this bit shall be zero. If the Requester does not support encryption and message authentication, then this bit shall be zero. In other words, this bit indicates whether `MAC_CAP` and `ENCRYPT_CAP` is involved accordingly in the handshake phase of a secure session or both encryption and message authentication capabilities are disabled in the session handshake phase of a secure session.
2	0	PUB_KEY_ID_CAP	If set, the public key of the Requester was provisioned to the Responder. The transport layer is responsible for identifying the Responder. In this case, the `CERT_CAP` and `MULTI_KEY_CAP` of the Requester shall be `0` .
2	1	CHUNK_CAP	If set, Requester shall support Large SPDM message transfer mechanism messages.
2	[5:2]	Reserved	Reserved.

Byte offset	Bit offset	Field	Description
2	[7:6]	EP_INFO_CAP	The `ENDPOINT_INFO` response capabilities of the Requester. • `00b`. The Requester does not support `ENDPOINT_INFO` response capabilities. • `01b`. The Requester supports the `ENDPOINT_INFO` response but cannot perform signature generation for this response. • `10b`. The Requester supports the `ENDPOINT_INFO` response and can generate signatures for this response. • `11b`. Reserved.
3	0	Reserved	Reserved.
3	1	EVENT_CAP	If set, the Requester is an Event Notifier. See Event mechanism for details.
3	[3:2]	MULTI_KEY_CAP	Shall be the Multiple Asymmetric Key capabilities of the Requester. • `00b`. Requester shall not support `Multiple Asymmetric Key` capabilities. • `01b`. Requester shall only support `Multiple Asymmetric Key` capabilities. • `10b`. Requester shall support `Multiple Asymmetric Key` capabilities, and Responder can use `RequesterMultiKeyConnSel` as Multiple Asymmetric Key Negotiation describes. • `11b`. Reserved. If set to `01b` or `10b`, the Requester shall support more than one key pair for at least one asymmetric algorithm for use in Requester authentication such as in mutual authentication. In the case of mutual authentication, these are the key pairs belonging to the Requester.
3	[6:4]	Reserved	Reserved.
3	7	LARGE_CERT_CAP	If set, Requester shall support using large fields in the `CERTIFICATE` response messages. Shall be `0b` if the Requester's `CERT_CAP` field is `0b`.

281 Table 14 — Flag fields definitions for the Responder shows the flag fields definitions for the Responder.

282 Unless otherwise stated, if a Responder indicates support for a capability associated with an SPDM request or response message, it means the Responder can receive the corresponding request and produce a successful

response. For example, if a Responder sets the `CERT_CAP` bit to `1`, the Responder can receive a `GET_CERTIFICATE` request and send back a successful `CERTIFICATE` response message.

283 **Table 14 — Flag fields definitions for the Responder**

Byte offset	Bit offset	Field	Description
0	0	CACHE_CAP	If set, the Responder shall support the ability to cache the *Negotiated State* across a Reset. This allows the Requester to skip reissuing the `GET_VERSION`, `GET_CAPABILITIES`, and `NEGOTIATE_ALGORITHMS` requests after a Reset. The Responder shall cache the selected cryptographic algorithms as one of the parameters of the Negotiated State. If the Requester chooses to skip issuing these requests after the Reset, the Requester shall also cache the same selected cryptographic algorithms.
0	1	CERT_CAP	If set, Responder shall support `DIGESTS` and `CERTIFICATE` response messages. Shall be `0b` if the Responder does not support asymmetric algorithms.
0	2	CHAL_CAP	If set, Responder shall support `CHALLENGE_AUTH` response message.
0	[4:3]	MEAS_CAP	`MEASUREMENTS` response capabilities of the Responder. • `00b`. The Responder shall not support `MEASUREMENTS` response capabilities. • `01b`. The Responder shall support `MEASUREMENTS` response but cannot perform signature generation for this response. • `10b`. The Responder shall support `MEASUREMENTS` response and can generate signatures for this response. • `11b`. Reserved. Note that, apart from affecting `MEASUREMENTS`, this capability also affects Param2 of `CHALLENGE`, Param1 of `KEY_EXCHANGE`, Param1 of `PSK_EXCHANGE`, and the MeasurementSummaryHash field of `KEY_EXCHANGE_RSP`, `CHALLENGE_AUTH`, and `PSK_EXCHANGE_RSP`. See the respective request and response clauses for further details.

Byte offset	Bit offset	Field	Description
0	5	MEAS_FRESH_CAP	• `0`. As part of `MEASUREMENTS` response message, the Responder may return `MEASUREMENTS` that were computed during the last Responder's Reset. • `1`. The Responder shall support recomputing all `MEASUREMENTS` without requiring a Reset and shall always return fresh `MEASUREMENTS` as part of `MEASUREMENTS` response message.
0	6	ENCRYPT_CAP	If set, Responder shall support message encryption in a secure session. If set, `PSK_CAP` or `KEY_EX_CAP` shall be set accordingly to indicate support.
0	7	MAC_CAP	If set, Responder shall support message authentication in a secure session. If set, `PSK_CAP` or `KEY_EX_CAP` shall be set accordingly to indicate support. `MAC_CAP` is not the same as the `HMAC` in the `RequesterVerifyData` or `ResponderVerifyData` fields of Session-Secrets-Exchange and Session-Secrets-Finish messages.
1	0	MUT_AUTH_CAP	If set, Responder shall support mutual authentication.
1	1	KEY_EX_CAP	If set, Responder shall support `KEY_EXCHANGE_RSP` response message. If set, `ENCRYPT_CAP` or `MAC_CAP` shall be set.
1	[3:2]	PSK_CAP	`Pre-Shared Key` capabilities of the Responder. • `00b`. Responder shall not support `Pre-Shared Key` capabilities. • `01b`. Responder shall support `Pre-Shared Key` but does not provide ResponderContext for session key derivation. • `10b`. Responder shall support `Pre-Shared Key` and provides ResponderContext for session key derivation. • `11b`. Reserved. If supported, `ENCRYPT_CAP` or `MAC_CAP` shall be set.

Byte offset	Bit offset	Field	Description
1	4	ENCAP_CAP	If set, Responder shall support `GET_ENCAPSULATED_REQUEST` , `ENCAPSULATED_REQUEST` , `DELIVER_ENCAPSULATED_RESPONSE` , and `ENCAPSULATED_RESPONSE_ACK` messages. Additionally, the transport may require the Responder to support these messages. `ENCAP_CAP` was previously deprecated because Basic mutual authentication is deprecated. Deprecation is removed since some messages, such as `KEY_UPDATE` , do not require mutual authentication but still require `ENCAP_CAP` .
1	5	HBEAT_CAP	If set, Responder shall support `HEARTBEAT` messages.
1	6	KEY_UPD_CAP	If set, Responder shall support `KEY_UPDATE` messages.
1	7	HANDSHAKE_IN_THE_CLEAR_CAP	If set, the Responder can only send and receive messages without encryption and message authentication during the Session Handshake Phase. If set, `KEY_EX_CAP` shall also be set. Setting this bit leads to changes in the contents of certain SPDM messages, as discussed in other parts of this specification. If the Responder supports Pre-Shared Keys (`PSK_CAP` is `01b`) and does not support asymmetric key exchange (`KEY_EX_CAP` is `0b`), then this bit shall be zero. If the Responder does not support encryption and message authentication, then this bit shall be zero. In other words, this bit indicates whether message authentication and/or encryption (`MAC_CAP` and `ENCRYPT_CAP`) are used in the handshake phase of a secure session.
2	0	PUB_KEY_ID_CAP	If set, the public key of the Responder was provisioned to the Requester. The transport layer is responsible for identifying the Requester. In this case, the `CERT_CAP` , `ALIAS_CERT_CAP` , and `MULTI_KEY_CAP` of the Responder shall be `0` .

Byte offset	Bit offset	Field	Description
2	1	CHUNK_CAP	If set, Responder shall support Large SPDM message transfer mechanism messages.
2	2	ALIAS_CERT_CAP	If set, the Responder shall use the `AliasCert` model. See Identity provisioning for details.
2	3	SET_CERT_CAP	If set, Responder shall support `SET_CERTIFICATE_RSP` response messages.
2	4	CSR_CAP	If set, Responder shall support `CSR` response messages. If this bit is set, `SET_CERT_CAP` shall be set.
2	5	CERT_INSTALL_RESET_CAP	If set, Responder may return an `ERROR` message of `ErrorCode=ResetRequired` to complete a certificate provisioning request. If this bit is set, `SET_CERT_CAP` shall be set and `CSR_CAP` can be set.
2	[7:6]	EP_INFO_CAP	The `ENDPOINT_INFO` response capabilities of the Responder. • `00b` . The Responder shall not support `ENDPOINT_INFO` response capabilities. • `01b` . The Responder shall support the `ENDPOINT_INFO` response but cannot perform signature generation for this response. • `10b` . The Responder shall support the `ENDPOINT_INFO` response and can generate signatures for this response. • `11b` . Reserved.
3	0	MEL_CAP	If set, Responder shall support `MEASUREMENT_EXTENSION_LOG` response message.
3	1	EVENT_CAP	If set, the Responder is an Event Notifier. See Event mechanism for details.

Byte offset	Bit offset	Field	Description
3	[3:2]	MULTI_KEY_CAP	Shall be the Multiple Asymmetric Key capabilities of the Responder. • `00b` . Responder shall not support `Multiple Asymmetric Key` capabilities. • `01b` . Responder shall only support `Multiple Asymmetric Key` capabilities. • `10b` . Responder shall support `Multiple Asymmetric Key` capabilities, and Requester can use `ResponderMultiKeyConn` as Multiple Asymmetric Key Negotiation describes. • `11b` . Reserved. If set to `01b` or `10b` , the Responder shall support more than one key pair for at least one asymmetric algorithm for the SPDM connection to use in Responder authentication.
3	4	GET_KEY_PAIR_INFO_CAP	If set, Responder shall support `KEY_PAIR_INFO` response messages. If the Responder sets `MULTI_KEY_CAP` , this bit shall also be set.
3	5	SET_KEY_PAIR_INFO_CAP	If set, Responder shall support `SET_KEY_PAIR_INFO_ACK` response message.
3	6	SET_KEY_PAIR_RESET_CAP	If set, Responder may return an `ERROR` message of `ErrorCode=ResetRequired` to complete a `SET_KEY_PAIR_INFO` request. If this bit is set, `SET_KEY_PAIR_INFO_CAP` shall be set.
3	7	LARGE_CERT_CAP	If set, Responder shall support using large fields in the `CERTIFICATE` response messages. Shall be `0b` if the Responder's `CERT_CAP` field is `0b` .

284 In the case where an SPDM implementation incorrectly returns an illegal combination of capability flags as they are defined by this specification (for example, `ENCRYPT_CAP` is set but both `KEY_EX_CAP` and `PSK_CAP` are cleared), the following guidance is provided: If a Responder detects an illegal capability flag combination reported by the Requester, it shall issue an `ERROR` message of `ErrorCode=InvalidRequest` .

285 10.3.1 Supported algorithms block

286 The Supported Algorithms Block reports all options from the `ALGORITHMS` response that are supported by the Responder. The Supported Algorithms Block shall conform to the Table 15 — NEGOTIATE_ALGORITHMS request message format, including all fields from `Param1` through the end of the message, inclusive. When constructing the

Supported Algorithms Block, the Responder shall follow all requirements for the Requester, and shall set all bits and values that reflect algorithms that the Responder supports.

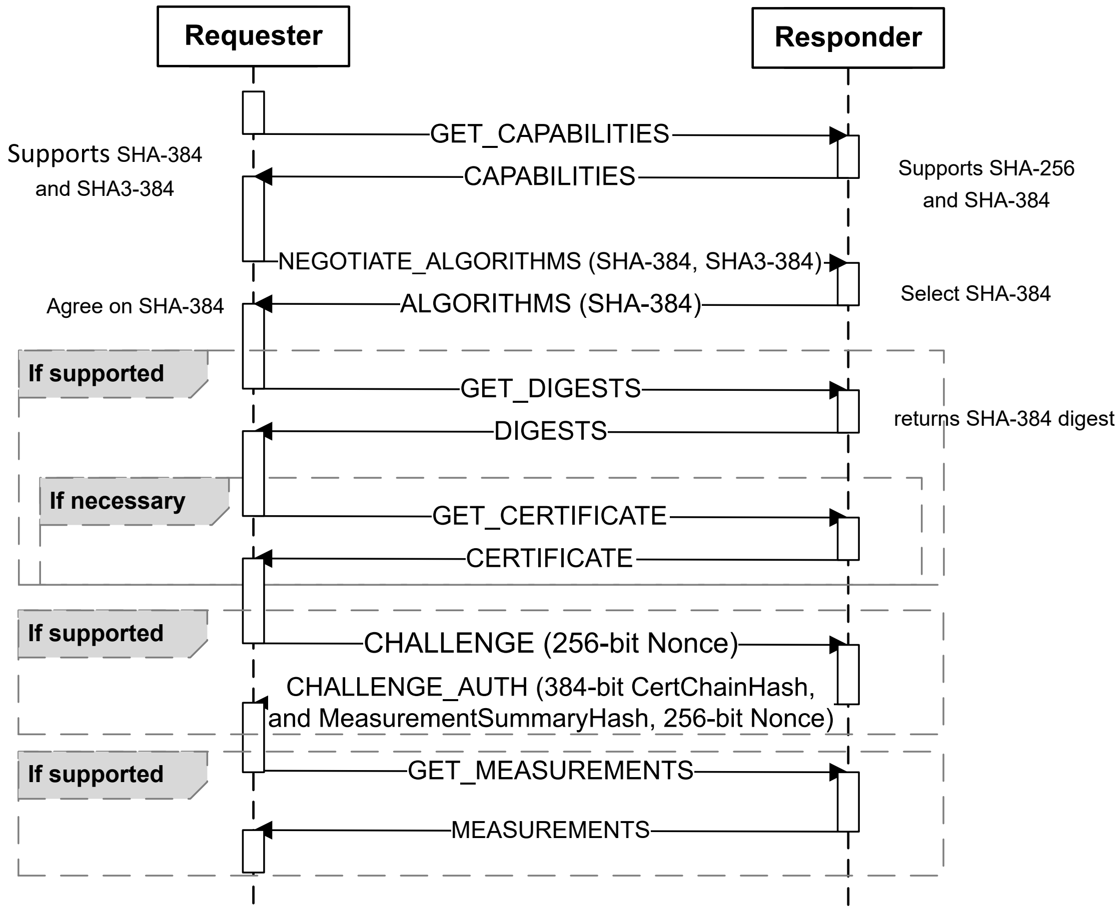
287 10.4 NEGOTIATE_ALGORITHMS request and ALGORITHMS response messages

288 This request message shall negotiate cryptographic algorithms. In SPDM, the Requester issues `NEGOTIATE_ALGORITHMS` to indicate which cryptographic algorithm(s) it supports for each type of cryptographic operation, and the Responder selects one algorithm of each type using the `ALGORITHMS` response message. The selected algorithms shall be used for all relevant cryptographic operations for the duration of the connection. The criteria a Responder uses to determine which algorithm to select when more than one are supported by both endpoints are outside the scope of this specification.

289 Figure 7 — Hashing algorithm selection: Example 1 illustrates how two endpoints negotiate a base hashing algorithm. Endpoint A issues a `NEGOTIATE_ALGORITHMS` request message, and endpoint B returns a selected mutually supported algorithm in the `ALGORITHMS` response.

290 **Figure 7 — Hashing algorithm selection: Example 1**

291



292 If the Requester and Responder support no common algorithms of a particular type, the Responder shall issue an `ALGORITHMS` response message with all appropriate selection field values set to zero to indicate that no selection was made. The Responder should respond to all subsequent requests by this Requester with an `ERROR` message of `ErrorCode=RequestResynch` . The Responder may continue to operate with limited functionality for operations that do not require negotiated cryptographic algorithms.

293 A Requester shall not issue a `NEGOTIATE_ALGORITHMS` request message until it receives a successful `CAPABILITIES` response message.

294 After a Requester issues a `NEGOTIATE_ALGORITHMS` request, it shall not issue any other SPDM requests, with the exception of `GET_VERSION` , until it receives a successful `ALGORITHMS` response message.

295 For each algorithm type, a Responder shall not select both an SPDM-enumerated algorithm and an extended algorithm.

296 The SPDM protocol accounts for the possibility that both endpoints issue `NEGOTIATE_ALGORITHMS` request messages independently of each other. In this case, the endpoint A Requester and endpoint B Responder communication pair

might select a different algorithm from the one selected by the endpoint B Requester and endpoint A Responder communication pair.

297 Table 15 — NEGOTIATE_ALGORITHMS request message format shows the `NEGOTIATE_ALGORITHMS` request message format.

298 **Table 15 — NEGOTIATE_ALGORITHMS request message format**

Byte offset	Field	Size (bytes)	Description
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	RequestResponseCode	1	Shall be `0xE3` = `NEGOTIATE_ALGORITHMS`. See Table 4 — SPDM request codes.
2	Param1	1	Shall be the number of algorithm structure tables in this request using `ReqAlgStruct`.
3	Param2	1	Reserved.
4	Length	2	Shall be the length of the entire request message, in bytes. Length shall be less than or equal to 128 bytes.
6	MeasurementSpecification	1	Bit mask. The Measurement specification field format table defines the format for this field. For each defined measurement specification a Requester supports, the Requester can set the appropriate bits.
7	OtherParamsSupport	1	Shall be the selection bit mask. Bit [3:0] - See Opaque Data Format Support and Selection Table Bit [4] - This field shall be the `ResponderMultiKeyConn` field as Multiple Asymmetric Key Negotiation describes. Bit [7:5] - Reserved.

Byte offset	Field	Size (bytes)	Description
8	BaseAsymAlgo	4	Shall be the bit mask listing Requester-supported SPDM-enumerated asymmetric key signature algorithms for the purpose of signature verification. If the Requester does not support any request/ response pair that requires signature verification, this value shall be set to zero. If the Requester will not send any requests that require a signature, this value should be set to zero. Let `SigLen` be the size of the signature in bytes. • Byte 0 Bit 0. TPM_ALG_RSASSA_2048 where `SigLen` =256. • Byte 0 Bit 1. TPM_ALG_RSAPSS_2048 where `SigLen` =256. • Byte 0 Bit 2. TPM_ALG_RSASSA_3072 where `SigLen` =384. • Byte 0 Bit 3. TPM_ALG_RSAPSS_3072 where `SigLen` =384. • Byte 0 Bit 4. TPM_ALG_ECDSA_ECC_NIST_P256 where `SigLen` =64 (32-byte r followed by 32-byte s). • Byte 0 Bit 5. TPM_ALG_RSASSA_4096 where `SigLen` =512. • Byte 0 Bit 6. TPM_ALG_RSAPSS_4096 where `SigLen` =512. • Byte 0 Bit 7. TPM_ALG_ECDSA_ECC_NIST_P384 where `SigLen` =96 (48-byte r followed by 48-byte s). • Byte 1 Bit 0. TPM_ALG_ECDSA_ECC_NIST_P521 where `SigLen` =132 (66-byte r followed by 66-byte s). • Byte 1 Bit 1. TPM_ALG_SM2_ECC_SM2_P256 where `SigLen` =64 (32-byte `SM2_R` followed by 32-byte `SM2_S`). • Byte 1 Bit 2. EdDSA ed25519 where `SigLen` =64 (32-byte R followed by 32-byte S). • Byte 1 Bit 3. EdDSA ed448 where `SigLen` =114 (57-byte R followed by 57-byte S). • All other values reserved.

Byte offset	Field	Size (bytes)	Description
12	BaseHashAlgo	4	Shall be the bit mask listing Requester-supported SPDM-enumerated cryptographic hashing algorithms. If the Requester does not support any request/response pair that requires hashing operations, this value shall be set to zero. • Byte 0 Bit 0. TPM_ALG_SHA_256 • Byte 0 Bit 1. TPM_ALG_SHA_384 • Byte 0 Bit 2. TPM_ALG_SHA_512 • Byte 0 Bit 3. TPM_ALG_SHA3_256 • Byte 0 Bit 4. TPM_ALG_SHA3_384 • Byte 0 Bit 5. TPM_ALG_SHA3_512 • Byte 0 Bit 6. TPM_ALG_SM3_256 • All other values reserved.

Byte offset	Field	Size (bytes)	Description
16	PqcAsymAlgo	4	Shall be the bit mask listing Requester-supported SPDM-enumerated PQC asymmetric key signature algorithms for the purpose of signature verification. If the Requester does not support any request/ response pair that requires signature verification, this value shall be set to zero. If the Requester will not send any requests that require a signature, this value shall be set to zero. Let `SigLen` be the size of the signature in bytes. • Byte 0 Bit 0. ML-DSA-44 where `SigLen` =2420. • Byte 0 Bit 1. ML-DSA-65 where `SigLen` =3309. • Byte 0 Bit 2. ML-DSA-87 where `SigLen` =4627. • Byte 0 Bit 3. SLH-DSA-SHA2-128s where `SigLen` =7856. • Byte 0 Bit 4. SLH-DSA-SHAKE-128s where `SigLen` =7856. • Byte 0 Bit 5. SLH-DSA-SHA2-128f where `SigLen` =17088. • Byte 0 Bit 6. SLH-DSA-SHAKE-128f where `SigLen` =17088. • Byte 0 Bit 7. SLH-DSA-SHA2-192s where `SigLen` =16224. • Byte 1 Bit 0. SLH-DSA-SHAKE-192s where `SigLen` =16224. • Byte 1 Bit 1. SLH-DSA-SHA2-192f where `SigLen` =35664. • Byte 1 Bit 2. SLH-DSA-SHAKE-192f where `SigLen` =35664. • Byte 1 Bit 3. SLH-DSA-SHA2-256s where `SigLen` =29792. • Byte 1 Bit 4. SLH-DSA-SHAKE-256s where `SigLen` =29792. • Byte 1 Bit 5. SLH-DSA-SHA2-256f where `SigLen` =49856. • Byte 1 Bit 6. SLH-DSA-SHAKE-256f where `SigLen` =49856. • All other values reserved.
20	Reserved	8	Reserved.

Byte offset	Field	Size (bytes)	Description
28	ExtAsymCount	1	Shall be the number of Requester-supported extended asymmetric key signature algorithms (=A) for the purpose of signature verification. `A` + `E` + `ExtAlgCount2` + `ExtAlgCount3` + `ExtAlgCount4` + `ExtAlgCount5` shall be less than or equal to 20. If the Requester does not support any request/response pair that requires signature verification, this value shall be set to zero.
29	ExtHashCount	1	Shall be the number of Requester-supported extended hashing algorithms (=E). `A` + `E` + `ExtAlgCount2` + `ExtAlgCount3` + `ExtAlgCount4` + `ExtAlgCount5` shall be less than or equal to 20. If the Requester does not support any request/response pair that requires hashing operations, this value shall be set to zero.
30	Reserved	1	Reserved.
31	MELspecification	1	Shall be the bit mask. The Measurement Extension Log specification field format table defines the format for this field. The Requester shall set the corresponding bit for each supported measurement extension log (MEL) specification.
32	ExtAsym	4 * `A`	Shall be the list of Requester-supported extended asymmetric key signature algorithms for the purpose of signature verification. Table 31 — Extended Algorithm field format describes the format of this field.
32 + 4 * `A`	ExtHash	4 * `E`	Shall be the list of the extended hashing algorithms supported by Requester. Table 31 — Extended Algorithm field format describes the format of this field.
32 + 4 * `A` + 4 * `E`	ReqAlgStruct	`AlgStructSize`	See the `AlgStructure` request field.

299 `AlgStructSize` is the sum of the size of the following algorithm structure tables. The algorithm structure table shall be present only if the Requester supports that `AlgType`. `AlgType` shall monotonically increase for subsequent entries.

300 Table 16 — Algorithm request structure shows the Algorithm request structure:

301 **Table 16 — Algorithm request structure**

Byte offset	Field	Size (bytes)	Description
0	AlgType	1	Shall be the type of algorithm. • 0x00 and 0x01. Reserved. • 0x02. DHE. • 0x03. `AEADCipherSuite` . • 0x04. `ReqBaseAsymAlg` . • 0x05. `KeySchedule` . • 0x06. `ReqPqcAsymAlg` . • 0x07. `KEMAlg` .
1	AlgCount	1	Shall be the Requester-supported fixed algorithms. • Bit [7:4]. Number of bytes required to describe Requester-supported SPDM-enumerated fixed algorithms (=FixedAlgCount). `FixedAlgCount` + 2 shall be a multiple of 4. • Bit [3:0]. Number of Requester-supported extended algorithms (= `ExtAlgCount`).
2	AlgSupported	`FixedAlgCount`	Shall be the bit mask listing Requester-supported SPDM-enumerated algorithms.
2 + FixedAlgCount	AlgExternal	4 * `ExtAlgCount`	Shall be the list of Requester-supported extended algorithms. Table 31 — Extended Algorithm field format describes the format of this field.

302 The following tables describe the Algorithm request structures mapped to their respective types:

- Table 17 — DHE structure
- Table 18 — AEAD structure
- Table 19 — ReqBaseAsymAlg structure
- Table 20 — KeySchedule structure
- Table 21 — ReqPqcAsymAlg structure
- Table 22 — KEMAlg structure

303 **Table 17 — DHE structure**

Byte offset	Field	Size (bytes)	Description
0	AlgType	1	Shall be `0x02 = DHE`
1	AlgCount	1	• Bit [7:4]. Shall be a value of 2. • Bit [3:0]. Number of Requester-supported extended DHE groups (= `ExtAlgCount2`).

Byte offset	Field	Size (bytes)	Description
2	AlgSupported	2	Shall be the bit mask listing Requester-supported SPDM-enumerated Diffie-Hellman Ephemeral (DHE) groups. Values in parentheses specify the size of the corresponding public values associated with each group. • Byte 0 Bit 0. ffdhe2048 (D = 256). • Byte 0 Bit 1. ffdhe3072 (D = 384). • Byte 0 Bit 2. ffdhe4096 (D = 512). • Byte 0 Bit 3. secp256r1 (D = 64, C = 32). • Byte 0 Bit 4. secp384r1 (D = 96, C = 48). • Byte 0 Bit 5. secp521r1 (D = 132, C = 66). • Byte 0 Bit 6. SM2_P256 (Part 3 and Part 5 of GB/T 32918 specification) (D = 64, C = 32). • All other values reserved.
4	AlgExternal	4 * `ExtAlgCount2`	Shall be the list of Requester-supported extended DHE groups and KEM. Table 31 — Extended Algorithm field format describes the format of this field.

304 **Table 18 — AEAD structure**

Byte offset	Field	Size (bytes)	Description
0	AlgType	1	Shall be the `0x03` = `AEAD`
1	AlgCount	1	• Bit [7:4]. Shall be a value of 2. • Bit [3:0]. Number of Requester-supported extended *AEAD* algorithms (= `ExtAlgCount3`).
2	AlgSupported	2	Shall be the bit mask listing Requester-supported SPDM-enumerated AEAD algorithms. • Byte 0 Bit 0. AES-128-GCM. 128-bit key; 96-bit IV (initialization vector); tag size is specified by transport layer. • Byte 0 Bit 1. AES-256-GCM. 256-bit key; 96-bit IV; tag size is specified by transport layer. • Byte 0 Bit 2. CHACHA20_POLY1305. 256-bit key; 96-bit IV; 128-bit tag. • Byte 0 Bit 3. AEAD_SM4_GCM. 128-bit key; 96-bit IV; tag size is specified by transport layer. • All other values reserved.
4	AlgExternal	4 * `ExtAlgCount3`	Shall be the list of Requester-supported extended AEAD algorithms. Table 31 — Extended Algorithm field format describes the format of this field.

305 **Table 19 — ReqBaseAsymAlg structure**

Byte offset	Field	Size (bytes)	Description
0	AlgType	1	Shall be `0x04` = `ReqBaseAsymAlg`
1	AlgCount	1	• Bit [7:4]. Shall be a value of 2. • Bit [3:0]. Number of Requester-supported extended asymmetric key signature algorithms for the purpose of signature generation (= `ExtAlgCount4`).
2	AlgSupported	2	Shall be the bit mask listing Requester-supported SPDM-enumerated asymmetric key signature algorithms for the purpose of signature generation. If the Requester does not support any request/ response pair that requires signature generation, this value shall be set to zero. • Byte 0 Bit 0. TPM_ALG_RSASSA_2048. • Byte 0 Bit 1. TPM_ALG_RSAPSS_2048. • Byte 0 Bit 2. TPM_ALG_RSASSA_3072. • Byte 0 Bit 3. TPM_ALG_RSAPSS_3072. • Byte 0 Bit 4. TPM_ALG_ECDSA_ECC_NIST_P256. • Byte 0 Bit 5. TPM_ALG_RSASSA_4096. • Byte 0 Bit 6. TPM_ALG_RSAPSS_4096. • Byte 0 Bit 7. TPM_ALG_ECDSA_ECC_NIST_P384. • Byte 1 Bit 0. TPM_ALG_ECDSA_ECC_NIST_P521. • Byte 1 Bit 1. TPM_ALG_SM2_ECC_SM2_P256. • Byte 1 Bit 2. EdDSA ed25519. • Byte 1 Bit 3. EdDSA ed448. • All other values reserved. For details of `SigLen` for each algorithm, see Table 15 — NEGOTIATE_ALGORITHMS request message format.
4	AlgExternal	4 * `ExtAlgCount4`	Shall be the list of Requester-supported extended Base and PQC asymmetric key signature algorithms for the purpose of signature generation. Table 31 — Extended Algorithm field format describes the format of this field.

306 **Table 20 — KeySchedule structure**

Byte offset	Field	Size (bytes)	Description
0	AlgType	1	Shall be `0x05` = `KeySchedule`

Byte offset	Field	Size (bytes)	Description
1	AlgCount	1	• Bit [7:4]. Shall be a value of 2. • Bit [3:0]. Number of Requester-supported extended key schedule algorithms (= `ExtAlgCount5`).
2	AlgSupported	2	Shall be the bit mask listing Requester-supported SPDM-enumerated key schedule algorithms. • Byte 0 Bit 0. SPDM Key Schedule. • All other values reserved.
4	AlgExternal	4 * `ExtAlgCount5`	Shall be the list of Requester-supported extended key schedule algorithms. Table 31 — Extended Algorithm field format describes the format of this field.

307 **Table 21 — ReqPqcAsymAlg structure**

Byte offset	Field	Size (bytes)	Description
0	AlgType	1	Shall be `0x06` = `ReqPqcAsymAlg`
1	AlgCount	1	• Bit [7:4]. Number of bytes for PQC signature algorithms (= `ReqPqcAsymAlgCount`). • Bit [3:0]. Shall be `0` .

Byte offset	Field	Size (bytes)	Description
2	AlgSupported	`ReqPqcAsymAlgCount`	Shall be the bit mask listing Requester-supported SPDM-enumerated PQC asymmetric key signature algorithms for the purpose of signature generation. If the Requester does not support any request/ response pair that requires signature generation, this value shall be set to zero. • Byte 0 Bit 0. ML-DSA-44. • Byte 0 Bit 1. ML-DSA-65. • Byte 0 Bit 2. ML-DSA-87. • Byte 0 Bit 3. SLH-DSA-SHA2-128s. • Byte 0 Bit 4. SLH-DSA-SHAKE-128s. • Byte 0 Bit 5. SLH-DSA-SHA2-128f. • Byte 0 Bit 6. SLH-DSA-SHAKE-128f. • Byte 0 Bit 7. SLH-DSA-SHA2-192s. • Byte 1 Bit 0. SLH-DSA-SHAKE-192s. • Byte 1 Bit 1. SLH-DSA-SHA2-192f. • Byte 1 Bit 2. SLH-DSA-SHAKE-192f. • Byte 1 Bit 3. SLH-DSA-SHA2-256s. • Byte 1 Bit 4. SLH-DSA-SHAKE-256s. • Byte 1 Bit 5. SLH-DSA-SHA2-256f. • Byte 1 Bit 6. SLH-DSA-SHAKE-256f. • All other values reserved.

308 **Table 22 — KEMAlg structure**

Byte offset	Field	Size (bytes)	Description
0	AlgType	1	Shall be `0x07` = `KEMAlg`
1	AlgCount	1	• Bit [7:4]. Number of bytes for KEM algorithms (= `KemAlgCount`). • Bit [3:0]. Shall be `0` .
2	AlgSupported	`KemAlgCount`	Shall be the bit mask for indicating a Requester-supported, SPDM-enumerated KEM algorithms for the purpose of key encapsulation. If the Requester does not support any request/response pair that requires KEM, this value shall be set to zero. • Byte 0 Bit 0. ML-KEM-512. • Byte 0 Bit 1. ML-KEM-768. • Byte 0 Bit 2. ML-KEM-1024. • All other values reserved.

309 Table 23 — ALGORITHMS response message format shows the `ALGORITHMS` response message format.

310 **Table 23 — Successful ALGORITHMS response message format**

Byte offset	Field	Size (bytes)	Description
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	RequestResponseCode	1	Shall be `0x63` = `ALGORITHMS`. See Table 5 — SPDM response codes.
2	Param1	1	Shall be the number of algorithm structure tables in this response using `RespAlgStruct`.
3	Param2	1	Reserved.
4	Length	2	Shall be the length of the response message, in bytes.
6	MeasurementSpecificationSel	1	Bit mask. If the Responder supports MEL (`MEL_CAP=1b` in its `CAPABILITIES` response) or measurements (`MEAS_CAP=01b` or `MEAS_CAP=10b` in its `CAPABILITIES` response), then the Responder shall select one of the measurement specifications supported by the Requester and Responder. No more than one bit shall be set. The Measurement specification field format table defines the format for this field.
7	OtherParamsSelection	1	Shall be the selected Parameter Bit Mask. The Responder shall select one of the opaque data formats supported by both the Requester and the Responder. Thus, no more than one bit shall be set for the opaque data format. • Bit [3:0]. See Opaque Data Format Support and Selection Table. • Bit 4 - This field shall be the `RequesterMultiKeyConnSel` as Multiple Asymmetric Key Negotiation describes. • Bit [7:5]. Reserved.

Byte offset	Field	Size (bytes)	Description
8	MeasurementHashAlgo	4	Shall be the bit mask indicating the SPDM-enumerated hashing algorithms used for measurements. • Byte 0 Bit 0. Raw Bit Stream Only. • Byte 0 Bit 1. TPM_ALG_SHA_256. • Byte 0 Bit 2. TPM_ALG_SHA_384. • Byte 0 Bit 3. TPM_ALG_SHA_512. • Byte 0 Bit 4. TPM_ALG_SHA3_256. • Byte 0 Bit 5. TPM_ALG_SHA3_384. • Byte 0 Bit 6. TPM_ALG_SHA3_512. • Byte 0 Bit 7. TPM_ALG_SM3_256. • If the Responder supports measurements (`MEAS_CAP=01b` or `MEAS_CAP=10b` in its `CAPABILITIES` response) and if `MeasurementSpecificationSel` is non-zero, then exactly one bit in this bit field shall be set. Otherwise, the Responder shall set this field to `0` . • All other values reserved. A Responder shall select bit 0 only if it supports raw bit streams as the only form of measurement; otherwise, the Responder shall select one of the other bits.
12	BaseAsymSel	4	Shall be the bit mask indicating the SPDM-enumerated asymmetric key signature algorithm selected for the purpose of signature generation. If the Responder does not support any request/response pair that requires signature generation, this value shall be set to zero. The total number of bits set in this field and PqcAsymSel shall be no more than one.
16	BaseHashSel	4	Shall be the bit mask indicating the SPDM-enumerated hashing algorithm selected. If the Responder does not support any request/response pair that requires hashing operations, this value shall be set to zero. The Responder shall set no more than one bit.
20	PqcAsymSel	4	Shall be the bit mask indicating the SPDM-enumerated PQC asymmetric key signature algorithm selected for the purpose of signature generation. If the Responder does not support any request/response pair that requires signature generation, this value shall be set to zero. The total number of bits set in this field and BaseAsymSel shall be no more than one.

Byte offset	Field	Size (bytes)	Description
24	Reserved	7	Reserved.
31	MELspecificationSel	1	Shall be the bit mask indicating MEL. The Responder shall select one of the MEL specifications supported by the Requester and Responder. No more than one bit shall be set. The Measurement Extension Log specification field format table defines the format for this field.
32	ExtAsymSelCount	1	Shall be the number of extended asymmetric key signature algorithms selected for the purpose of signature generation. Shall be either `0` or `1` (=A'). If the Responder does not support any request/ response pair that requires signature generation, this value shall be set to zero.
33	ExtHashSelCount	1	Shall be the number of extended hashing algorithms selected. Shall be either `0` or `1` (=E'). If the Responder does not support any request/response pair that requires hashing operations, this value shall be set to zero.
34	Reserved	2	Reserved.
36	ExtAsymSel	4 * `A'`	Shall be the extended asymmetric key signature algorithm selected for the purpose of signature generation. The Responder shall use this asymmetric signature algorithm for all subsequent applicable response messages to the Requester. The extended algorithm field format table describes the format of this field.
36 + 4 * `A'`	ExtHashSel	4 * `E'`	Shall be the extended hashing algorithm selected. The Responder shall use this hashing algorithm during all subsequent response messages to the Requester. The Requester shall use this hashing algorithm during all subsequent applicable request messages to the Responder. The extended algorithm field format table describes the format of this field.
36 + 4 * `A'` + 4 * `E'`	RespAlgStruct	`AlgStructSize`	See Table 24 — Response AlgStructure field format.

311 `AlgStructSize` is the sum of the sizes of all the algorithm structure tables, as the following tables show. An algorithm structure table should be present only if the Responder supports that `AlgType` . `AlgType` shall monotonically increase for subsequent entries.

312 **Table 24 — Response AlgStructure field format**

Byte offset	Field	Size (bytes)	Description
0	AlgType	1	Shall be the type of algorithm. • 0x00 and 0x01. Reserved. • 0x02. DHE. • 0x03. `AEADCipherSuite` . • 0x04. `ReqBaseAsymAlg` . • 0x05. `KeySchedule` . • 0x06. `ReqPqcAsymAlg` . • 0x07. `KEMAlg` . • All other values reserved.
1	AlgCount	1	Shall be the bit mask listing Responder-supported fixed algorithm requested by the Requester. • Bit [7:4]. Number of bytes required to describe Requester-supported SPDM-enumerated fixed algorithms (=FixedAlgCount). `FixedAlgCount` + 2 shall be a multiple of 4. • Bit [3:0]. Number of Requester-supported, Responder-selected, extended algorithms (= `ExtAlgCount'`). This value shall be either 0 or 1.
2	AlgSupported	`FixedAlgCount`	Shall be the bit mask for indicating a Requester-supported, Responder-selected, SPDM-enumerated algorithm. Responder shall set at most one bit to 1.
2 + `FixedAlgCount`	AlgExternal	4 * `ExtAlgCount'`	If present: shall be a Requester-supported, Responder-selected, extended algorithm. Responder shall select at most one extended algorithm. Table 31 — Extended Algorithm field format describes the format of this field.

313 The following tables describe the algorithm types and their associated fixed fields:

- Table 25 — DHE structure
- Table 26 — AEAD structure
- Table 27 — ReqBaseAsymAlg structure
- Table 28 — KeySchedule structure
- Table 29 — ReqPqcAsymAlg structure
- Table 30 — KEMAlg structure
- Table 31 — Extended Algorithm field format

314 **Table 25 — DHE structure**

Byte offset	Field	Size (bytes)	Description
0	AlgType	1	Shall be `0x02 = DHE`

Byte offset	Field	Size (bytes)	Description
1	AlgCount	1	• Bit [7:4]. Shall be a value of 2. • Bit [3:0]. Shall be the number of Requester-supported, Responder-selected, extended DHE groups (= `ExtAlgCount2'`). This value shall be either 0 or 1.
2	AlgSupported	2	Shall be the bit mask for indicating a Requester-supported, Responder-selected, SPDM-enumerated DHE group. Values in parentheses specify the size of the corresponding public values associated with each group. The total number of bits set in this field and the `AlgSupported` field of KEMAlg structure (Table 30) shall be no more than one. • Byte 0 Bit 0. ffdhe2048 (D = D' = 256). • Byte 0 Bit 1. ffdhe3072 (D = D' = 384). • Byte 0 Bit 2. ffdhe4096 (D = D' = 512). • Byte 0 Bit 3. secp256r1 (D' = 64, C' = 32) • Byte 0 Bit 4. secp384r1 (D' = 96, C' = 48). • Byte 0 Bit 5. secp521r1 (D = D' = 132, C = C' = 66). • Byte 0 Bit 6. SM2_P256 (Part 3 and Part 5 of GB/T 32918) (D = D' = 64, C = C' = 32). • All other values reserved.
4	AlgExternal	4 * `ExtAlgCount2'`	If present: shall be a Requester-supported, Responder-selected, extended DHE or KEM algorithm. Table 31 — Extended Algorithm field format describes the format of this field.

315 **Table 26 — AEAD structure**

Byte offset	Field	Size (bytes)	Description
0	AlgType	1	Shall be `0x03` = `AEAD`
1	AlgCount	1	• Bit [7:4]. Shall be a value of 2. • Bit [3:0]. Shall be the number of Requester-supported, Responder-selected, extended AEAD algorithms (= `ExtAlgCount3'`). This value shall be either 0 or 1.

Byte offset	Field	Size (bytes)	Description
2	AlgSupported	2	Shall be the bit mask for indicating a Requester-supported, Responder-selected, SPDM-enumerated AEAD algorithm. • Byte 0 Bit 0. AES-128-GCM. • Byte 0 Bit 1. AES-256-GCM. • Byte 0 Bit 2. CHACHA20_POLY1305. • Byte 0 Bit 3. AEAD_SM4_GCM. • All other values reserved.
4	AlgExternal	4 * `ExtAlgCount3'`	If present: shall be a Requester-supported, Responder-selected, extended AEAD algorithm. Table 31 — Extended Algorithm field format describes the format of this field.

316 **Table 27 — ReqBaseAsymAlg structure**

Byte offset	Field	Size (bytes)	Description
0	AlgType	1	Shall be `0x04` = `ReqBaseAsymAlg`
1	AlgCount	1	• Bit [7:4]. Shall be a value of 2. • Bit [3:0]. Number of Requester-supported, Responder-selected, extended asymmetric key signature algorithms (= `ExtAlgCount4'`) for the purpose of signature verification. This value shall be either 0 or 1.

Byte offset	Field	Size (bytes)	Description
2	AlgSupported	2	Shall be the bit mask for indicating a Requester-supported, Responder-selected, SPDM-enumerated asymmetric key signature algorithm for the purpose of signature verification. If the Responder does not support any request/response pair that requires signature verification, this value shall be set to zero. If the Responder will not send any messages that require a signature, this value should be set to zero. The total number of bits set in this field and in the `AlgSupported` field of `ReqPqcAsymAlg` shall be no more than one. • Byte 0 Bit 0. TPM_ALG_RSASSA_2048. • Byte 0 Bit 1. TPM_ALG_RSAPSS_2048. • Byte 0 Bit 2. TPM_ALG_RSASSA_3072. • Byte 0 Bit 3. TPM_ALG_RSAPSS_3072. • Byte 0 Bit 4. TPM_ALG_ECDSA_ECC_NIST_P256. • Byte 0 Bit 5. TPM_ALG_RSASSA_4096. • Byte 0 Bit 6. TPM_ALG_RSAPSS_4096. • Byte 0 Bit 7. TPM_ALG_ECDSA_ECC_NIST_P384. • Byte 1 Bit 0. TPM_ALG_ECDSA_ECC_NIST_P521. • Byte 1 Bit 1. TPM_ALG_SM2_ECC_SM2_P256. • Byte 1 Bit 2. EdDSA ed25519. • Byte 1 Bit 3. EdDSA ed448. • All other values reserved. For details of `SigLen` for each algorithm, see Table 15 — NEGOTIATE_ALGORITHMS request message format.
4	AlgExternal	4 * `ExtAlgCount4'`	If present: shall be a Requester-supported, Responder-selected extended Base or PQC asymmetric key signature algorithm for the purpose of signature verification. Table 31 — Extended Algorithm field format describes the format of this field.

317 **Table 28 — KeySchedule structure**

Byte offset	Field	Size (bytes)	Description
0	AlgType	1	Shall be `0x05` = `KeySchedule`

Byte offset	Field	Size (bytes)	Description
1	AlgCount	1	• Bit [7:4]. Shall be a value of 2. • Bit [3:0]. Shall be the number of Requester-supported, Responder-selected, extended key schedule algorithms (= `ExtAlgCount5'`). This value shall be either 0 or 1.
2	AlgSupported	2	Shall be the bit mask for indicating a Requester-supported, Responder-selected, SPDM-enumerated key schedule algorithm. • Byte 0 Bit 0. SPDM key schedule. • All other values reserved.
4	AlgExternal	4 * `ExtAlgCount5'`	If present: shall be a Requester-supported, Responder-selected, extended key schedule algorithm. Table 31 — Extended Algorithm field format describes the format of this field.

318 **Table 29 — ReqPqcAsymAlg structure**

Byte offset	Field	Size (bytes)	Description
0	AlgType	1	Shall be `0x06` = `ReqPqcAsymAlg`
1	AlgCount	1	• Bit [7:4]. Number of bytes for PQC signature algorithms (= `ReqPqcAsymAlgCount`). • Bit [3:0]. Shall be `0` .

Byte offset	Field	Size (bytes)	Description
2	AlgSupported	`ReqPqcAsymAlgCount`	Shall be the bit mask for indicating a Requester-supported, Responder-selected, SPDM-enumerated PQC asymmetric key signature algorithm for the purpose of signature verification. If the Responder does not support any request/response pair that requires signature verification, this value shall be set to zero. If the Responder will not send any messages that require a signature, this value shall be set to zero. • Byte 0 Bit 0. ML-DSA-44. • Byte 0 Bit 1. ML-DSA-65. • Byte 0 Bit 2. ML-DSA-87. • Byte 0 Bit 3. SLH-DSA-SHA2-128s. • Byte 0 Bit 4. SLH-DSA-SHAKE-128s. • Byte 0 Bit 5. SLH-DSA-SHA2-128f. • Byte 0 Bit 6. SLH-DSA-SHAKE-128f. • Byte 0 Bit 7. SLH-DSA-SHA2-192s. • Byte 1 Bit 0. SLH-DSA-SHAKE-192s. • Byte 1 Bit 1. SLH-DSA-SHA2-192f. • Byte 1 Bit 2. SLH-DSA-SHAKE-192f. • Byte 1 Bit 3. SLH-DSA-SHA2-256s. • Byte 1 Bit 4. SLH-DSA-SHAKE-256s. • Byte 1 Bit 5. SLH-DSA-SHA2-256f. • Byte 1 Bit 6. SLH-DSA-SHAKE-256f. • All other values reserved. For details of `SigLen` for each algorithm, see Table 15 — NEGOTIATE_ALGORITHMS request message format.

319 **Table 30 — KEMAlg structure**

Byte offset	Field	Size (bytes)	Description
0	AlgType	1	Shall be `0x07` = `KEMAlg`
1	AlgCount	1	• Bit [7:4]. Number of bytes for KEM algorithms (= `KemAlgCount`). • Bit [3:0]. Shall be `0` .

Byte offset	Field	Size (bytes)	Description
2	AlgSupported	`KemAlgCount`	Shall be the bit mask for indicating a Requester-supported, Responder-selected, SPDM-enumerated KEM algorithms for the purpose of key encapsulation. If the Requester does not support any request/response pair that requires KEM, this value shall be set to zero. • Byte 0 Bit 0. ML-KEM-512. • Byte 0 Bit 1. ML-KEM-768. • Byte 0 Bit 2. ML-KEM-1024. • All other values reserved.

320 **Table 31 — Extended Algorithm field format**

Byte offset	Field	Size (bytes)	Description
0	Registry ID	1	Shall represent the registry or standards body. The **ID** column of Table 65 — Registry or standards body ID describes the value of this field.
1	Reserved	1	Reserved.
2	Algorithm ID	2	Shall indicate the desired algorithm. The registry or standards body owns the value of this field. See Table 65 — Registry or standards body ID. At present, DMTF does not define any algorithms for use in extended algorithms fields.

321 **Table 32 — Opaque Data Format Support and Selection**

Bit offset	Field	Description
0	OpaqueDataFmt0	If set, this bit shall indicate that the format for all `OpaqueData` fields in this specification is defined by the device vendor or other standards body.
1	OpaqueDataFmt1	If set, this bit shall indicate that the format for all `OpaqueData` fields in this specification is defined by the General opaque data format.
[3:2]	Reserved	Reserved.

322 The Opaque Data Format Selection Table shows the bit definition for the format of the Opaque data fields. A Requester may set more than one bit in the table to indicate each supported format. A Responder shall select no more than one of the bits supported by both the Requester and the Responder in this table. If the Requester or the Responder does not set a bit, then all `OpaqueData` fields in this specification shall be absent by setting the respective `OpaqueDataLength` field to a value of zero.

323 **Table 33 — Measurement Specification Field Format**

Bit offset	Field	Description
0	DMTFmeasSpec	This bit shall indicate a DMTF-defined measurement specification. Table 58 — DMTF measurement specification format defines the format for this measurement specification.
[1:7]	Reserved	Reserved

324 The Measurement Specification Field Format Table describes the field format for Measurement specification related fields. The selected measurement specification (`MeasurementSpecificationSel`) is used in the `MEASUREMENTS` response. See Measurement block and `GET_MEASUREMENTS` for details.

325 **Table 34 — Measurement Extension Log Specification Field Format**

Bit offset	Field	Description
0	DMTFmelSpec	This bit indicates a DMTF-defined measurement extension log specification. Refer to the DMTF Measurement Extension Log Format clause for details. If the Requester supports the DMTF-defined measurement extension log specification, it shall set this bit to 1 in `MELspecification` . If the Responder selects the DMTF-defined measurement extension log specification for constructing the MEL, it shall set this bit to 1 in `MELspecificationSel` .
[1:7]	Reserved	Reserved

326 The Measurement Extension Log Specification Field Format Table describes the field format for MEL specification related fields. The selected MEL specification (`MELspecificationSel`) is used in construction of the MEL.

327 10.4.1 Connection behavior after VCA

328 With the successful completion of the `ALGORITHMS` message, all the parameters of the SPDM connection have been determined. Thus, all SPDM message exchanges after the VCA messages shall comply with the selected parameters in the `ALGORITHMS` message, with the exception of `GET_VERSION` and `VERSION` messages, or unless otherwise stated in this specification. To explain this behavior, suppose a Responder supports both RSA and ECDSA asymmetric algorithms. For an SPDM connection, the Responder selects the `TPM_ALG_RSASSA_2048` asymmetric algorithm in `BaseAsymSel` and the `TPM_ALG_SHA_256` hash algorithm in `BaseHashSel` . If the Requester on that same connection issues `GET_DIGESTS` , the Responder returns `TPM_ALG_SHA_256` digests only for those populated slots that can provide a `TPM_ALG_RSASSA_2048` signature for a `CHALLENGE_AUTH` response. The Responder would violate this requirement if it returns one or more digests of populated slots that perform ECDSA signatures or if it uses a different hash algorithm to create the digests.

329 Unless otherwise stated in this specification, and with the exception of `GET_VERSION` , if a Requester issues a request

that violates one or more of the negotiated or selected parameters of a given connection, the Responder shall either silently discard the request or return an `ERROR` message with an appropriate error code.

330 ### 10.4.2 Multiple asymmetric key negotiation

331 The Requester and Responder can negotiate the parameters of multiple asymmetric key support for the SPDM connection. As with other parameters in this request and response, the Responder makes the selection and the Requester indicates its support. There are two sets of multiple asymmetric key use parameters to negotiate: one for Responder authentication and one for Requester authentication.

332 ### 10.4.3 Multiple asymmetric key use for Responder authentication

333 The Responder shall report the multiple asymmetric keys capability in the `MULTI_KEY_CAP` field of `CAPABILITIES`.

334 If `MULTI_KEY_CAP` is `10b`, the `ResponderMultiKeyConn` field in `NEGOTIATE_ALGORITHMS` determines whether or not the SPDM connection uses multiple asymmetric keys for Responder authentication. The Requester makes the decision for the SPDM connection in the `ResponderMultiKeyConn` field. If the Requester sets the `ResponderMultiKeyConn` field, the Responder shall support multiple asymmetric keys in the SPDM connection for Responder authentication. If `ResponderMultiKeyConn` is not set, the Responder shall support only one key pair per supported asymmetric algorithm for this SPDM connection.

335 If `MULTI_KEY_CAP` is `01b`, the Responder determines that the SPDM connection uses multiple asymmetric keys. The `ResponderMultiKeyConn` field in `NEGOTIATE_ALGORITHMS` shall be set to acknowledge the Responder capability.

336 If `MULTI_KEY_CAP` is `00b`, the Responder determines that the SPDM connection does not use multiple asymmetric keys. The `ResponderMultiKeyConn` field in `NEGOTIATE_ALGORITHMS` shall be cleared.

337 ### 10.4.4 Multiple asymmetric key use for Requester authentication

338 The Requester shall report the multiple asymmetric keys capability for Requester authentication in the `MULTI_KEY_CAP` field of `GET_CAPABILITIES`.

339 If `MULTI_KEY_CAP` is `10b`, the `RequesterMultiKeyConnSel` field in the `ALGORITHMS` message determines whether or not the SPDM connection uses multiple asymmetric keys for Requester authentication, such as in mutual authentication. The Responder makes the decision for the SPDM connection in `RequesterMultiKeyConnSel`. If the Responder sets the `RequesterMultiKeyConnSel` field, the Requester shall support multiple asymmetric keys in this SPDM connection for Requester authentication. If `RequesterMultiKeyConnSel` is not set, the Requester shall support only one key pair per supported asymmetric algorithm for this SPDM connection.

340 If `MULTI_KEY_CAP` is `01b`, the Requester determines that the SPDM connection uses multiple asymmetric keys. The `RequesterMultiKeyConnSel` field in the `ALGORITHMS` message shall be set to acknowledge the Requester capability.

341 If `MULTI_KEY_CAP` is `00b`, the Requester determines that the SPDM connection does not use multiple asymmetric keys. The `RequesterMultiKeyConnSel` field in the `ALGORITHMS` message shall be cleared.

342
10.4.5 Multiple asymmetric key connection

343 For the remainder of this specification, the boolean variables `MULTI_KEY_CONN_REQ` and `MULTI_KEY_CONN_RSP` indicate whether or not the responding SPDM endpoint supports more than one key pair for one or more asymmetric algorithms for key pairs belonging to it in this SPDM connection. If the responding endpoint is the Requester, then `MULTI_KEY_CONN_REQ` is used. See Table 35 — MULTI_KEY_CONN_REQ value calculation. If the responding endpoint is the Responder, then `MULTI_KEY_CONN_RSP` is used. See Table 36 — MULTI_KEY_CONN_RSP value calculation.

344 **Table 35 — MULTI_KEY_CONN_REQ value calculation**

`MULTI_KEY_CAP` in `GET_CAPABILITIES`	`RequesterMultiKeyConnSel` in `ALGORITHMS`	`MULTI_KEY_CONN_REQ`
`00b`	0	false
`00b`	1	invalid
`01b`	0	invalid
`01b`	1	true
`10b`	0	false
`10b`	1	true

345 **Table 36 — MULTI_KEY_CONN_RSP value calculation**

`MULTI_KEY_CAP` in `CAPABILITIES`	`ResponderMultiKeyConn` in `NEGOTIATE_ALGORITHMS`	`MULTI_KEY_CONN_RSP`
`00b`	0	false
`00b`	1	invalid
`01b`	0	invalid
`01b`	1	true
`10b`	0	false
`10b`	1	true

346 If the responding SPDM endpoint has `MULTI_KEY_CAP` set to `00b`, then the corresponding `MULTI_KEY_CONN_REQ` or `MULTI_KEY_CONN_RSP` shall be false.

347 If the responding SPDM endpoint has `MULTI_KEY_CAP` set to `01b`, then the corresponding `MULTI_KEY_CONN_REQ` or `MULTI_KEY_CONN_RSP` shall be true.

348 If the responding SPDM endpoint has `MULTI_KEY_CAP` set to `10b`, then the value of the corresponding `MULTI_KEY_CONN_REQ` or `MULTI_KEY_CONN_RSP` depends on the peer endpoint. If the responding SPDM endpoint is the Requester and if `RequesterMultiKeyConnSel` is set by the Responder, then the value of `MULTI_KEY_CONN_REQ` shall be true. If the responding SPDM endpoint is the Responder and if `ResponderMultiKeyConn` is set by the Requester, then

the value of `MULTI_KEY_CONN_RSP` shall be true. In all other cases, the value of the corresponding `MULTI_KEY_CONN_REQ` or `MULTI_KEY_CONN_RSP` shall be false.

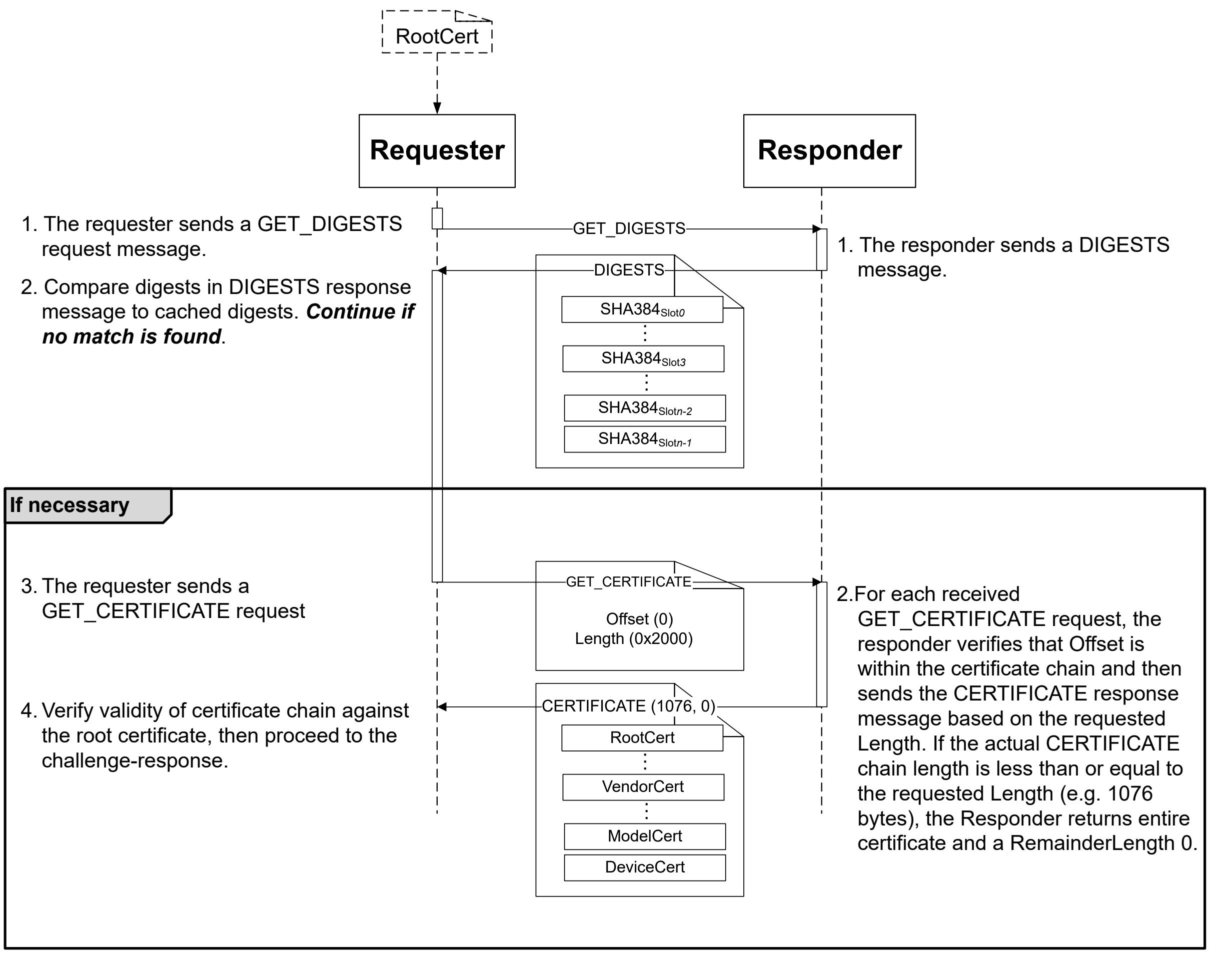
349 10.5 Responder identity authentication

350 This clause describes request messages and response messages associated with the identity of the Responder's authentication operations. The `GET_DIGESTS` and `GET_CERTIFICATE` messages shall be supported by a Responder that returns `CERT_CAP=1` in its `CAPABILITIES` response message. The `CHALLENGE` message that this clause defines shall be supported by a Responder that returns `CHAL_CAP=1` in its `CAPABILITIES` response message. The `GET_DIGESTS` and `GET_CERTIFICATE` messages are not applicable if the public key of the Responder was provisioned to the Requester in a trusted environment.

351 Figure 8 — Responder authentication: Example certificate retrieval flow shows the high-level request-response message flow and sequence for *certificate* retrieval.

352 **Figure 8 — Responder authentication: Example certificate retrieval flow**

353



354 The `GET_DIGESTS` request message and `DIGESTS` response message can optimize the amount of data required to be transferred from the Responder to the Requester, due to the potentially large size of a certificate chain. The cryptographic hash values of every certificate chain stored on an endpoint are returned with the `DIGESTS` response message, enabling the Requester to compare these values to previously retrieved and cached certificate chain hash values and detect any changes to the certificate chains stored on the device before issuing a `GET_CERTIFICATE` request message.

355 For the runtime challenge-response flow, the signature field in the `CHALLENGE_AUTH` response message payload shall contain the signature generated by using the private key associated with the leaf certificate over the hash of the message transcript. See Table 51 — Request ordering and message transcript computation rules for M1/M2.

356 This ensures cryptographic binding between a specific request message from a specific Requester and a specific response message from a specific Responder, which enables the Requester to detect the presence of an active adversary attempting to downgrade cryptographic algorithms or SPDM versions.

357 Furthermore, a Requester-generated *nonce* protects the challenge-response from replay attacks, whereas a Responder-generated nonce prevents the Responder from signing over arbitrary data that the Requester dictates. The message transcript generation for the signature computation is restarted as of the most recent `GET_VERSION` request received.

358 10.6 Requester identity authentication

359 If a Requester supports mutual authentication, it shall comply with all requirements placed on a Responder as specified in Responder identity authentication.

360 If a Responder supports mutual authentication, it shall comply with all requirements placed on a Requester as specified in Responder identity authentication. The preceding two statements essentially describe a role reversal.

361 10.7 Certificates and certificate chains

362 Each SPDM endpoint that supports identity authentication using certificates shall carry at least one complete certificate chain. A certificate chain contains an ordered list of certificates, presented as the binary (byte) concatenation of the fields that Table 37 — Certificate chain format shows. In the context of this specification, a complete certificate chain is one where: (i) the first certificate either is signed by a Root Certificate (a certificate that specifies a trust anchor) or is a Root Certificate itself, (ii) each subsequent certificate is signed by the preceding certificate, and (iii) the final certificate contains the public key used to authenticate the SPDM endpoint. The final certificate is called the *leaf certificate*.

363 If an SPDM endpoint does not support multiple asymmetric keys (`MULTI_KEY_CAP=0`), the SPDM endpoint shall contain a single public-private key pair per supported algorithm for its leaf certificates, regardless of how many certificate chains are stored on the device. The Responder selects a single asymmetric key signature algorithm per Requester regardless of the value of `MULTI_KEY_CAP` field.

364 Certificate chains are stored in logical locations called *slots*. Each supported slot shall either be empty or contain one complete certificate chain. A device shall not contain more than eight slots. Slots are numbered 0 through 7 inclusive. Slot 0 is populated by default. If a device uses the `DeviceCert` model (`ALIAS_CERT_CAP=0b` in its `CAPABILITIES`

response) and if the corresponding `MULTI_KEY_CONN_REQ` or `MULTI_KEY_CONN_RSP` is false, then the certificate chain in every populated slot shall use the `DeviceCert` model. If a device uses the `AliasCert` model (`ALIAS_CERT_CAP=1b` in its `CAPABILITIES` response) and if the corresponding `MULTI_KEY_CONN_REQ` or `MULTI_KEY_CONN_RSP` is false, then the certificate chain in every populated slot shall use the `AliasCert` model. If the corresponding `MULTI_KEY_CONN_REQ` or `MULTI_KEY_CONN_RSP` is false, the `CertModel` field in certificate info table shall always be zero, no matter whether the device uses the `DeviceCert` model or the `AliasCert` model.

365 If the corresponding `MULTI_KEY_CONN_REQ` or `MULTI_KEY_CONN_RSP` is true, the certificate model for each populated certificate slot can be different. Multiple asymmetric key support allows the use of the generic certificate model. The use of the `GenericCert` model shall be prohibited when the corresponding `MULTI_KEY_CONN_REQ` or `MULTI_KEY_CONN_RSP` is false.

366 In all cases, the certificate model for slot 0 shall be either the device certificate model or the alias certificate model.

367 Additional slots may be populated through the supply chain such as by a platform integrator or by an end user such as an IT administrator. A slot mask identifies the certificate chains in the eight slots. Similarly, if the Requester supports mutual authentication and if `MULTI_KEY_CONN_REQ` is false, a Requester device shall use either the `DeviceCert` model or the `AliasCert` model and the certificate chain in every populated slot shall use the same model. Note that the Requester does not have capability flags to indicate the certificate model.

368 If an endpoint supports certificates, then Slot 0 is the default certificate chain slot. Slot 0 shall contain a valid certificate chain unless the device has not yet had a certificate chain provisioned and is in a trusted environment.

369 Each certificate in a chain shall be in ASN.1 DER-encoded X.509 v3 format as RFC 5280 defines. The ASN.1 DER encoding of each individual certificate can be analyzed to determine its length.

370 To allow for flexibility in supporting multiple certificate models, the minimum number of certificates within a certificate chain shall be one and a chain shall contain a leaf certificate.

371 The leaf certificate in the device certificate model shall be the `DeviceCert` leaf certificate. The leaf certificate in an alias certificate model shall be the `AliasCert` leaf certificate. In a generic certificate model, the leaf certificate shall be the `GenericCert` leaf certificate. When the corresponding `MULTI_KEY_CONN_REQ` or `MULTI_KEY_CONN_RSP` is false and a certificate chain consists of a single certificate, that certificate can only be a `DeviceCert` leaf certificate. When the corresponding `MULTI_KEY_CONN_REQ` or `MULTI_KEY_CONN_RSP` is true and a certificate chain consists of a single certificate, that certificate is either a `DeviceCert` or a `GenericCert` leaf certificate.

372 When authenticating an SPDM endpoint, a valid certificate slot (`SlotID`), for slots 0 - 7 inclusively is a supported certificate slot which contains both a certificate chain and its corresponding key pair. If a request uses an invalid certificate slot, the responding SPDM endpoint shall either respond with an `ERROR` message or silently discard the request.

373 Table 37 — Certificate chain format describes the certificate chain format:

374 **Table 37 — Certificate chain format**

Byte offset	Field	Size (bytes)	Description
0	Length	4	Shall be the total length of the certificate chain, in bytes, including all fields in this table. This field is little endian.

Byte offset	Field	Size (bytes)	Description
4	RootHash	`H`	Shall be the digest of the Root Certificate. Note that the Root Certificate is ASN.1 DER-encoded for this digest. This field shall be in hash byte order. `H` is the output size, in bytes, of the hash algorithm selected by the most recent `ALGORITHMS` response.
4 + `H`	Certificates	`Length` - (4 + `H`)	Shall be a complete certificate chain consisting of one or more ASN.1 DER-encoded X.509 v3 certificates. This field shall be in Encoded ASN.1 byte order.

375 10.8 GET_DIGESTS request and DIGESTS response messages

376 This request message shall retrieve the certificate chain digests.

377 Table 38 — GET_DIGESTS request message format shows the `GET_DIGESTS` request message format.

378 The digests in Table 39 — Successful DIGESTS response message format shall be computed over the certificate chain as Table 37 — Certificate chain format shows.

379 When the corresponding `MULTI_KEY_CONN_REQ` or `MULTI_KEY_CONN_RSP` is true, certificate slots have four states that can be reported by the endpoint. The sub-bullet of each state describes how the state is represented in the `DIGESTS` response.

1. Does not exist
   - The corresponding bit in `SupportedSlotMask` is not set.
2. Exists and empty
   - The corresponding bit in `SupportedSlotMask` is set and the corresponding bit in `ProvisionedSlotMask` is not set.
3. Exists with key
   - The corresponding bits in `SupportedSlotMask` and `ProvisionedSlotMask` are set, but the value of the corresponding `CertModel` field is zero.
4. Exists with key and cert
   - The corresponding bits in `SupportedSlotMask` and `ProvisionedSlotMask` are set, and the value of the corresponding `CertModel` field is non-zero.

380 When a certificate slot does not exist, it shall remain in this state for the remainder of the SPDM connection. The "exists and empty" state indicates the presence of a certificate slot where neither a key nor a certificate has been provisioned yet. The "Exists with key" state indicates the certificate slot has only an asymmetric key associated with it but no certificate chain. The "Exists with key and cert" state indicates the certificate has both an asymmetric key assigned to it and a certificate chain. The "Exists with key and cert" state is a fully provisioned state. When a certificate slot exists, the typical progression of states starts at "exists and empty", followed by "Exists with key", and ends with "Exists with key and cert".

381 **Table 38 — GET_DIGESTS request message format**

Byte offset	Field	Size (bytes)	Description
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	RequestResponseCode	1	Shall be `0x81` = `GET_DIGESTS` . See Table 4 — SPDM request codes.
2	Param1	1	Reserved.
3	Param2	1	Reserved.

382 **Table 39 — Successful DIGESTS response message format**

Byte offset	Field	Size (bytes)	Description
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	RequestResponseCode	1	`0x01` = `DIGESTS` . See Table 5 — SPDM response codes.
2	Param1	1	SupportedSlotMask. This field indicates which slots the responding SPDM endpoint supports. If certificate slot X exists in the responding SPDM endpoint, the bit in position X of this byte shall be set. (Bit 0 is the least significant bit of the byte.) Likewise, if certificate slot X does not exist in the responding SPDM endpoint, bit X of this byte shall not be set and certificate slot X shall be an invalid value in various slot ID fields (`SlotID`) across all SPDM request messages that contain this field.
3	Param2	1	ProvisionedSlotMask. If slot K contains a certificate chain that supports the currently negotiated algorithms for the connection, bit K of this byte shall be set. (Bit 0 is the least significant bit of the byte.) Additionally, if the corresponding `MULTI_KEY_CONN_REQ` or `MULTI_KEY_CONN_RSP` is true and if slot K contains an associated key pair that supports the currently negotiated algorithms for the connection, bit K of this byte shall be set. For all fields from `Digest` to `KeyUsageMask` inclusive, the number of fields returned (denoted by `n`) shall be equal to the number of bits set in this byte. These fields shall be returned in order of increasing slot number. If a bit is set in this field, the corresponding bit in `SupportedSlotMask` shall also be set.

Byte offset	Field	Size (bytes)	Description
4	Digest[0]	`H`	Digest of the certificate chain in `CertSlot[0]` . This field shall be in hash byte order.
...	...	...	...
4 + `H` * (n - 1)	Digest[n-1]	`H`	Digest of the certificate chain in `CertSlot[n-1]` . This field shall be in hash byte order. If a certificate chain is not present in this slot, the value of this field shall be all zeros.
4 + (`H` * n)	KeyPairID[0]	1	Shall be the `KeyPairID` of the key pair associated with `CertSlot[0]` . This field shall be present if the corresponding `MULTI_KEY_CONN_REQ` or `MULTI_KEY_CONN_RSP` is true. Otherwise, it shall be absent.
...	...	...	...
3 + (`H` + 1) * n	KeyPairID[n-1]	1	Shall be the `KeyPairID` of the key pair associated with `CertSlot[n-1]` . This field shall be present if the corresponding `MULTI_KEY_CONN_REQ` or `MULTI_KEY_CONN_RSP` is true. Otherwise, it shall be absent.
4 + (`H` + 1) * n	CertificateInfo[0]	1	Shall be the certificate information for `CertSlot[0]` . The format of this field shall be the format that the certificate info table defines. This field shall be present if the corresponding `MULTI_KEY_CONN_REQ` or `MULTI_KEY_CONN_RSP` is true. Otherwise, it shall be absent.
...	...	...	...
3 + (`H` + 2) * n	CertificateInfo[n-1]	1	Shall be the certificate information for `CertSlot[n-1]` . The format of this field shall be the format that the certificate info table defines. This field shall be present if the corresponding `MULTI_KEY_CONN_REQ` or `MULTI_KEY_CONN_RSP` is true. Otherwise, it shall be absent.
4 + (`H` + 2) * n	KeyUsageMask[0]	2	Shall be the key usage bit mask for `CertSlot[0]` . The format of this field shall be the format that the key usage bit mask table defines. This field shall be present if the corresponding `MULTI_KEY_CONN_REQ` or `MULTI_KEY_CONN_RSP` is true. Otherwise, it shall be absent.

Byte offset	Field	Size (bytes)	Description
...	...	...	...
2 + (`H` + 4) * n	KeyUsageMask[n-1]	2	Shall be the key usage bit mask for `CertSlot[n-1]`. The format of this field shall be the format that the key usage bit mask table defines. This field shall be present if the corresponding `MULTI_KEY_CONN_REQ` or `MULTI_KEY_CONN_RSP` is true. Otherwise, it shall be absent.

383 Table 40 — Certificate info shows the format for the `CertificateInfo` fields.

384 **Table 40 — Certificate info**

Bit offset	Field	Description
[2:0]	CertModel	The value of this field shall indicate the certificate model that the certificate slot uses. • Value of 0 indicates either that the certificate slot does not contain any certificates or that the corresponding `MULTI_KEY_CONN_REQ` or `MULTI_KEY_CONN_RSP` is false. • Value of 1 indicates that the certificate slot uses the `DeviceCert` model. • Value of 2 indicates that the certificate slot uses the `AliasCert` model. • Value of 3 indicates that the certificate slot uses the `GenericCert` model. • All other values reserved.
[7:3]	Reserved	Reserved

385 Table 41 — Key usage bit mask shows the format for the `KeyUsageMask` fields.

386 **Table 41 — Key usage bit mask**

Bit offset	Field	Description
0	KeyExUse	If set, the `SlotID` fields in `KEY_EXCHANGE`, `KEY_EXCHANGE_RSP` and `FINISH` can specify this certificate slot. If not set, the `SlotID` fields in `KEY_EXCHANGE`, `KEY_EXCHANGE_RSP` and `FINISH` shall not specify this certificate slot.

Bit offset	Field	Description
1	ChallengeUse	If set, the `SlotID` fields in `CHALLENGE` and `CHALLENGE_AUTH` can specify this certificate slot. If not set, the `SlotID` fields in `CHALLENGE` and `CHALLENGE_AUTH` shall not specify this certificate slot.
2	MeasurementUse	If set, the `SlotID` fields in `GET_MEASUREMENTS` and `MEASUREMENTS` can specify this certificate slot. If not set, the `SlotID` fields in `GET_MEASUREMENTS` and `MEASUREMENTS` shall not specify this certificate slot.
3	EndpointInfoUse	If set, the `SlotID` fields in `GET_ENDPOINT_INFO` and `ENDPOINT_INFO` can specify this certificate slot. If not set, the `SlotID` fields in `GET_ENDPOINT_INFO` and `ENDPOINT_INFO` shall not specify this certificate slot.
[13:4]	Reserved	Reserved
14	StandardsKeyUse	If set, this field shall indicate usage defined by standards other than specifications defined by DMTF.
15	VendorKeyUse	If set, this field shall indicate usage defined by a vendor.

387 For slot 0, at least one of `KeyExUse` , `ChallengeUse` , `MeasurementUse` , and `EndpointInfoUse` shall be set. The corresponding capability bits shall be set appropriately.

388 10.9 GET_CERTIFICATE request and CERTIFICATE response messages

389 This request message shall retrieve the certificate chain from the specified slot number.

390 Table 42 — GET_CERTIFICATE request message format shows the `GET_CERTIFICATE` request message format.

391 GET_CERTIFICATE request attributes shows the `GET_CERTIFICATE` request attributes.

392 Table 44 — Successful CERTIFICATE response message format shows the `CERTIFICATE` response message format.

393 Table 45 — CERTIFICATE response attributes shows the `CERTIFICATE` response attributes.

394 The Requester sends one or more `GET_CERTIFICATE` requests to retrieve the certificate chain of the Responder.

395 **Table 42 — GET_CERTIFICATE request message format**

Byte offset	Field	Size (bytes)	Description
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	RequestResponseCode	1	Shall be `0x82` = `GET_CERTIFICATE`. See Table 4 — SPDM request codes.
2	Param1	1	Bit [7]. A Requester can use the `LargeOffset` and `LargeLength` fields when the Responder's `LARGE_CERT_CAP=01b`. This field shall be `0b` if the Responder's `LARGE_CERT_CAP` field is `0b`. If a Requester uses `Offset` and `Length` fields although the certificate chain size cannot be represented by these fields, the Responder shall send an `Error` message with `ErrorCode=CertChainTooLarge`. This request shall use: • `Offset` and `Length` fields when this field is set to `0b`. • `LargeOffset` and `LargeLength` with `Offset` and `Length` become reserved when this field is set to `1b`. Bit [6:4]. Reserved Bit [3:0]. Shall be the `SlotID`. Slot number of the Responder certificate chain to read. The value in this field shall be between 0 and 7 inclusive.
3	Param2	1	Request attributes. See GET_CERTIFICATE request attributes.
4	Offset	2	Shall be the offset in bytes from the start of the certificate chain to where the read request message begins. The Responder shall send its certificate chain starting from this offset. For the first `GET_CERTIFICATE` request for a given slot, the Requester shall set this field to 0. For subsequent requests, `Offset` is set to the next portion of the certificate in that slot. This field is reserved if `Param1 Bit [7]` = `1b`.
6	Length	2	Shall be the length of certificate chain data, in bytes, to be returned in the corresponding response. This field is an unsigned 16-bit integer. This field is reserved if `Param1 Bit [7]` = `1b`.

Byte offset	Field	Size (bytes)	Description
8	LargeOffset	4 or 0	Shall be the offset in bytes from the start of the large certificate chain to where the read request message begins. The Responder shall send its large certificate chain starting from this offset. For the first `GET_CERTIFICATE` request for a given slot, the Requester shall set this field to 0. For subsequent requests, `LargeOffset` is set to the next portion of the certificate in that slot. If `Param1 Bit[7]` = `0b`, then this field shall be absent.
12	LargeLength	4 or 0	Shall be the length of large certificate chain data, in bytes, to be returned in the corresponding response. This field is an unsigned 32-bit integer. If `Param1 Bit[7]` = `0b`, then this field shall be absent.

396 **Table 43 — GET_CERTIFICATE request attributes**

Bit offset	Field	Description
0	SlotSizeRequested	When `SlotSizeRequested=1b` in the `GET_CERTIFICATE` request, the Responder shall return the number of bytes available for certificate chain storage in the: • `RemainderLength` field of the response if `Param1 Bit[7]=0b` in the `GET_CERTIFICATE` request. • `LargeRemainderLength` field of the response if `Param1 Bit[7]=1b` in the `GET_CERTIFICATE` request. When `SlotSizeRequested=1b`, the `Offset`, `Length`, `LargeOffset`, and `LargeLength` fields in the GET_CERTIFICATE request shall be ignored by the Responder.
[7:1]	Reserved	Reserved.

397 **Table 44 — Successful CERTIFICATE response message format**

Byte offset	Field	Size (bytes)	Description
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	RequestResponseCode	1	Shall be `0x02` = `CERTIFICATE`. See Table 5 — SPDM response codes.

Byte offset	Field	Size (bytes)	Description
2	Param1	1	Bit [7]. This field shall be the same as the Requester's `Param1 Bit[7]`. • `0b` : use `PortionLength` and `RemainderLength` fields. • `1b` : use `LargePortionLength` and `LargeRemainderLength`. `PortionLength` and `RemainderLength` become reserved. Bit [6:4]. Reserved Bit [3:0]. Shall be the `SlotID`. Slot number of the certificate chain returned.
3	Param2	1	The format of this field shall be the format that Table 45 — CERTIFICATE response attributes defines.
4	PortionLength	2	Shall be the number of bytes of this portion of the certificate chain. This should be less than or equal to `Length` received as part of the request. For example, the Responder might set this field to a value less than `Length` received as part of the request due to limitations on the transmit buffer of the Responder. If the requested `Length` field is `0` then this field shall be set to `0`. If `SlotSizeRequested=1b` in the request, this field shall be set to zero. This field is reserved if `Param1 Bit[7]` = `1b`.
6	RemainderLength	2	Shall be the number of bytes of the certificate chain that have not been sent yet, after the current response. For the last response, this field shall be 0 as an indication to the Requester that the entire certificate chain has been sent. If the requested `Length` field is `0` and `SlotSizeRequested=0b` in the request, then this field shall return the actual size of the certificate chain in the slot. See Table 43 — GET_CERTIFICATE request attributes for more detail. This field is reserved if `Param1 Bit[7]` = `1b`.
8	CertChain	`PortionLength` or 0	Shall be the requested contents of the target certificate chain, as described in Certificates and certificate chains. If `SlotSizeRequested=1b` in the request, this field shall be absent. If the requested `Length` field is `0`, then this field shall be absent. If `Param1 Bit[7]` = `1b`, then this field shall be absent.

Byte offset	Field	Size (bytes)	Description
8	LargePortionLength	4 or 0	Shall be the number of bytes of this portion of the large certificate chain. This should be less than or equal to `LargeLength` received as part of the request. For example, the Responder might set this field to a value less than `LargeLength` received as part of the request due to limitations on the transmit buffer of the Responder. If the requested `LargeLength` field is `0`, then this field shall be absent. If `SlotSizeRequested=1b` in the request, this field shall be set to zero. If `Param1 Bit[7]` = `0b`, then this field shall be absent.
12	LargeRemainderLength	4 or 0	Shall be the number of bytes of the large certificate chain that have not been sent yet, after the current response. For the last response, this field shall be 0 as an indication to the Requester that the entire large certificate chain has been sent. If the requested `LargeLength` field is `0` and `SlotSizeRequested=0b` in the request, then this field shall return the actual size of the large certificate chain in the slot. See Table 43 — GET_CERTIFICATE request attributes for more detail. If `Param1 Bit[7]` = `0b`, then this field shall be absent.
16	LargeCertChain	`LargePortionLength` or 0	Shall be the requested contents of the target large certificate chain, as described in Certificates and certificate chains. If `SlotSizeRequested=1b` in the request, this field shall be absent. If the requested `LargeLength` field is `0`, then this field shall be absent. If `Param1 Bit[7]` = `0b`, then this field shall be absent.

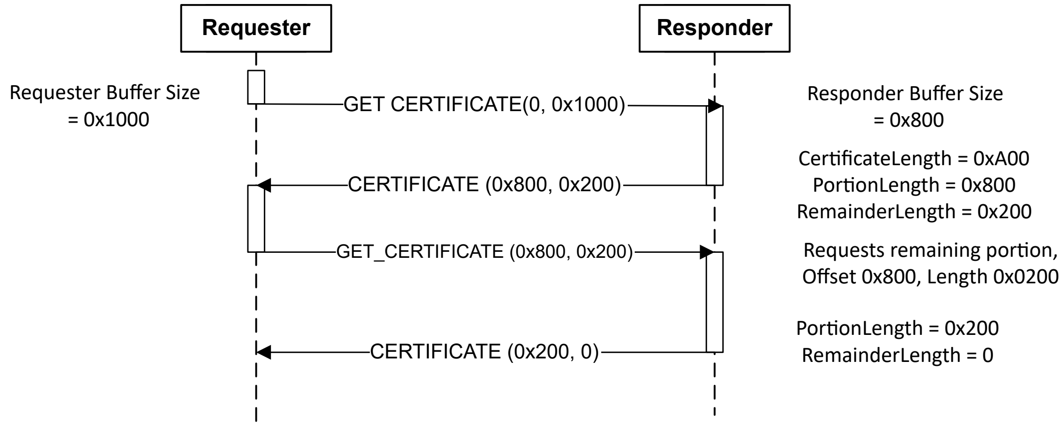
398 **Table 45 — CERTIFICATE response attributes**

Bit offset	Field	Description
[2:0]	CertificateInfo	The value of this field shall be the certificate model of the slot. The format of this field shall be the format of the `CertModel` field that the certificate info table defines.
All other bits	Reserved	Reserved.

399 Figure 9 — Responder cannot return full length data flow shows the high-level request-response message flow when the Responder cannot return the entire data requested by the Requester in the first response.

400 **Figure 9 — Responder cannot return full length data flow**

401



402 Endpoints that support the large SPDM message transfer mechanism message set shall use the large SPDM message transfer mechanism messages to manage the transfer of the requested certificate chain when the `CERTIFICATE` response is larger than either the `DataTransferSize` of the Requester or the transmit buffer of the Responder. Specifically:

- The Requester sets `Param1 Bit[7]` to `0b`, `Offset` to `0`, and `Length` to `0xFFFF` in the `GET_CERTIFICATE` request, the Responder shall set `PortionLength` equal to the size of the complete certificate chain stored in the requested slot, shall set `RemainderLength` to `0`, and shall store the contents of the complete certificate chain in `CertChain` in the `CERTIFICATE` response. Then the Responder shall fragment and return this response message in chunks, as per the clauses presented in `CHUNK_GET` request and `CHUNK_RESPONSE` response message. In this case, the Responder shall not return a partial certificate chain.
- The Requester sets `Param1 Bit[7]` to `1b`, `LargeOffset` to `0`, and `LargeLength` to `0xFFFFFFFF` in the `GET_CERTIFICATE` request, the Responder shall set `LargePortionLength` equal to the size of the complete certificate chain stored in the requested slot, shall set `LargeRemainderLength` to `0`, and shall store the contents of the complete certificate chain in `LargeCertChain` in the `CERTIFICATE` response. Then the Responder shall fragment and return this response message in chunks, as per the clauses presented in `CHUNK_GET` request and `CHUNK_RESPONSE` response message. In this case, the Responder shall not return a partial certificate chain.

403 By setting `SlotSizeRequested=1b` in the request attributes, the Requester can query the size of the Responder's certificate slot. The Requester should query the slot size before any action that uses slot storage, because the Responder might change the value of the slot size based on other actions.

404

10.9.1 Mutual authentication requirements for GET_CERTIFICATE and CERTIFICATE messages

405 If the Requester supports mutual authentication, the requirements placed on the Responder in GET_CERTIFICATE request and CERTIFICATE response messages clause shall also apply to the Requester. If the Responder supports mutual authentication, the requirements placed on the Requester in the GET_CERTIFICATE request and

CERTIFICATE response messages clauses shall also apply to the Responder. The preceding two sentences essentially describe a role reversal.

406
10.9.2 SPDM certificate requirements and recommendations

407 This specification defines a number of X.509 v3 required and optional fields for compliant SPDM certificates. SPDM certificates also adhere to the requirements as RFC 5280 defines. Unless stated otherwise, the following clauses apply to those certificates in the chain that are specific to a device instance, that is, the leaf certificate in the `DeviceCert` model or the `DeviceCert`, all intermediate `AliasCert`s, and the leaf certificate in the `AliasCert` model. See identity provisioning.

408 In addition, the `Subject Alternative Name` certificate extension `otherName` field is recommended for providing device information. See the Definition of otherName using the DMTF OID.

409 In Table 46 — Field requirements, the requirements columns define the requirement for the corresponding certificate models. In these columns, the corresponding field with a value of "Mandatory" shall be present in the leaf certificate. Likewise, the corresponding field with a value of "Optional" can be present or absent in the leaf certificate. As a note, this table reflects the minimum requirements from the perspective of this specification. The vendor, users of the SPDM endpoint, and other standards such as RFC 5280 can place additional or more-restrictive requirements.

410 **Table 46 — Field requirements**

Field	`DeviceCert` / `AliasCert` Requirements	`GenericCert` Requirements	Description
Basic Constraints	Mandatory	Mandatory	The CA value shall be `FALSE`.
Version	Mandatory	Mandatory	The version of the encoded certificate shall be present and shall be `3` (encoded as value `2`).
Serial Number	Mandatory	Mandatory	The CA-assigned serial number shall be present with a positive integer value.

Field	`DeviceCert` / `AliasCert` Requirements	`GenericCert` Requirements	Description
Signature Algorithm	Mandatory	Optional	If present, the `Signature` algorithm that the CA uses shall be present.
Issuer	Mandatory	Optional	If present, the CA distinguished name shall be specified.
Subject Name	Mandatory	Optional	If present, the subject name shall be present and shall represent the distinguished name associated with the leaf certificate.
Validity	Mandatory	Optional	If present, see Certificate validity details, and RFC 5280.
Subject Public Key Info	Mandatory	Mandatory	The device public key and the algorithm shall be present.
Key Usage	Mandatory	Optional	If present, the key usage bit for digital signature shall be set.

411 For intermediate and root certificates the basic constraints field shall be present and the CA value shall be `TRUE`.

412 **Table 47 — Optional fields**

Field	Description
Subject Alternative Name otherName	In some cases, it might be desirable to provide device-specific information as part of the leaf certificate. DMTF chose the `otherName` field with a specific format to represent the device information. The use of the `otherName` field also provides flexibility for other alliances to provide device-specific information as part of the leaf certificate. See the Definition of otherName using the DMTF OID. Note that `otherName` field formats specified by other standards are permissible in the certificate.
Extended Key Usage (EKU)	If present in a certificate, the Extended Key Usage extension indicates one or more purposes for which the public key should be used. See Extended Key Usage authentication OIDs.
SPDM Non-critical Certificate Extension	If present in a certificate, the SPDM Non-critical Certificate Extension indicates one or more non-critical OIDs associated with the certificate. See SPDM Non-Critical Certificate Extension OID.

413 **Certificate validity details**

414 As per RFC 5280, the certificate validity period is the time interval during which the CA warrants that it will maintain information about the status of the certificate. The field is represented as an ASN.1-encoded SEQUENCE of two dates: the date when the certificate validity period begins (`notBefore`) and the date when the certificate validity period ends (`notAfter`).

415 For a leaf certificate whose chain is stored in Slot 0, the `notBefore` date should be the date of certificate creation, and the `notAfter` date should be set to GeneralizedTime value `99991231235959Z` . Immutable leaf certificates' `notAfter` dates should be set appropriately to ensure that the leaf certificate will not expire during the practical lifetime of the device.

416 For leaf certificates whose chains are stored in Slots 1-7, the `notBefore` date should be the date of certificate creation. The `notAfter` date can be set according to end user requirements, including values that will result in certificate expiration and thus require certificate renewal and device recertification during the lifetime of the device.

417 Definition of otherName using the DMTF OID shows the definition of otherName using the DMTF OID:

418 **Definition of otherName using the DMTF OID**

```
id-DMTF OBJECT IDENTIFIER ::= { 1 3 6 1 4 1 412 }

id-DMTF-spdm OBJECT IDENTIFIER ::= { id-DMTF 274 }

DMTFOtherName ::= SEQUENCE {
    type-id   DMTF-oid
    value [0] EXPLICIT ub-DMTF-device-info
}
-- OID for DMTF device info --
id-DMTF-device-info  OBJECT IDENTIFIER ::= { 1 3 6 1 4 1 412 274 1 }
DMTF-oid                               ::= OBJECT IDENTIFIER (id-DMTF-device-info)
```

```
-- All printable characters except ":" --
DMTF-device-string                      ::= UTF8String (ALL EXCEPT ":")

-- Device Manufacturer --
DMTF-manufacturer                       ::= DMTF-device-string

-- Device Product --
DMTF-product                            ::= DMTF-device-string

-- Device Serial Number --
DMTF-serialNumber                       ::= DMTF-device-string

-- Device information string  --
ub-DMTF-device-info                     ::= UTF8String({DMTF-manufacturer":"DMTF-product":"DMTF-
         serialNumber})
```

419 The Leaf certificate example shows an example leaf certificate.

420 10.9.2.1 Extended Key Usage authentication OIDs

421 The following Extended Key Usage purposes are defined for SPDM certificate authentication:

- SPDM Responder Authentication { id-DMTF-spdm 3 }: The presence of this OID shall indicate that a leaf certificate can be used for Responder authentication purposes.
- SPDM Requester Authentication { id-DMTF-spdm 4 }: The presence of this OID shall indicate that a leaf certificate can be used for Requester authentication purposes.

422 The presence of both OIDs shall indicate that the leaf certificate can be used for both Requester and Responder authentication purposes. If present, these OIDs shall appear in the leaf certificate.

423 A Responder device that supports mutual authentication should include the `SPDM Responder Authentication` OID in the Extended Key Usage field of its leaf certificate. A Requester device that supports mutual authentication should include the `SPDM Requester Authentication` OID in the Extended Key Usage field of its leaf certificate. Note that alternate OIDs specified by other standards are permissible in the certificate.

424 10.9.2.2 SPDM Non-Critical Certificate Extension OID

425 The `id-DMTF-spdm-extension` OID is a container of non-critical SPDM OIDs and their corresponding values. The OID value for `id-DMTF-spdm-extension` shall be { id-DMTF-spdm 6 }. Furthermore, this OID is a Certificate Extension as defined in RFC 5280, and its encoding shall follow the `Extension` syntax also defined in RFC 5280. The `Extension` syntax defines three parameters: `extnID`, `critical`, and `extnValue`. The values of these three parameters for `id-DMTF-spdm-extension` shall be the DER encoding of the ASN.1 value as the DMTF SPDM Extension Format defines.

426 **Definition of DMTF SPDM Extension Format**

```
id-DMTF-spdm-extension Extension ::=
{
    extnID       { id-DMTF-spdm 6 }
    critical     FALSE
    extnValue    id-spdm-cert-oids
}

id-spdm-cert-oids ::= SEQUENCE SIZE (1..MAX) OF id-spdm-cert-oid

id-spdm-cert-oid ::=  SEQUENCE
{
    spdmOID              OBJECT IDENTIFIER
    spdmOIDdefinition    OCTET STRING OPTIONAL
}
```

427 The `spdmOID` field shall contain an OID defined in this specification. Only designated OIDs, permitted by this specification, shall be allowed in `spdmOID`. The `spdmOIDdefinition` field shall be a DER encoding of the ASN.1 value of the definition indicated by `spdmOID`.

428 These clauses describe the definitions and formats of the SPDM OIDs contained in `id-DMTF-spdm-extension`. If present, these OIDs shall only be contained in `id-DMTF-spdm-extension`.

429 10.9.2.2.1 Hardware identity OID

430 The `id-DMTF-hardware-identity` OID is defined to help identify the hardware identity certificate in a chain regardless of the certificate chain model used (`DeviceCert` or `AliasCert`). If the `AliasCert` model is used, this OID shall not be present in any alias certificates in the chain. The `id-DMTF-hardware-identity` OID shall have a format as Hardware identity OID format defines.

431 **Hardware identity OID format**

```
id-DMTF-hardware-identity id-spdm-cert-oid :: = {
    spdmOID           { id-DMTF-spdm 2 }
    spdmOIDdefinition ABSENT
}
```

432 10.9.2.2.2 Mutable certificate OID

433 Mutable certificates may include the `id-DMTF-mutable-certificate` OID to identify them as mutable. If used, this OID shall be present in all mutable certificates in the chain. The `id-DMTF-mutable-certificate` OID shall have a format as Mutable certificate OID format defines.

434 **Mutable certificate OID format**

```
id-DMTF-mutable-certificate id-spdm-cert-oid ::= {
    spdmOID           { id-DMTF-spdm 5 }
    spdmOIDdefinition ABSENT
}
```

435 10.10 CHALLENGE request and CHALLENGE_AUTH response messages

436 This request message shall authenticate a Responder through the challenge-response protocol.

437 Table 48 — CHALLENGE request message format shows the `CHALLENGE` request message format.

438 Table 49 — Successful CHALLENGE_AUTH response message format shows the `CHALLENGE_AUTH` response message format.

439 Table 50 — CHALLENGE_AUTH response attribute shows the `CHALLENGE_AUTH` response attribute.

440 **Table 48 — CHALLENGE request message format**

Byte offset	Field	Size (bytes)	Description
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	RequestResponseCode	1	Shall be `0x83` = `CHALLENGE`. See Table 4 — SPDM request codes.
2	Param1	1	Shall be the `SlotID`. Slot number of the Responder certificate chain that shall be used for authentication. If the public key of the Responder was provisioned to the Requester in a trusted environment, the value in this field shall be `0xFF`; otherwise it shall be between 0 and 7 inclusive.
3	Param2	1	Shall be the type of measurement summary hash requested: • `0x0`. No measurement summary hash requested. • `0x1`. *TCB* measurements only. • `0xFF`. All measurements. • All other values reserved. If a Responder does not support measurements (`MEAS_CAP=00b` in its `CAPABILITIES` response), the Requester shall set this value to `0x0`.
4	Nonce	32	The Requester should choose a random value.

Byte offset	Field	Size (bytes)	Description
36	Context	8	The Requester can include application-specific information in Context. The Requester should fill this field with zeros if it has no context to provide.

441 **Table 49 — Successful CHALLENGE_AUTH response message format**

Byte offset	Field	Size (bytes)	Description
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	RequestResponseCode	1	Shall be `0x03` = `CHALLENGE_AUTH`. See Table 5 — SPDM response codes.
2	Param1	1	Shall be the Response Attribute Field. See Table 50 — CHALLENGE_AUTH response attribute.
3	Param2	1	Shall be the slot mask. The bit in position K of this byte shall be set to `1b` if and only if slot number K is in the "Exists with key and cert" state, that is, slot K has a key provisioned and contains a certificate chain (Bit 0 is the least significant bit of the byte). This field is reserved if the public key of the Responder was provisioned to the Requester in a trusted environment.
4	CertChainHash	`H`	Shall be either the hash of the certificate chain as Table 37 — Certificate chain format describes or, if the public key of the Responder was provisioned to the Requester in a trusted environment, the public key used for authentication. The Requester can use this value to check that the certificate chain or public key matches the one requested. This field shall be in hash byte order.
4 + `H`	Nonce	32	Shall be the Responder-selected random value.

Byte offset	Field	Size (bytes)	Description
36 + `H`	MeasurementSummaryHash	`MSHLength` = `H` or 0	If the Responder does not support measurements (`MEAS_CAP=00b` in its `CAPABILITIES` response) or if the requested `Param2` = `0x0` , this field shall be absent. If the requested `Param2` = `0x1` , this field shall be the combined hash of measurements of all measurable components considered to be in the TCB required to generate this response, computed as `hash(Concatenate(MeasurementBlock[0], MeasurementBlock[1], ...))` , where `MeasurementBlock[x]` denotes a measurement of an element in the TCB and `hash` is the negotiated base hashing algorithm. Measurements are concatenated in ascending order based on their measurement index as Table 57 — Measurement block format describes. If the requested `Param2` = `0x1` and if there are no measurable components in the TCB required to generate this response, this field shall be `0` . If the requested `Param2` = `0xFF` , this field shall be computed as `hash(Concatenate(MeasurementBlock[0], MeasurementBlock[1], ..., MeasurementBlock[n]))` of all supported measurement blocks available in the measurement index range `0x01` - `0xFE` , concatenated in ascending index order. Indices with no associated measurements shall not be included in the hash calculation. See the Measurement index assignments clause. If the Responder supports both raw bit stream and digest representations for a given measurement index, the Responder shall use the digest form. This field shall be in hash byte order.
36 + `H` + `MSHLength`	OpaqueDataLength	2	Shall be the size of the `OpaqueData` field that follows in bytes. The value should not be greater than 1024 bytes. Shall be `0` if no `OpaqueData` is provided.
38 + `H` + `MSHLength`	OpaqueData	`OpaqueDataLength`	The Responder can include Responder-specific information and/or information that its transport defines. If present, this field shall conform to the selected opaque data format in `OtherParamsSelection` .

Byte offset	Field	Size (bytes)	Description
38 + `H` + `MSHLength` + OpaqueDataLength	RequesterContext	8	This field shall be identical to the `Context` field of the corresponding request message.
46 + `H` + `MSHLength` + OpaqueDataLength	Signature	`SigLen`	Shall be the Responder's signature. `SigLen` is the size of the asymmetric-signing algorithm output that the Responder selected in the last `ALGORITHMS` response message to the Requester. The CHALLENGE_AUTH signature generation and CHALLENGE_AUTH signature verification clauses, respectively, define the signature generation and verification processes.

442 **Table 50 — CHALLENGE_AUTH response attribute**

Bit offset	Field	Description
[3:0]	SlotID	Shall contain the `SlotID` in the `Param1` field of the corresponding `CHALLENGE` request. If the Responder's public key was provisioned to the Requester previously, this field shall be `0xF` . The Requester can use this value to check that the certificate matched what was requested.
[6:4]	Reserved	Reserved.
7	**DEPRECATED:** BasicMutAuthReq	**DEPRECATED:** When mutual authentication is supported by both Responder and Requester, the Responder shall set this bit to indicate that the Responder wants to authenticate the identity of the Requester using the basic mutual authentication flow. The Requester shall not set this bit in a basic mutual authentication flow. See Basic mutual authentication flow. If mutual authentication is not supported, this bit shall be zero.

443
10.10.1 CHALLENGE_AUTH signature generation

444 To complete the `CHALLENGE_AUTH` signature generation process, the Responder shall complete these steps:

445 1. The Responder shall construct M1, and the Requester shall construct M2 message transcripts. For Responder authentication, see the request ordering and message transcript computation rules for M1/M2 table. For Requester authentication in the mutual authentication scenario, see the Mutual authentication message transcript clause.

446 ◦ If a response contains `ErrorCode=ResponseNotReady` :

447 Concatenation function is performed on the contents of both the original request and the successful response received during `RESPOND_IF_READY` . Neither the error response (`ResponseNotReady`) nor the `RESPOND_IF_READY` request shall be included in M1/M2.

448 ◦ If a response contains an `ErrorCode` other than `ResponseNotReady` :

449 No concatenation function is performed on the contents of both the original request and response.

450 2. The Responder shall generate:

```
Signature = SPDMsign(PrivKey, M1, "challenge_auth signing");
```

451 where:

- `SPDMsign` is described in Signature generation.
- `PrivKey` shall be the private key associated with the leaf certificate of the Responder in `slot=Param1` of the `CHALLENGE` request message. If the public key of the Responder was provisioned to the Requester, then `PrivKey` shall be the associated private key.

452 10.10.2 CHALLENGE_AUTH signature verification

453 Any modifications to the previous request messages or to the corresponding response messages by an active person-in-the-middle adversary or media error will result in `M2 != M1` and thus lead to verification failure.

454 To complete the `CHALLENGE_AUTH` signature verification process, the Requester shall complete this step:

455 1. The Requester shall perform:

```
result = SPDMsignatureVerify(PubKey, Signature, M2, "challenge_auth signing");
```

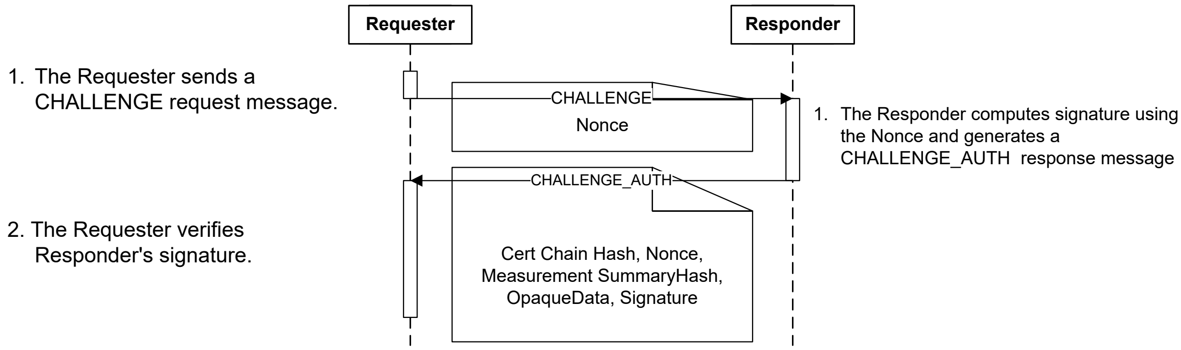
456 where:

- `SPDMsignatureVerify` is described in Signature verification. If `result` is `success`, the verification was successful.
- `PubKey` shall be the public key associated with the leaf certificate of the Responder with `slot=Param1` of the `CHALLENGE` request message. If the public key of the Responder was provisioned to the Requester, `PubKey` is the provisioned public key.

457 Figure 10 — Responder authentication: Runtime challenge-response flow shows the high-level request-response message flow and sequence for the authentication of the Responder for runtime challenge-response.

458 **Figure 10 — Responder authentication: Runtime challenge-response flow**

459



460 10.10.2.1 Request ordering and message transcript computation rules for M1 and M2

461 This clause applies to Responder-only authentication.

462 Table 51 — Request ordering and message transcript computation rules for M1/M2 defines how the message transcript is constructed for M1 and M2, which are used in signature calculation and verification in the `CHALLENGE_AUTH` response message.

463 The possible request orderings leading up to and including `CHALLENGE` shall be:

- `GET_VERSION` , `GET_CAPABILITIES` , `NEGOTIATE_ALGORITHMS` , `GET_DIGESTS` , `GET_CERTIFICATE` , `CHALLENGE` (A1, B1, C1)
- `GET_VERSION` , `GET_CAPABILITIES` , `NEGOTIATE_ALGORITHMS` , `GET_DIGESTS` , `CHALLENGE` (A1, B3, C1)
- `GET_VERSION` , `GET_CAPABILITIES` , `NEGOTIATE_ALGORITHMS` , `GET_CERTIFICATE` , `CHALLENGE` (A1, B4, C1)
- `GET_VERSION` , `GET_CAPABILITIES` , `NEGOTIATE_ALGORITHMS` , `CHALLENGE` (A1, B2, C1)
- `GET_DIGESTS` , `GET_CERTIFICATE` , `CHALLENGE` (A2, B1, C1)
- `GET_DIGESTS` , `CHALLENGE` (A2, B3, C1)
- `GET_CERTIFICATE` , `CHALLENGE` (A2, B4, C1)
- `CHALLENGE` (A2, B2, C1)

464 Immediately after Reset, M1 and M2 shall be `null` .

465 After the Requester receives a successful `CHALLENGE_AUTH` response or the Requester sends a `GET_MEASUREMENTS` request, M1 and M2 shall be set to `null` . If a `Negotiated State` has been established, this will remain intact.

466 If a Requester sends a `GET_VERSION` message, the Requester and Responder shall set M1 and M2 to `null` , clear all `Negotiated State` and recommence construction of M1 and M2 starting with the new `GET_VERSION` message.

467 For additional rules, see general ordering rules.

468 **Table 51 — Request ordering and message transcript computation rules for M1/M2**

Requests	Implementation conditions	M1/M2=Concatenate(A, B, C)
Initial value	N/A	`M1/M2=null`
`GET_VERSION` issued	Requester issues this request to allow the Requester and Responder to determine an agreed-upon `Negotiated State`. Also issued when the Requester detects an out-of-sync condition, or when the signature verification fails, or when the Responder provides an unexpected error response.	`M1/M2=null`
`GET_VERSION`, `GET_CAPABILITIES`, `NEGOTIATE_ALGORITHMS` issued	Requester shall always issue these requests in this order.	`A1=` VCA
`GET_VERSION`, `GET_CAPABILITIES`, `NEGOTIATE_ALGORITHMS` skipped	After M1/M2 were re-initialized to `null` due to a Reset or a completed `CHALLENGE_AUTH` response, Requester skipped these requests if the Responder had previously indicated `CACHE_CAP=1`. In this case, the Requester and Responder shall proceed with the previously determined `Negotiated State`. These requests and responses are still required for M1/M2 construction.	`A2=` VCA
`GET_DIGESTS`, `GET_CERTIFICATE` issued	After `NEGOTIATE_ALGORITHMS` request completion or after M1/M2 were re-initialized to `null` due to a Reset or a completed `CHALLENGE_AUTH` response, Requester issued these requests in this order if it had skipped the previous three requests.	`B1=Concatenate(GET_DIGESTS, DIGESTS, GET_CERTIFICATE, CERTIFICATE)`
`GET_DIGESTS`, `GET_CERTIFICATE` skipped	After M1/M2 were re-initialized to `null` due to a Reset or a completed `CHALLENGE_AUTH` response, Requester skipped these requests because it could use previously cached certificate information.	`B2=null`
`GET_DIGESTS` issued, `GET_CERTIFICATE` skipped	After M1/M2 were re-initialized to `null` due to a Reset or a completed `CHALLENGE_AUTH` response, Requester skipped the `GET_CERTIFICATE` request because it could use the previously cached `CERTIFICATE` response.	`B3=Concatenate(GET_DIGESTS, DIGESTS)`
`GET_DIGESTS` skipped, `GET_CERTIFICATE` issued	After M1/M2 were re-initialized to `null` due to a Reset or a completed `CHALLENGE_AUTH` response, Requester skipped the `GET_DIGESTS` request because it could use the previously cached `CERTIFICATE` response to make a byte-by-byte comparison.	`B4=Concatenate(GET_CERTIFICATE, CERTIFICATE)`
`CHALLENGE` issued	Requester issued this request to complete security verification of current requests and responses. The Signature bytes of `CHALLENGE_AUTH` shall not be included in C.	`C1=Concatenate(CHALLENGE, CHALLENGE_AUTH(excluding Signature))`. See Table 48 — CHALLENGE request message format.

Requests	Implementation conditions	M1/M2=Concatenate(A, B, C)
`CHALLENGE` completion	Completion of `CHALLENGE` sets M1/M2 to `null`.	`M1/M2=null`
Other issued	If the Requester issued commands other than `GET_DIGESTS`, `GET_CERTIFICATE`, and `CHALLENGE` and skipped `CHALLENGE` completion, then M1/M2 are set to `null`.	`M1/M2=null`

469 The Basic mutual authentication flow is DEPRECATED. Implementations should use session-based mutual authentication as Figure 21 — Session-based mutual authentication example shows or optimized session-based mutual authentication as Figure 22 — Optimized session-based mutual authentication example shows.

---

470 DEPRECATED

471 10.10.3 Basic mutual authentication

472 Unless otherwise stated, if the Requester supports mutual authentication, the requirements placed on the Responder in the CHALLENGE request and CHALLENGE_AUTH response messages clause shall also apply to the Requester. Unless otherwise stated, if the Responder supports mutual authentication, the requirements placed on the Requester in the CHALLENGE request and CHALLENGE_AUTH response messages clause shall also apply to the Responder. The preceding two sentences essentially describe a role reversal, unless otherwise stated.

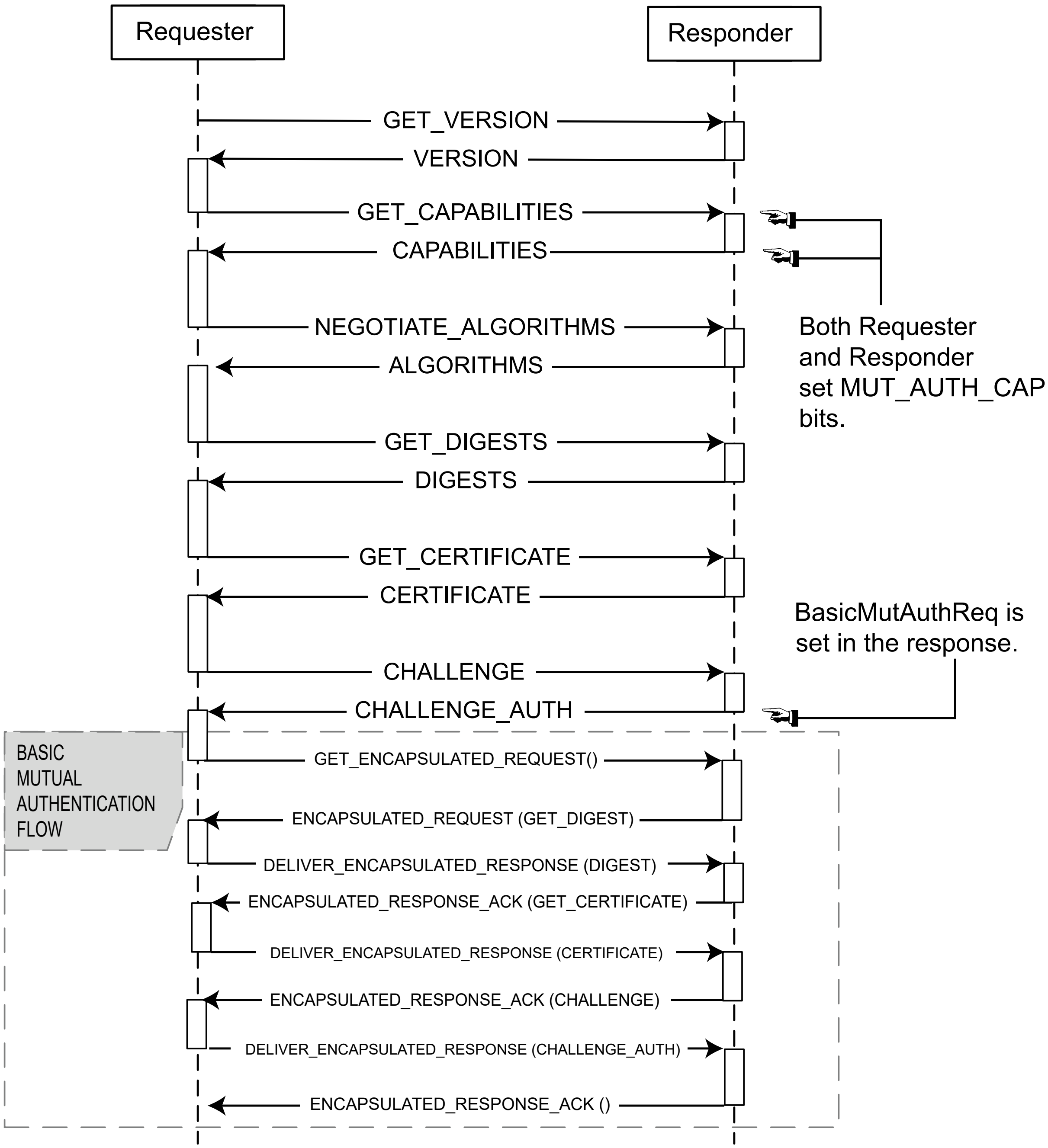
473 The basic mutual authentication flow shall start when the Requester successfully receives a `CHALLENGE_AUTH` with **BasicMutAuthReq** set. This flow shall utilize message encapsulation as described in the GET_ENCAPSULATED_REQUEST request and ENCAPSULATED_REQUEST response messages clauses to retrieve request messages. The basic mutual authentication flow shall end when the encapsulated request flow ends.

474 This flow shall only allow `GET_DIGESTS`, `GET_CERTIFICATE`, `CHALLENGE`, and their corresponding responses to be encapsulated. If other requests are encapsulated, the Requester can send an `ERROR` message of `ErrorCode=UnexpectedRequest` and shall terminate the flow.

475 Figure 11 — Mutual authentication basic flow illustrates, as an example, the basic mutual authentication flow.

476 **Figure 11 — Mutual authentication basic flow**

477



478 **10.10.3.1 Mutual authentication message transcript**

479 This clause applies to the Responder authenticating the Requester in a basic mutual authentication scenario.

480 Table 52 — Basic mutual authentication message transcript defines how the message transcript is constructed for

M1 and M2, which are used in signature calculation and verification in the `CHALLENGE_AUTH` response message when the Responder authenticates the Requester.

481 The possible request orderings for the basic mutual authentication flow shall be one of the following (the Flow ID is in parenthesis):

- `GET_DIGESTS` , `GET_CERTIFICATE` , `CHALLENGE` (*BMAF0*)
- `GET_DIGESTS` , `CHALLENGE` (*BMAF1*)
- `GET_CERTIFICATE` , `CHALLENGE` (*BMAF2*)
- `CHALLENGE` (*BMAF3*)

482 When the basic mutual authentication flow starts, that is, when `GET_ENCAPSULATED_REQUEST` is issued, M1 and M2 shall be set to `null` .

483 **Table 52 — Basic mutual authentication message transcript**

Flow ID	M1/M2
BMAF0	Concatenate(`VCA` , `GET_DIGESTS` , `DIGESTS` , `GET_CERTIFICATE` , `CERTIFICATE` , `CHALLENGE` , `CHALLENGE_AUTH` without the signature)
BMAF1	Concatenate(`VCA` , `GET_DIGESTS` , `DIGESTS` , `CHALLENGE` , `CHALLENGE_AUTH` without the signature)
BMAF2	Concatenate(`VCA` , `GET_CERTIFICATE` , `CERTIFICATE` , `CHALLENGE` , `CHALLENGE_AUTH` without the signature)
BMAF3	Concatenate(`VCA` , `CHALLENGE` , `CHALLENGE_AUTH` without the signature)

484 For `GET_CERTIFICATE` and `CERTIFICATE` , these messages might need to be issued multiple times to retrieve the entire certificate chain. Thus, each instance of the request and response shall be part of M1/M2 in the order that they are issued.

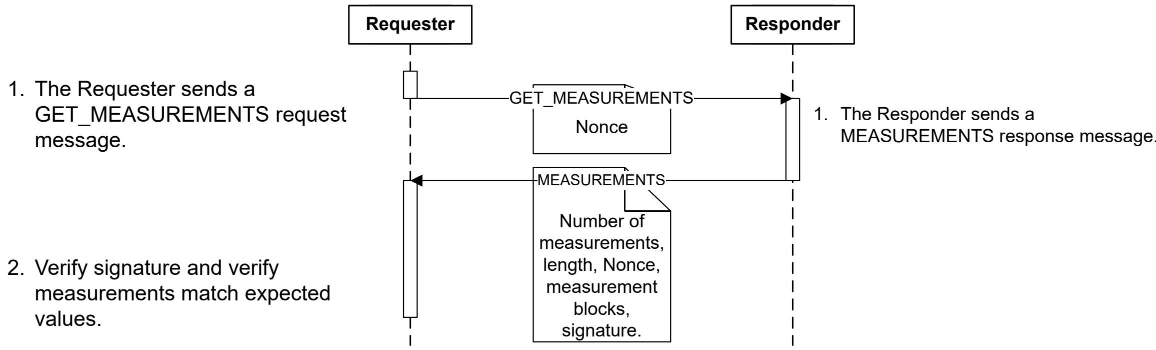
485 DEPRECATED

486 10.11 Firmware and other measurements

487 This clause describes request messages and response messages associated with endpoint measurement. All request messages in this clause shall be supported by an endpoint that returns `MEAS_CAP=01b` or `MEAS_CAP=10b` in its `CAPABILITIES` response.

488 Figure 12 — Measurement retrieval flow shows the high-level request-response flow and sequence for endpoint measurement. If the `MEAS_FRESH_CAP` bit in the `CAPABILITIES` response message returns 0 and if the Requester requires fresh measurements, the Responder shall be Reset before `GET_MEASUREMENTS` is resent. The mechanisms employed for Resetting the Responder are outside the scope of this specification.

489 **Figure 12 — Measurement retrieval flow**

490 

491
10.12 GET_MEASUREMENTS request and MEASUREMENTS response messages

492 Measurements in SPDM are represented in the form of measurement *blocks*. A measurement block defines the measurement block structure. A device can present measurements of different elements of its internal state, as well as metadata to assist in the attestation of its state via measurements, as separate blocks. The `GET_MEASUREMENTS` request message enables a Requester to query a Responder for the number of individual measurement blocks it supports and request either specific blocks or all available blocks. The `MEASUREMENTS` response message returns the requested blocks. A collection of one or more measurement blocks is called a *measurement record*.

493 Because issuing `GET_MEASUREMENTS` clears the M1/M2 message transcript, it is recommended that a Requester does not send this message until it has received at least one successful `CHALLENGE_AUTH` response message from the Responder. This ensures that the information in message pairs `GET_DIGESTS` / `DIGESTS` and `GET_CERTIFICATE` / `CERTIFICATE` has been authenticated at least once.

494 Table 53 — GET_MEASUREMENTS request message format shows the `GET_MEASUREMENTS` request message format.

495 Table 54 — GET_MEASUREMENTS request attributes shows the `GET_MEASUREMENTS` request message attributes.

496 Table 56 — Successful MEASUREMENTS response message format shows the `MEASUREMENTS` response message format. The measurement blocks in `MeasurementRecord` shall be sorted in ascending order by index.

497 **Table 53 — GET_MEASUREMENTS request message format**

Byte offset	Field	Size (bytes)	Description
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	RequestResponseCode	1	Shall be `0xE0` = `GET_MEASUREMENTS`. See Table 4 — SPDM request codes.

Byte offset	Field	Size (bytes)	Description
2	Param1	1	Shall be Request attributes. See Table 54 — GET_MEASUREMENTS request attributes.
3	Param2	1	Shall be a `Measurement` operation. • A value of `0x0` shall query the Responder for the total number of measurement blocks available. • A value of `0xFF` shall request all measurement blocks. • A value between `0x1` and `0xFE`, inclusive, shall request the Measurement block at the index corresponding to that value.
4	Nonce	`NL` = 32 or 0	The Requester should choose a random value. This field is only present if Bit [0] of `Param1` is `1`. See Table 54 — GET_MEASUREMENTS request attributes.
4 + `NL`	SlotIDParam	`SL` = 1 or 0	This field is only present if Bit [0] of `Param1` is `1`. • Bit [7:4]. Reserved. • Bit [3:0]. Shall be the `SlotID`. Slot number of the Responder certificate chain that shall be used for authenticating the measurement(s). If the Responder's public key was provisioned to the Requester previously, this field shall be `0xF`. See Table 54 — GET_MEASUREMENTS request attributes.
4 + `NL` + `SL`	Context	8	The Requester can include application-specific information in Context. The Requester should fill this field with zeros if it has no context to provide.

498 **Table 54 — GET_MEASUREMENTS request attributes**

Bit offset	Field	Description
0	SignatureRequested	If the Responder can generate a signature (`MEAS_CAP` is `10b` in the `CAPABILITIES` response and either `BaseAsymSel` or `ExtAsymSelCount` is non-zero) a value of `1` indicates that a signature on the measurement log is required. The `Nonce` field shall be present in the request when this bit is set. The Responder shall generate and send a signature in the response. A value of `0` indicates that the Requester does not require a signature. The Responder shall not generate a signature in the response. The `Nonce` field shall be absent in the request. For Responders that cannot generate a signature (`MEAS_CAP` is `01b` in the `CAPABILITIES` response or both `BaseAsymSel` and `ExtAsymSelCount` are zero), the Requester shall always use `0` .
1	RawBitStreamRequested	This bit is applicable only if the measurement specification supports only two representations, raw bit stream and digest, such as when `MeasurementSpecification` of the Measurement block format is set to `DMTF` , as Table 57 — Measurement block format describes. If the measurement specification supports other representations, this bit is ignored. If the Responder can return either a raw bit stream or a hash for the requested measurement, value `1` shall request the Responder to return the raw bit stream version of such measurement. If the Responder cannot return a raw bit stream for the measurement (for example, if the raw bit stream contains confidential data that the Responder cannot expose), it shall return the corresponding hash. Another scenario in which the Responder cannot return a raw bit stream is when the `MEASUREMENTS` message is greater than the `MaxSPDMmsgSize` of the Requester. In cases where the Responder cannot return a raw bit stream, the Requester can simply request a digest. Value `0` shall request the Responder to return a hash version of the measurement. If the Responder cannot return a hash of the measurement (for example, if the measurement represents a data structure where a digest is not applicable), it shall return the corresponding raw bit stream.

Bit offset	Field	Description
2	NewMeasurementRequested	If the Responder has pending updates to measurement blocks that have not yet taken effect, then value `1` shall be used to request the Responder to return new values of the measurement blocks at the indices requested in `Param2`. Value `0` shall be used to request the Responder to return the current values of the measurement blocks at the requested indices. If the Responder has no pending updates to the measurement blocks at the requested indices, then the Responder shall return the current values of the measurement blocks, regardless of the value of `NewMeasurementRequested`.
[7:3]	Reserved	Reserved.

499 **Measurement index assignments**

500 This specification imposes no requirements on the scope, type, or format of measurement a device associates with a particular measurement index in the range `0x1` to `0xEF`. As a result, Responders can use the same index to report different types of measurements based on their implementation. If available, a Requester can use a measurement manifest to discover information about the specific measurement types available from a particular Responder and the indices to which they correspond. When measurements follow the DMTF measurement specification format that Table 58 — DMTF measurement specification format describes, a measurement with a `DMTFSpecMeasurementValueType[6:0]` equal to either `0x04` or `0x0A` is the measurement manifest. If a Requester specifies a measurement index that a Responder does not support then the Responder shall respond with an `ERROR` message of `ErrorCode=InvalidRequest`.

501 To aid interoperability, this specification reserves indices `0xF0` to `0xFE` inclusive for specific purposes. If a Responder supports a type of measurement that Table 55 — Measurement index assigned range defines, it shall always assign to it the corresponding index value. A Responder shall not assign indices `0xF0` to `0xFE` to measurements types other than those that Table 55 — Measurement index assigned range defines.

502 **Table 55 — Measurement index assigned range**

Measurement Index	Measurement type	Description
0xF0 - 0xFC	Reserved	Reserved.
0xFD	Measurement manifest	Shall be the metadata on available measurements, as type `DMTFSpecMeasurementValueType[6:0]` = `0x04` or `DMTFSpecMeasurementValueType[6:0]` = `0x0A` defines.
0xFE	Device mode	Shall be structured device mode information, as type `DMTFSpecMeasurementValueType[6:0]` = `0x05` defines.

503 **Table 56 — Successful MEASUREMENTS response message format**

Byte offset	Field	Size (bytes)	Description
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	RequestResponseCode	1	Shall be `0x60` = `MEASUREMENTS` . See Table 5 — SPDM response codes.
2	Param1	1	When `Param2` in the requested measurement operation is `0` , this parameter shall return the total number of measurement indices on the device. Otherwise, this field is reserved.
3	Param2	1	Bit [7:6]. Reserved. Bit [5:4]. Content changed. If this message contains a signature, this field shall indicate if one or more `MeasurementRecord` fields of previous `MEASUREMENTS` responses in the same measurement log have changed. `00b` : The Responder does not detect changes of `MeasurementRecord` fields of previous `MEASUREMENTS` responses in the same measurement log, or this message does not contain a signature. `01b` : The Responder detected that one or more `MeasurementRecord` fields of previous `MEASUREMENTS` responses in the measurement log being signed have changed. The Requester might consider issuing `GET_MEASUREMENTS` again to acquire latest measurements. `10b` : The Responder detected no change in `MeasurementRecord` fields of previous `MEASUREMENTS` responses in the measurement log being signed. `11b` : Reserved. Bit [3:0]. Shall be the `SlotID` . If this message contains a signature, this field shall contain the slot number of the certificate chain specified in the `GET_MEASUREMENTS` request, or `0xF` if the Responder's public key was provisioned to the Requester previously. If this message does not contain a signature, this field shall be set to `0x0` .
4	NumberOfBlocks	1	Shall be the number of measurement blocks in the `MeasurementRecord` . If Param2 in the requested measurement operation is `0` , this field shall be `0` .
5	MeasurementRecordLength	3	Shall be the size of the `MeasurementRecord` in bytes. If `Param2` in the requested measurement operation is `0` , this field shall be `0` .

Byte offset	Field	Size (bytes)	Description
8	MeasurementRecord	`L` = `MeasurementRecordLength`	Shall be the concatenation of all measurement blocks that correspond to the requested Measurement operation. Measurement block defines the measurement block structure.
8 + `L`	Nonce	32	The Responder should choose a random value. This field shall always be present.
40 + `L`	OpaqueDataLength	2	Shall be the size of the `OpaqueData` field that follows in bytes. The value should not be greater than 1024 bytes. Shall be `0` if no `OpaqueData` is provided.
42 + `L`	OpaqueData	`OpaqueDataLength`	The Responder can include Responder-specific information and/or information that its transport defines. If present, this field shall conform to the selected opaque data format in `OtherParamsSelection`.
42 + `L` + `OpaqueDataLength`	RequesterContext	8	This field shall be identical to the `Context` field of the corresponding request message.
50 + `L` + `OpaqueDataLength`	`Signature`	`SigLen`	Shall be `Signature` of the measurement log, excluding the Signature field and signed using the private key associated with the leaf certificate. The Responder shall use the asymmetric signing algorithm it selected during the last `ALGORITHMS` response message to the Requester, and `SigLen` is the size of that asymmetric signing algorithm output. This field is conditional and is only present in the `MEASUREMENTS` response corresponding to a `GET_MEASUREMENTS` request with `Param1[0]` set to `1`.

504 **10.12.1 Measurement block**

505 Each measurement block that the `MEASUREMENTS` response message defines shall contain a four-byte descriptor, offsets 0 through 3, followed by the measurement data that corresponds to a particular measurement index and measurement type.

506 Table 57 — Measurement block format shows the format for a measurement block:

507 **Table 57 — Measurement block format**

Byte offset	Field	Size (bytes)	Description
0	Index	1	Shall be the index. When `Param2` of the `GET_MEASUREMENTS` request is between `0x1` and `0xFE` , inclusive, this field shall match the request. Otherwise, this field shall represent the index of the measurement block, where the index starts at 1 and ends at the index of the last measurement block.
1	MeasurementSpecification	1	Bit mask. The value shall indicate the measurement specification that the requested `Measurement` follows and shall match the selected measurement specification (`MeasurementSpecificationSel`) in the `ALGORITHMS` message. See Table 23 — Successful ALGORITHMS response message format. Only one bit shall be set. The Measurement specification field format table defines the format for this field.
2	MeasurementSize	2	Shall be the size of `Measurement` , in bytes.
4	Measurement	`MeasurementSize`	Shall be the measurement value whose format the selected measurement specification (`MeasurementSpecificationSel`) defines. If `DMTFmeasSpec` is selected, the format of this field shall be as Table 58 — DMTF measurement specification format defines.

508 **10.12.1.1 DMTF specification for the Measurement field of a measurement block**

509 The present clause is the specification for the format of the `Measurement` field in a measurement block when the `MeasurementSpecification` field's Bit 0 (DMTF) is set. Table 58 — DMTF measurement specification format specifies this format.

510 **10.12.1.1.1 Measurement manifest**

511 A measurement manifest refers to a data structure that describes the contents of other indices or itself contains measurements. For instance, a manifest may describe which indices describe the different firmware modules' measurements. When the Table 58 — DMTF measurement specification format is in use, this specification defines multiple overarching manifest formats, as described in the DMTFSpecMeasurementValueType values table.

512 When `DMTFSpecMeasurementValueType[6:0]=0x04` , the measurement manifest type is a freeform manifest. When read, the manifest data is placed in the `Measurement` field of the Table 57 — Measurement block format. The format of a freeform manifest is implementation specific and outside the scope of this specification.

513 When `DMTFSpecMeasurementValueType[6:0]=0x0A` , the measurement manifest type is a structured measurement manifest. The structured manifest starts with an `SVH` header as Table 61 — Manifest measurement block format

describes. The `SVH` header is used to indicate the standards body or vendor that defines the manifest format. The format of the `Manifest` data in a structured measurement manifest is outside the scope of this specification.

514 **10.12.1.1.2 Hash-extend measurements**

515 A device may support reporting of measurements through an "extend" scheme, which works as follows:

```
initialize HEM = MH bytes of 0s
for each extend operation, perform HEM = hash(Concatenate(HEM, DataToExtend)) for all data elements to
extend
```

516 An example of such a scheme is the Platform Configuration Register "extend" function in Trusted Platform Modules. The `hash()` function is the measurement hash algorithm specified by the most recent `ALGORITHMS` response message. The initial value of a hash-extend measurement (HEM) shall be `MH` bytes whose bits are all set to `0`, where `MH` is the size of `MeasurementHashAlgo` in the most recent `ALGORITHMS` response message. The hash-extend measurement is updated by "extending" the current value to include the next data to extend (`DataToExtend`). The extend operation is calculating the digest of the current value concatenated with the data to extend. Then repeat the extend operation for additional data to extend.

517 Hash-extend measurements are reported in a measurement block. A Responder that reports hash-extend measurements shall set `DMTFSpecMeasurementValueType[6:0]` to `0x8` for the corresponding measurement index.

518 **Table 58 — DMTF measurement specification format**

Byte offset	Field	Size (bytes)	Description
0	DMTFSpecMeasurementValueType	1	Composed of: • Bit [7]. Shall indicate the representation in `DMTFSpecMeasurementValue`. • Bit [6:0]. Indicates what is being measured by `DMTFSpecMeasurementValue`. These values are set independently and are interpreted as follows: • `[7]=0b`. Digest. • `[7]=1b`. Raw bit stream. The Responder should ensure the raw bit stream does not contain secrets. • See DMTFSpecMeasurementValueType values for defined values for `DMTFSpecMeasurementValueType[6:0].`
1	DMTFSpecMeasurementValueSize	2	Shall be the size of `DMTFSpecMeasurementValue`, in bytes. When `DMTFSpecMeasurementValueType[7]=0b`, the `DMTFSpecMeasurementValueSize` shall be derived from the measurement hash algorithm that the `ALGORITHMS` response message returns.

Byte offset	Field	Size (bytes)	Description
3	DMTFSpecMeasurementValue	`MS`	Shall be the cryptographic hash or raw bit stream, as indicated in `DMTFSpecMeasurementValueType[7]` . For cryptographic hashes or digests, this field shall be in hash byte order. The vendor defines the byte order for raw bit streams.

519 **Table 59 — DMTFSpecMeasurementValueType values**

DMTFSpecMeasurementValueType[6:0]	Description
0x0	Immutable ROM.
0x1	Mutable firmware.
0x2	Hardware configuration, such as straps.
0x3	Firmware configuration, such as configurable firmware policy.
0x4	Freeform measurement manifest. When `DMTFSpecMeasurementValueType[6:0]=0x4` , the Responder should support setting `DMTFSpecMeasurementValueType[7]` to either `0b` or `1b` . The format of this manifest is device specific.
0x5	Structured representation of debug and device mode. See Device mode field of a measurement block. When `DMTFSpecMeasurementValueType[6:0]=0x5` , `DMTFSpecMeasurementValueType[7]` shall be set to `1b` .
0x6	Mutable firmware's version number. This specification does not mandate a format for firmware version number. When `DMTFSpecMeasurementValueType[6:0]=0x6` , `DMTFSpecMeasurementValueType[7]` should be set to `1b` .
0x7	Mutable firmware's security version number, which should be formatted as an 8-byte unsigned integer. When `DMTFSpecMeasurementValueType[6:0]=0x7` , `DMTFSpecMeasurementValueType[7]` should be set to `1b` .
0x8	Hash-extend measurement. The measurement reported is an `HEM` value as defined in Hash-extend measurements. When `DMTFSpecMeasurementValueType[6:0]=0x8` , `DMTFSpecMeasurementValueType[7]` shall be set to `0b` .
0x9	Informational. The measurement is for the Requester's information only and does not carry sensitive security attributes. For example, human-readable boot progress information. When `DMTFSpecMeasurementValueType[6:0]=0x9` , `DMTFSpecMeasurementValueType[7]` shall be set to `1b` .

DMTFSpecMeasurementValueType[6:0]	Description
0xA	Structured measurement manifest. When `DMTFSpecMeasurementValueType[6:0]=0xA` , the Responder shall support setting `DMTFSpecMeasurementValueType[7]` to `1b` , and should support setting `DMTFSpecMeasurementValueType[7]` to `0b` . The manifest shall follow the format described in Manifest format for a measurement block.
All other values	Reserved.

520 **Table 60 — Device mode field of a measurement block**

521 10.12.1.2 Device mode field of a measurement block

Byte offset	Field	Size (bytes)	Description
0	OperationalModeCapabilities	4	Fields with bits set to 1 indicate support for reporting the associated state in `OperationalModeState` . • Bit [0]. Shall indicate support for reporting device in manufacturing mode. • Bit [1]. Shall indicate support for reporting device in validation mode. • Bit [2]. Shall indicate support for reporting device in normal operational mode. • Bit [3]. Shall indicate support for reporting device in recovery mode. • Bit [4]. Shall indicate support for reporting device in Return Merchandise Authorization (RMA) mode. • Bit [5]. Shall indicate support for reporting device in decommissioned mode. • All other values reserved.
4	OperationalModeState	4	Fields with bits set to 1 indicate true for the reported state. • Bit [0]. Shall indicate the device is in manufacturing mode. • Bit [1]. Indicates the device is in validation mode. • Bit [2]. Shall indicate the device is in normal operational mode. • Bit [3]. Shall indicate the device is in recovery mode. • Bit [4]. Shall indicate the device is in RMA mode. • Bit [5]. Shall indicate the device is in decommissioned mode. • All other values reserved.

Byte offset	Field	Size (bytes)	Description
8	DeviceModeCapabilities	4	Fields with bits set to 1 indicate support for reporting the associated state in `DeviceModeState`. • Bit [0]. Shall indicate support for reporting non-invasive debug mode is active. • Bit [1]. Shall indicate support for reporting invasive debug mode is active. • Bit [2]. Shall indicate support for reporting non-invasive debug mode has been active this Reset cycle. • Bit [3]. Shall indicate support for reporting invasive debug mode has been active this Reset cycle. • Bit [4]. Shall indicate support for reporting invasive debug mode has been active on this device at least once since exiting manufacturing mode. • All other values reserved.
12	DeviceModeState	4	Fields with bits set to 1 indicate true for the reported state. • Bit [0]. Shall indicate non-invasive debug mode is active. • Bit [1]. Shall indicate invasive debug mode is active. • Bit [2]. Shall indicate non-invasive debug mode has been active this Reset cycle. • Bit [3]. Shall indicate invasive debug mode has been active this Reset cycle. • Bit [4]. Shall indicate invasive debug mode has been active on this device at least once since exiting manufacturing mode. • All other values reserved.

522 **10.12.1.3 Manifest format for a measurement block**

523 When `DMTFSpecMeasurementValueType[6:0]=0xA`, the response shall be either a manifest or the digest of a manifest. If `DMTFSpecMeasurementValueType[7]=0b`, then the `Measurement` field of the Measurement block shall contain a digest of the structure described in Table 61 — Manifest measurement block format. If `DMTFSpecMeasurementValueType[7]=1b`, then the `Measurement` field of the Measurement block shall contain a manifest in the format described in Table 61 — Manifest measurement block format.

524 **Table 61 — Manifest measurement block format**

Byte offset	Field	Size (bytes)	Description
0	SVH	2 + `VendorIDLen`	Shall be a standards body or vendor-defined header, as described in Table 69 — Standards body or vendor-defined header (SVH).
2 + `VendorIDLen`	Manifest	Variable	Shall contain the manifest data, as defined by the registry, standards body, or vendor specified in the `ID` and `VendorID` fields.

525 **10.12.2 MEASUREMENTS signature generation**

526 While a Requester can opt to require a signature in each of the request-response messages, it is advisable that the cost of the signature generation process is minimized by amortizing it over multiple request-response messages where applicable. In this scheme, the Requester issues a number of requests without requiring signatures followed by a final request requiring a signature over the entire set of request-response messages exchanged. The steps to complete this scheme are as follows:

527 1. The Responder shall construct measurement log L1 and the Requester shall construct measurement log L2 over their observed messages:

```
L1/L2 = Concatenate(VCA, GET_MEASUREMENTS_REQUEST1, MEASUREMENTS_RESPONSE1, ...,
                    GET_MEASUREMENTS_REQUESTn-1, MEASUREMENTS_RESPONSEn-1,
                    GET_MEASUREMENTS_REQUESTn, MEASUREMENTS_RESPONSEn)
```

528 where:

- `Concatenate` is the standard concatenation function.
- `GET_MEASUREMENTS_REQUEST1` is the entire first `GET_MEASUREMENTS` request message under consideration, where the Requester has not requested a signature on that specific `GET_MEASUREMENTS` request.
- `MEASUREMENTS_RESPONSE1` is the entire `MEASUREMENTS` response message without the signature bytes that the Responder sent in response to `GET_MEASUREMENTS_REQUEST1`.
- `GET_MEASUREMENTS_REQUESTn-1` is the entire last consecutive `GET_MEASUREMENTS` request message under consideration, where the Requester has not requested a signature on that specific `GET_MEASUREMENTS` request.
- `MEASUREMENTS_RESPONSEn-1` is the entire `MEASUREMENTS` response message without the signature bytes that the Responder sent in response to `GET_MEASUREMENTS_REQUESTn-1`.
- `GET_MEASUREMENTS_REQUESTn` is the entire first `GET_MEASUREMENTS` request message under consideration, where the Requester has requested a signature on that specific `GET_MEASUREMENTS` request. *n* is a number greater than or equal to `1`. When *n* equals `1`, the Requester has not made any `GET_MEASUREMENTS` requests without signature prior to issuing a `GET_MEASUREMENTS` request with signature.
- `MEASUREMENTS_RESPONSEn` is the entire `MEASUREMENTS` response message without the signature

bytes that the Responder sent in response to `GET_MEASUREMENTS_REQUESTn`.

529 Completion of `MEASUREMENTS` with signature shall re-initialize L1/L2 to `null`. Any communication between Requester and Responder other than a `GET_MEASUREMENTS` request or response re-initializes L1/L2 computation to `null`. The `GET_MEASUREMENTS` requests and `MEASUREMENTS` responses before the L1/L2 re-initialization will not be covered by the signature in the final `MEASUREMENTS` response. Consequently, it is recommended that the Requester not use the measurements before verifying the signature.

530 An `ERROR` message of `ErrorCode=ResponseNotReady` or `ErrorCode=LargeResponse` shall not re-initialize L1/L2 and the Requester and Responder shall continue to construct L1/L2 with `GET_MEASUREMENTS` and `MEASUREMENTS`. An error response with any error code other than `ResponseNotReady` or `LargeResponse` shall re-initialize L1/L2 to `null`.

531 2. The Responder shall generate:

```
Signature = SPDMsign(PrivKey, L1, "measurements signing");
```

532 where:

- `SPDMsign` is described in Signature generation.
- `PrivKey` shall be the private key of the Responder associated with the leaf certificate stored in `SlotID` of `SlotIDParam` in `GET_MEASUREMENTS`. If the public key of the Responder was provisioned to the Requester, then `PrivKey` shall be the associated private key.

533 **10.12.3 MEASUREMENTS signature verification**

534 To complete the `MEASUREMENTS` signature verification process, the Requester shall complete this step:

535 1. The Requester shall perform:

```
result = SPDMsignatureVerify(PubKey, Signature, L2, "measurements signing")
```

536 where:

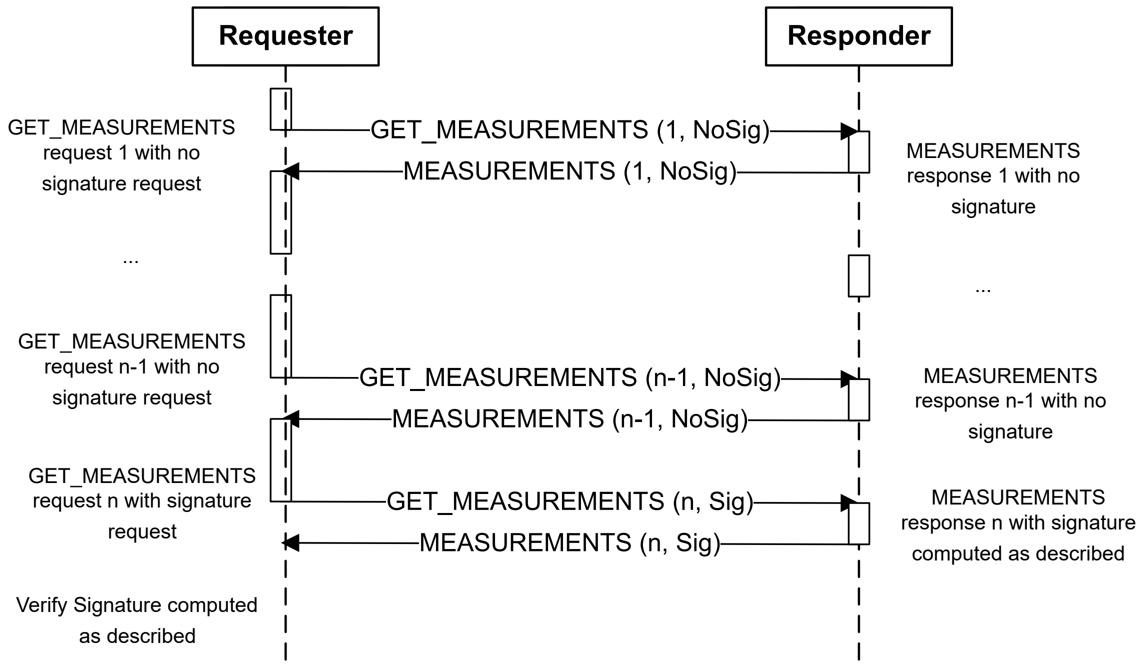
- `SPDMsignatureVerify` is described in Signature verification. A successful verification is when `result` is `success`.
- `PubKey` shall be the public key associated with the leaf certificate stored in `SlotID` of `SlotIDParam` in `GET_MEASUREMENTS`. `PubKey` is extracted from the `CERTIFICATE` response. If the public key of the Responder was provisioned to the Requester, then `PubKey` shall be the provisioned public key.

537 Figure 13 — Measurement signature computation example shows an example of a typical Requester-Responder

protocol where the Requester issues 1 to *n*-1 `GET_MEASUREMENTS` requests without a signature, which is followed by a single `GET_MEASUREMENTS` request *n* with a signature.

538 **Figure 13 — Measurement signature computation example**

539



540 10.13 ERROR response message

541 For an SPDM operation that results in an error, the Responder should send an `ERROR` message to the Requester.

542 Table 62 — ERROR response message format shows the `ERROR` response format.

543 Table 63 — Error code and error data shows the detailed error code, error data, and extended error data.

544 Table 64 — ResponseNotReady extended error data shows the `ResponseNotReady` extended error data.

545 Table 65 — Registry or standards body ID shows the registry or standards body ID.

546 Table 66 — ExtendedErrorData format for vendor or other standards-defined ERROR response message shows the `ExtendedErrorData` format definition for vendor or other standards-defined `ERROR` response messages.

547 **Table 62 — ERROR response message format**

Byte offset	Field	Size (bytes)	Description
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	RequestResponseCode	1	Shall be `0x7F` = `ERROR`. See Table 5 — SPDM response codes.
2	Param1	1	Shall be the ErrorCode. See Table 63 — Error code and error data.
3	Param2	1	Shall be the Error data. See Table 63 — Error code and error data.
4	ExtendedErrorData	0-32	Shall be Optional extended data. See Table 63 — Error code and error data.

548 **Table 63 — Error code and error data**

`ErrorCode`	Value	Description	Error data	`ExtendedErrorData`
Reserved	0x00	Reserved.	Reserved	Reserved
InvalidRequest	0x01	One or more request fields are invalid	`0x00`	No extended error data is provided.
Reserved	0x02	Reserved.	Reserved	No extended error data is provided.
Busy	0x03	The Responder received the request message and the Responder decided to ignore the request message, but the Responder might be able to process the request message if the request message is sent again in the future.	`0x00`	No extended error data is provided.
UnexpectedRequest	0x04	The Responder received an unexpected request message. For example, `CHALLENGE` before `NEGOTIATE_ALGORITHMS`.	`0x00`	No extended error data is provided.
Unspecified	0x05	Unspecified error occurred.	`0x00`	No extended error data is provided.
DecryptError	0x06	The receiver cannot decrypt or verify data during the session.	Reserved	No extended error data is provided.

`ErrorCode`	Value	Description	Error data	`ExtendedErrorData`
UnsupportedRequest	0x07	The `RequestResponseCode` or the `SubCode` (if applicable) in the request message is unsupported.	`RequestResponseCode` or the `SubCode` in the request message.	No extended error data is provided
RequestInFlight	0x08	The Responder has delivered an encapsulated request to which it is still waiting for the response.	Reserved	No extended error data is provided.
InvalidResponseCode	0x09	The Requester delivered an invalid response for an encapsulated response.	Reserved	No extended error data is provided.
SessionLimitExceeded	0x0A	Maximum number of concurrent sessions reached.	Reserved	No extended error data is provided.
SessionRequired	0x0B	The Request message received by the Responder is only allowed within a session.	Reserved	No extended error data is provided.
ResetRequired	0x0C	The device requires a reset to complete the requested operation. This `ErrorCode` can be sent in response to the `GET_DIGESTS` , `GET_CERTIFICATE` , `GET_CSR` , `SET_CERTIFICATE` , `GET_KEY_PAIR_INFO` or `SET_KEY_PAIR_INFO` message.	Bit[7:3]. Reserved. Bit[2:0]. If sent in response to `GET_CSR` , the Responder-assigned `CSRTrackingTag` . Otherwise, shall be `0` .	No extended error data is provided.

`ErrorCode`	Value	Description	Error data	`ExtendedErrorData`
ResponseTooLarge	0x0D	Used in the following scenarios. • The response is greater than the `MaxSPDMmsgSize` of the requesting SPDM endpoint. • The `CHUNK_CAP` of the requesting endpoint is `0` and the response is larger than the size of the transmit buffer of the responding SPDM endpoint • The `CHUNK_CAP` of the requesting endpoint is `1`, the `CHUNK_CAP` of the responding endpoint is `0`, and the response is larger than the `DataTransferSize` of the requesting endpoint.	Reserved	See Table 67 — ExtendedErrorData format for ResponseTooLarge.
RequestTooLarge	0x0E	The request is greater than the `MaxSPDMmsgSize` of the receiving SPDM endpoint.	Reserved	Reserved
LargeResponse	0x0F	The response is greater than `DataTransferSize` and less than or equal to `MaxSPDMmsgSize` of the requesting SPDM endpoint, or greater than the transmit buffer size of the responding SPDM endpoint.	Reserved	See Table 68 — ExtendedErrorData format for LargeResponse.
MessageLost	0x10	The SPDM message is lost. For example, this error code can be used to indicate the loss of a Large Request, Large Response, or the request in a `ResponseNotReady`.	Reserved	Reserved

`ErrorCode`	Value	Description	Error data	`ExtendedErrorData`
InvalidPolicy	0x11	The Responder received one or more messages that violated its security policy. For example, if a Responder requires both encryption and MAC capabilities in a secure session, and the Requester only supports encryption, then the Responder would return this error code if the Requester sends `KEY_EXCHANGE` .	Reserved	Reserved
CertChainTooLarge	0x12	• The `GET_CERTIFICATE` message is requesting a certificate chain with a size that cannot be represented by the `Offset` and `Length` fields. The Requester should use `LargeOffset` and `LargeLength` fields for subsequent `GET_CERTIFICATE` messages. • The `SET_CERTIFICATE` message contains a certificate chain with a size that requires large fields but `LARGE_CERT_CAP=0b` .	Reserved	See Table 68.1 — ExtendedErrorData format for CertChainTooLarge.
Reserved	0x13–0x40	Reserved	Reserved	Reserved
VersionMismatch	0x41	Requested SPDM version is not supported or is a different version from the selected version.	`0x00`	No extended error data is provided.
ResponseNotReady	0x42	See the RESPOND_IF_READY request message format.	`0x00`	See Table 64 — ResponseNotReady extended error data.

`ErrorCode`	Value	Description	Error data	`ExtendedErrorData`
RequestResynch	0x43	Responder is requesting Requester to reissue `GET_VERSION` to re-synchronize. An example is following a firmware update.	`0x00`	No extended error data is provided.
OperationFailed	0x44	An internal error occurred upon servicing the request issued by the Requester.	`0x00`	No extended error data is provided.
NoPendingRequests	0x45	The Responder does not have any pending request for a `GET_ENCAPSULATED_REQUEST` message.	Reserved	Reserved
Reserved	0x46–0xFE	Reserved.	Reserved	Reserved
Vendor or Standards-Defined	0xFF	Vendor or standards-defined	Shall indicate the registry or standards body using one of the values in the **ID** column of Table 59 — Registry or standards body ID.	See Table 66 — ExtendedErrorData format for vendor or other standards-defined ERROR response message for format definition.

549 **Table 64 — ResponseNotReady extended error data**

Byte offset	Field	Size (bytes)	Description
0	RDTExponent	1	Shall be the exponent expressed in logarithmic (base-2 scale) to calculate `RDT` time in µs after which the Responder can provide successful completion response. For example, the raw value 8 indicates that the Responder will be ready in 2^8 = 256 µs. Requester should use `RDT` to avoid continuous pinging and issue the `RESPOND_IF_READY` request message, as Table 70 — RESPOND_IF_READY request message format shows, after `RDT` time. For timing requirement details, see Table 7 — Timing specification for SPDM messages.
1	RequestCode	1	Shall be the request code that triggered this response.

Byte offset	Field	Size (bytes)	Description
2	Token	1	Shall be the opaque handle that the Requester shall pass in with the `RESPOND_IF_READY` request message, as Table 70 — RESPOND_IF_READY request message format shows. The Responder can use the value in this field to provide the correct response when the Requester issues a `RESPOND_IF_READY` request.
3	RDTM	1	Shall be the multiplier used to compute WT_{Max} in µs to indicate that the response might be dropped after this delay. The multiplier shall always be greater than 1. The Responder might also stop processing the initial request if the same Requester issues a different request. For timing requirement details, see Table 7 — Timing specification for SPDM messages.

550 **Table 65 — Registry or standards body ID**

551 For algorithm encoding in extended algorithm fields, consult the respective registry or standards body unless otherwise specified.

ID	Vendor ID length (bytes)	Registry or standards body name	Description
0x0	0	DMTF	DMTF does not have a Vendor ID registry.
0x1	2	TCG	`VendorID` is identified by using TCG Vendor ID Registry. For extended algorithms, see TCG Algorithm Registry.
0x2	2	USB	`VendorID` is identified by using the vendor ID assigned by USB.
0x3	2	PCI-SIG	`VendorID` is identified using PCI-SIG Vendor ID.
0x4	4	IANA	The Private Enterprise Number (PEN) assigned by the Internet Assigned Numbers Authority (IANA) identifies the vendor.

ID	Vendor ID length (bytes)	Registry or standards body name	Description
0x5	4	HDBaseT	`VendorID` is identified by using HDBaseT HDCD entity.
0x6	2	MIPI	The Manufacturer ID assigned by MIPI identifies the vendor.
0x7	2	CXL	`VendorID` is identified by using CXL vendor ID.
0x8	2	JEDEC	`VendorID` is identified by using JEDEC vendor ID.
0x9	0	VESA	For fields and formats defined by the VESA standards body, there is no Vendor ID registry.
0xA	Variable	IANA CBOR	The CBOR Tag Registry that identifies the format of the element, as assigned by the Internet Assigned Numbers Authority (IANA). The encoding of the CBOR tag indicates the length of the tag. When a CBOR Tag is used with a standards body or vendor-defined header, the `VendorIDLen` field shall be set to the length of the encoded CBOR tag, followed by the data payload, which starts with an encoded CBOR tag.

ID	Vendor ID length (bytes)	Registry or standards body name	Description
0xB	2	DMTF-DSP	DMTF does not have a Vendor ID registry that identifies a vendor. However, this ID provides all DMTF specifications (DSP) the ability to carve out a namespace directly under its control. Thus, the `VendorID` field shall be the DSP number. For example, this specification, DSP 274, will have a value of 274 (0x112) populated in that field. When using this `ID` , the `VendorID` field shall always be present with a valid DSP value and the `VendorIDLen` shall always be 2.

552 **Table 66 — ExtendedErrorData format for vendor or other standards-defined ERROR response message**

Byte offset	Field	Size (bytes)	Description
0	Len	1	Shall be the length of the `VendorID` field. If the vendor defines the error, the value of this field shall equal the "Vendor ID length", as Table 65 — Registry or standards body ID describes, of the corresponding registry or standards body name. If a registry or standards body defines the error, this field shall be zero (`0`), which also indicates that the `VendorID` field is not present. The `Error Data` field in the `ERROR` message indicates the registry or standards body name (that is, `Param2`) and is one of the values in the **ID** column of Table 65 — Registry or standards body ID.

Byte offset	Field	Size (bytes)	Description
1	VendorID	`Len`	The value of this field shall indicate the Vendor ID as assigned by the registry or standards body. Table 65 — Registry or standards body ID describes the length of this field. Shall be in little-endian format. The name of the registry or standards body in the `ERROR` is indicated in the `Error Data` field (that is, `Param2`) and is one of the values in the **ID** column of Table 65 — Registry or standards body ID.
1 + `Len`	OpaqueErrorData	Variable	The vendor or standards body defines this value.

553 **Table 67 — ExtendedErrorData format for ResponseTooLarge**

Byte offset	Field	Size (bytes)	Description
0	ActualSize	4	Shall be the size of the actual response.

554 **Table 68 — ExtendedErrorData format for LargeResponse**

Byte offset	Field	Size (bytes)	Description
0	Handle	1	Shall be a unique value that identifies the Large SPDM Response and shall be the same value for all chunks of the same large SPDM message. The value of this field should either sequentially increase or sequentially decrease with each large SPDM message with the expectation that it will wrap around after reaching the maximum or minimum value, respectively, of this field. See CHUNK_GET request and CHUNK_RESPONSE response message.

555 **Table 68.1 — ExtendedErrorData format for CertChainTooLarge**

Byte offset	Field	Size (bytes)	Description
0	CertChainLength	4	Shall be the size of the certificate chain.

556 10.13.1 Standards body or vendor-defined header

557 This specification uses the format that Table 69 — Standards body or vendor-defined header (SVH) describes to help identify the entity that defines the format for a given payload. The clauses in the other parts of this specification indicate to which payload this header applies. Note, if the payload format in question is defined by a standards body, the SVH header does not require the use of the `VendorID` field. Instead, the `ID` field would be set to the ID of the standards body, `VendorIDLen` would be set to `0` , and `VendorID` would be absent. A standards body, registry, or vendor that defines a payload format should also define the values to use in the SVH header.

558 **Table 69 — Standards body or vendor-defined header (SVH)**

Byte offset	Field	Size (bytes)	Description
0	ID	1	Shall be one of the values in the `ID` column of Table 65 — Registry or standards body ID.
1	VendorIDLen	1	Shall be the `Length` in bytes of the `VendorID` field. If the format of the given payload is specified by a standards body or registry itself, this field shall be 0. Otherwise, if the format of the given payload is specified by an organization that is identified on the vendor ID list indicated in the `ID` field, this field shall be the length indicated in the "Vendor ID length" column of Table 65 — Registry or standards body ID for the respective `ID` .
2	VendorID	`VendorIDLen`	If `VendorIDLen` is greater than zero, this field shall be the ID of the vendor corresponding to the `ID` field. Otherwise, this field shall be absent.

559

10.14 RESPOND_IF_READY request message format

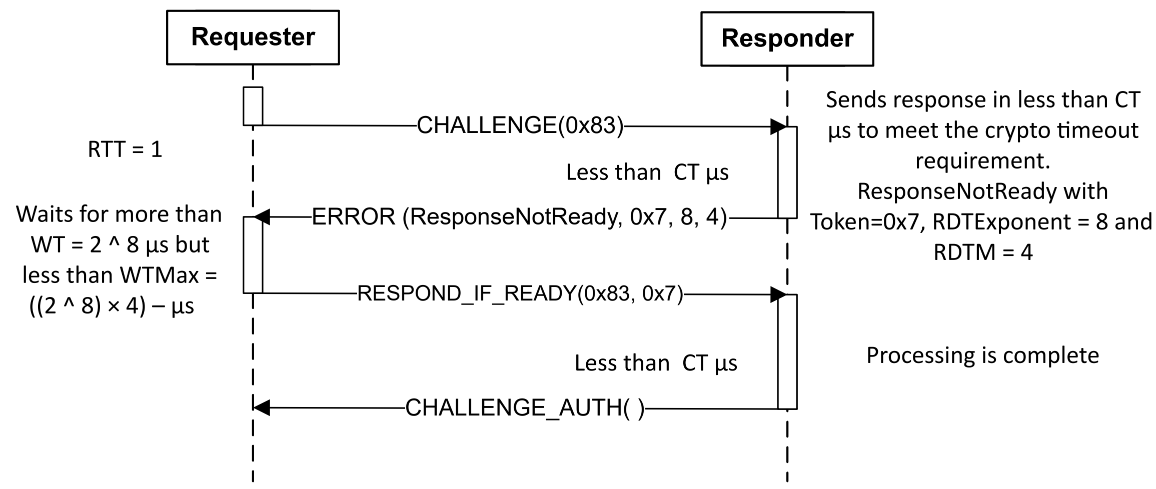
560 This request message shall ask for the response to the original request upon receipt of the `ResponseNotReady` error code. If the response to the original request is ready, the Responder shall return that response message. If the response to the original request is not ready, the Responder shall return an `ERROR` message of `ErrorCode` = `ResponseNotReady` and return the same token as the previous `ResponseNotReady` response message.

561 The validity of the `RESPOND_IF_READY` request (see the SPDM Request and Response messages validity table) is defined by the original request that caused the `RESPOND_IF_READY` flow. This means the last request to which the Responder sent an `ERROR` message of `ErrorCode=ResponseNotReady` .

562 Figure 14 — `RESPOND_IF_READY` flow leading to completion shows the `RESPOND_IF_READY` flow:

563 **Figure 14 — `RESPOND_IF_READY` flow leading to completion**

564



565 Table 70 — `RESPOND_IF_READY` request message format shows the `RESPOND_IF_READY` request message format.

566 **Table 70 — `RESPOND_IF_READY` request message format**

Byte offset	Field	Size (bytes)	Description
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	RequestResponseCode	1	Shall be `0xFF` = `RESPOND_IF_READY` . See Table 4 — SPDM request codes.
2	Param1	1	Shall be the original request code that triggered the `ResponseNotReady` error code response. Shall match the request code returned as part of the `ResponseNotReady` extended error data.
3	Param2	1	Shall be the token that was returned as part of the `ResponseNotReady` extended error data.

567 10.15 VENDOR_DEFINED_REQUEST request message

568 A Requester intending to define a unique request to meet its needs can use this request message. Table 71 — VENDOR_DEFINED_REQUEST request message format defines the format.

569 The Requester should send this request message only after sending the `GET_VERSION` , `GET_CAPABILITIES` , and `NEGOTIATE_ALGORITHMS` request sequence.

570 If the vendor wishes to have the requests authenticated, then the vendor shall indicate how the transcript and/or message transcript are changed to add the vendor-defined commands.

571 Table 71 — VENDOR_DEFINED_REQUEST request message format shows the `VENDOR_DEFINED_REQUEST` request message format.

572 **Table 71 — VENDOR_DEFINED_REQUEST request message format**

Byte offset	Field	Size (bytes)	Description
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	RequestResponseCode	1	Shall be `0xFE` = `VENDOR_DEFINED_REQUEST`. See Table 4 — SPDM request codes.
2	Param1	1	Reserved.
3	Param2	1	Reserved.
4	StandardID	2	Shall indicate the registry or standards body by using one of the values in the **ID** column of Table 65 — Registry or standards body ID.
6	Len	1	Shall be the length of the `Vendor ID` field. If the `VendorDefinedReqPayload` is standards-defined, `Len` shall be `0`. If the `VendorDefinedReqPayload` is vendor-defined, `Len` shall equal "Vendor ID length", as Table 65 — Registry or standards body ID describes.
7	VendorID	Len	Shall be the Vendor ID as assigned by the registry or standards body. Shall be in little-endian format.
7 + `Len`	ReqLength	2	Shall be the length of the `VendorDefinedReqPayload`.
7 + `Len` + 2	VendorDefinedReqPayload	`ReqLength`	This field shall be used to send the request payload.

573 Other DMTF specifications may define `VENDOR_DEFINED_REQUEST` with `StandardID` set to 0. See VendorDefinedReqPayload and VendorDefinedRespPayload defined by DMTF specifications for more information.

574

10.16 VENDOR_DEFINED_RESPONSE response message

575 A Responder can use this response message in response to `VENDOR_DEFINED_REQUEST`. Table 72 — VENDOR_DEFINED_RESPONSE response message format defines the format.

576 **Table 72 — VENDOR_DEFINED_RESPONSE response message format**

Byte offset	Field	Size (bytes)	Description
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	RequestResponseCode	1	Shall be `0x7E` = `VENDOR_DEFINED_RESPONSE`. See Table 5 — SPDM response codes.

Byte offset	Field	Size (bytes)	Description
2	Param1	1	Reserved.
3	Param2	1	Reserved.
4	StandardID	2	Shall indicate the registry or standards body using one of the values in the **ID** column of Table 65 — Registry or standards body ID.
6	Len	1	Shall be the length of the `Vendor ID` field. If the `VendorDefinedRespPayload` is standards-defined, length shall be `0`. If the `VendorDefinedRespPayload` is vendor-defined, length shall equal "Vendor ID length" as Table 65 — Registry or standards body ID describes.
7	VendorID	`Len`	Shall indicate the Vendor ID as assigned by the registry or standards body. Shall be in little-endian format.
7 + `Len`	RespLength	2	Shall be the length of the `VendorDefinedRespPayload`
7 + `Len` + 2	VendorDefinedRespPayload	`RespLength`	This value shall be used to send the response payload.

577 ### 10.16.1 VendorDefinedReqPayload and VendorDefinedRespPayload defined by DMTF specifications

578 Other DMTF specifications may define `VENDOR_DEFINED_REQUEST` and `VENDOR_DEFINED_RESPONSE` messages with `StandardID` set to 0 ("DMTF", as defined in Table 65 — Registry or standards body ID) and `Len` set to 0. In this case, `VENDOR_DEFINED_REQUEST` and `VENDOR_DEFINED_RESPONSE` messages shall specify the underlying DMTF specification that defines them. A DMTF specification which defines the data model of VendorDefinedReqPayload for `VENDOR_DEFINED_REQUEST` and the data model of VendorDefinedRespPayload for `VENDOR_DEFINED_RESPONSE` shall follow Table 73 — Format of VendorDefinedReqPayload and VendorDefinedRespPayload when `StandardID` is DMTF.

579 **Table 73 — Format of VendorDefinedReqPayload and VendorDefinedRespPayload when `StandardID` is DMTF**

Byte offset	Field	Size (bytes)	Description
0	DSPNumber	2	Shall be the DMTF specification's DSP number as a 16-bit integer. For example, DSP0287 would use `0x011F`.
2	DSPVersion	2	Shall be the version number of the DMTF specification whose DSP number is populated in the `DSPNumber` field. The format of the version number shall follow Table 10 — VersionNumberEntry definition.

Byte offset	Field	Size (bytes)	Description
4	VendorPayload	Variable	Shall be the actual payload data defined by the DMTF specification whose DSP number is populated in the `DSPNumber` field.

580 DMTF DSP can also use `StandardID` set to 0xB (DMTF-DSP). If the DMTF DSP uses a `StandardID` of 0xB, then that DMTF DSP defines the format for both `VendorDefinedReqPayload` and `VendorDefinedRespPayload`.

581 10.17 KEY_EXCHANGE request and KEY_EXCHANGE_RSP response messages

582 This request message shall initiate a handshake between Requester and Responder intended to authenticate the Responder (or, optionally, both parties), negotiate cryptographic parameters (in addition to those negotiated in the last `NEGOTIATE_ALGORITHMS` / `ALGORITHMS` exchange), and establish shared keying material.

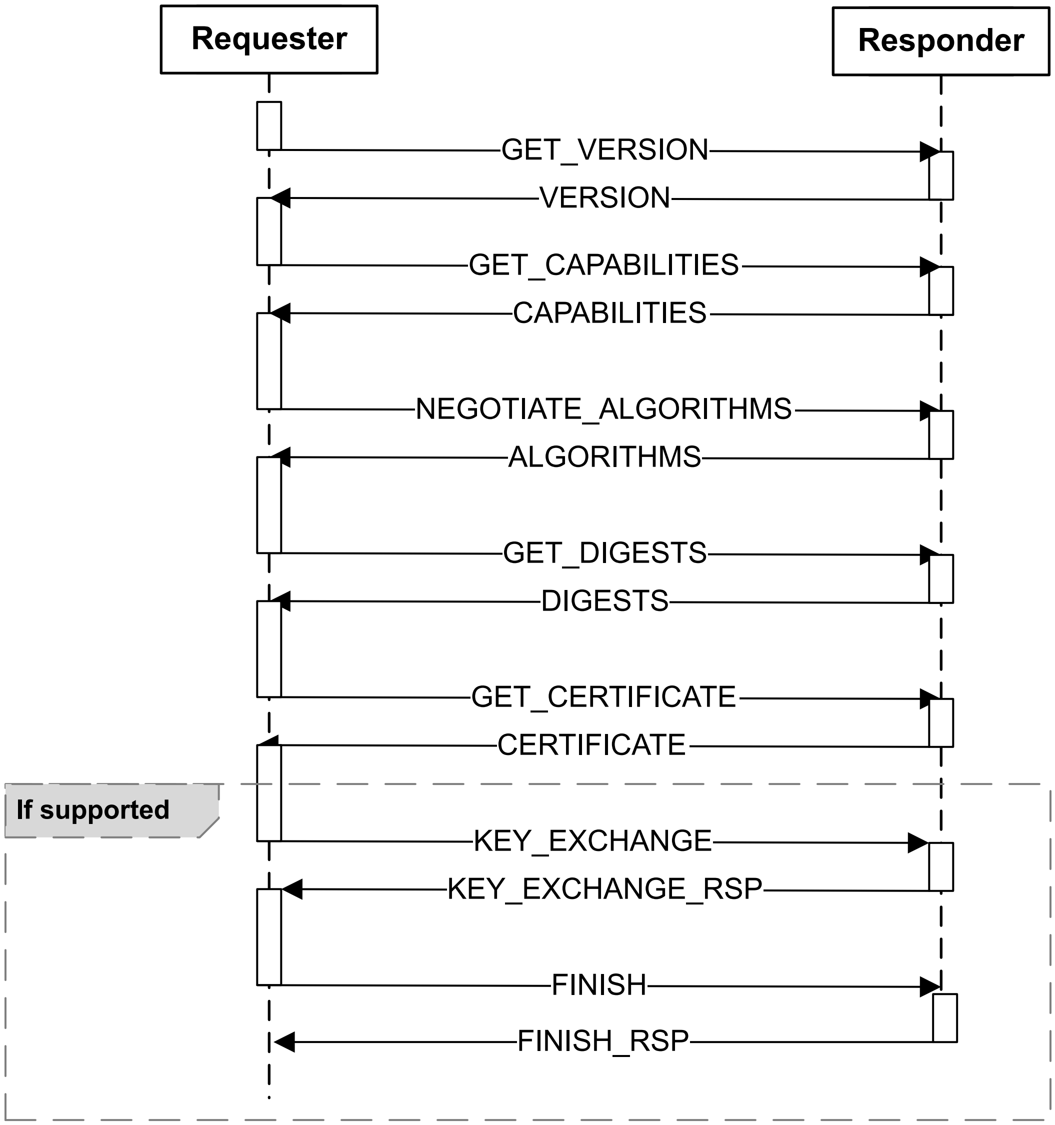
Table 74 — KEY_EXCHANGE request message format shows the `KEY_EXCHANGE` request message format, and Table 76 — Successful KEY_EXCHANGE_RSP response message format shows the `KEY_EXCHANGE_RSP` response message format. The handshake is completed by the successful exchange of the `FINISH` request and `FINISH_RSP` response messages presented in the next clause. The handshake depends on the tight coupling between these two request/response message pairs.

583 The Requester-Responder pair can support two modes of handshakes. If `HANDSHAKE_IN_THE_CLEAR_CAP` is set in both the Requester and the Responder, all SPDM messages exchanged during the Session Handshake Phase are sent in the clear (outside of a secure session). Otherwise both the Requester and the Responder use encryption and/or message authentication during the Session Handshake Phase using the Handshake secret derived at the completion of the `KEY_EXCHANGE_RSP` message for subsequent message communication until the completion of the `FINISH_RSP` message.

584 Figure 15 — Responder authentication key exchange example shows an example of a Responder authentication key exchange:

585 **Figure 15 — Responder authentication key exchange example**

586

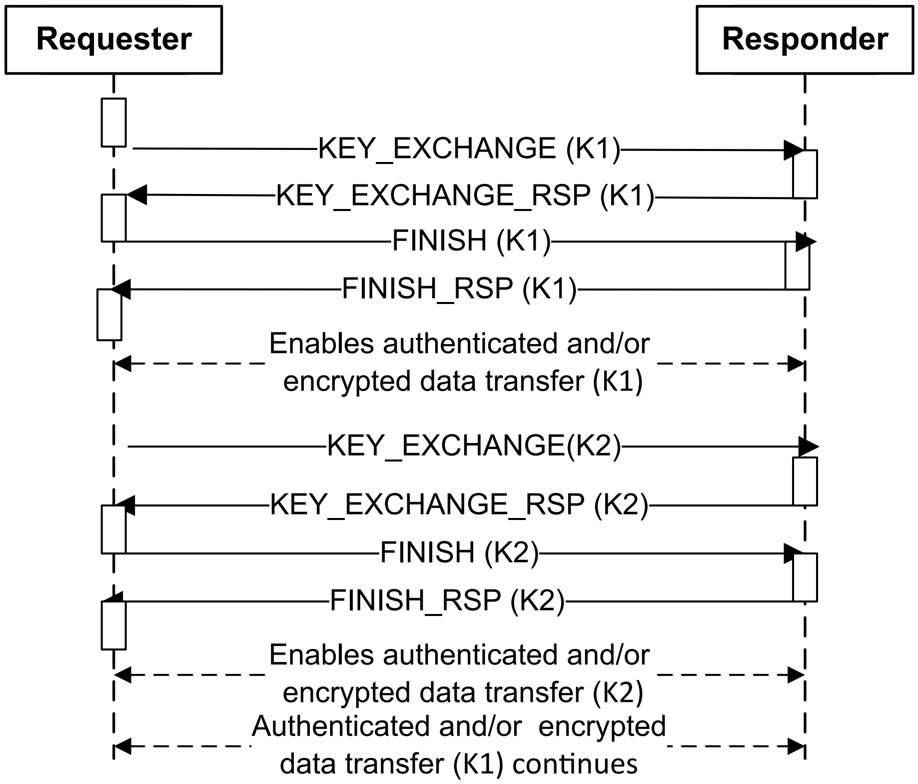


587 Figure 16 — Responder authentication multiple key exchange example shows an example of multiple sessions using two independent sets of root session keys that coexist at the same time. When `HANDSHAKE_IN_THE_CLEAR_CAP` = `0` for the Requester and/or Responder, the specification does not require a specific temporal relationship between the second `KEY_EXCHANGE` request message and the first `FINISH_RSP` response message. However, to simplify implementation, a Responder might respond with an `ERROR` message of `ErrorCode=Busy` to the second `KEY_EXCHANGE` request message until the first `FINISH_RSP` response message is complete. If the handshake is performed in the clear (that is, if `HANDSHAKE_IN_THE_CLEAR_CAP` = `1` for both Requester and Responder), a Requester shall not send a second `KEY_EXCHANGE` request message until the first `FINISH_RSP` response message is received. A

Responder shall respond with an `ERROR` message of `ErrorCode=UnexpectedRequest` if it receives a second `KEY_EXCHANGE` request message before the first `FINISH` request is received.

588 **Figure 16 — Responder authentication multiple key exchange example**

589



590 10.17.1 DHE scheme

591 If the Requester and Responder negotiated a DHE scheme during algorithm negotiation, the handshake includes an ephemeral Diffie-Hellman (DHE) key exchange in which the Requester and Responder each generate an ephemeral (that is, temporary) Diffie-Hellman key pair and exchange the public keys of those key pairs in the `ExchangeData` fields of the `KEY_EXCHANGE` request message and `KEY_EXCHANGE_RSP` response message. The Responder generates a DHE secret by using the private key of the DHE key pair of the Responder and the public key of the DHE key pair of the Requester provided in the `KEY_EXCHANGE` request message. Similarly, the Requester generates a DHE secret by using the private key of the DHE key pair of the Requester and the public key of the DHE key pair of the Responder provided in the `KEY_EXCHANGE_RSP` response message. The DHE secrets are computed as specified in clause 7.4 of RFC 8446. Assuming that the public keys were received correctly, both the Requester and Responder generate identical DHE secrets from which session secrets are generated.

592 Diffie-Hellman group parameters are determined by the DHE group in use, which is selected in the most recent

`ALGORITHMS` response. The contents of the `ExchangeData` field are computed as specified in clause 4.2.8 of RFC 8446. Specifically, if the DHE key exchange is based on finite-fields (FFDHE), the `ExchangeData` field in `KEY_EXCHANGE` and `KEY_EXCHANGE_RSP` shall contain the computed public value (Y = g^X mod p) for the specified group (see Table 17 — DHE structure for group definitions) encoded as a big-endian integer and padded to the left with zeros to the size of p in bytes. If the key exchange is based on elliptic curves (ECDHE), the `ExchangeData` field in `KEY_EXCHANGE` and `KEY_EXCHANGE_RSP` shall contain the serialization of X and Y, which are the binary representations of the x and y values respectively in network byte order, padded on the left by zeros if necessary. The size of each number representation occupies as many octets as are implied by the curve parameters selected. Specifically, X is [0: C - 1] and Y is [C : D - 1], where C and D are determined by the group (see Table 17 — DHE structure).

593 For SM2_P256 key exchange, the identifiers ID_A and ID_B that the GB/T 32918.3-2016 specification defines are needed to derive the shared secret. If this algorithm is selected, the ID for the Requester (that is, ID_A) shall be the concatenation of "Requester-KEP-dmtf-spdm-v" and `SPDMversionString`. Likewise, the ID for the Responder (that is, ID_B) shall be the concatenation of "Responder-KEP-dmtf-spdm-v" and `SPDMversionString`.

594 A Requester should generate a new DHE key pair for each `KEY_EXCHANGE` request message that the Requester sends. A Responder should generate a new DHE key pair for each `KEY_EXCHANGE_RSP` response message that the Responder sends.

595 10.17.2 ML-KEM scheme

596 If the Requester and Responder negotiated an ML-KEM scheme during algorithm negotiation, then the handshake starts with the Requester generating an ephemeral ML-KEM key pair and sending the encapsulation key `ek` to the Responder in the `KEY_EXCHANGE` request message. Upon receiving the Requester's `ek`, the Responder randomly generates 32-byte `m`, encapsulates `m` with `ek`, and arrives at ciphertext `c` and 32-byte shared secret key `K`. The ciphertext `c` alongside the Responder's signature and integrity MAC is sent to the Requester in the `KEY_EXCHANGE_RSP` response message for the Requester to decapsulate with the Requester's decapsulation key `dk`. The decapsulation results in shared secret `K'`.

597 A Requester should generate a new ML-KEM key pair for each `KEY_EXCHANGE` request message. A Responder should generate a new `m` for calculating `K` and `c` upon receiving each `KEY_EXCHANGE` request message.

598 10.17.3 Message formats

599 **Table 74 — KEY_EXCHANGE request message format**

Byte offset	Field	Size (bytes)	Description
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	RequestResponseCode	1	Shall be `0xE4` = `KEY_EXCHANGE`. See Table 4 — SPDM request codes.

Byte offset	Field	Size (bytes)	Description
2	Param1	1	Shall be the type of measurement summary hash requested: `0x0` : No measurement summary hash requested. `0x1` : TCB measurements only. `0xFF` : All measurements. All other values reserved. If a Responder does not support measurements (`MEAS_CAP=00b` in its `CAPABILITIES` response), the Requester shall set this value to `0x0` .
3	Param2	1	Shall be the `SlotID` . Slot number of the Responder certificate chain that shall be used for authentication. If the public key of the Responder was provisioned to the Requester in a trusted environment, the value in this field shall be `0xFF` ; otherwise it shall be between 0 and 7 inclusive.
4	ReqSessionID	2	Shall be the two-byte Requester contribution to allow construction of a unique four-byte session ID between a Requester-Responder pair. The final session ID (SessionID) = Concatenate(ReqSessionID, RspSessionID).
6	SessionPolicy	1	Shall be the session policy as Table 75 — Session policy defines.
7	Reserved	1	Reserved.
8	RandomData	32	Shall be the Requester-provided random data.

Byte offset	Field	Size (bytes)	Description
40	ExchangeData	D	If the Requester and Responder negotiated a DHE scheme during algorithm negotiation, then this field shall be the DHE public information generated by the Requester. If the DHE group selected in the most recent `ALGORITHMS` response is finite-field-based (FFDHE), the `ExchangeData` represents the computed public value. If the selected DHE group is elliptic-curve-based (ECDHE), the `ExchangeData` represents the X and Y values in network byte order. Specifically, X is [0: C - 1] and Y is [C : D - 1]. In both cases the size of D (and C for ECDHE) is derived from the selected DHE group, as described in Table 25 — DHE structure. If the Requester and Responder negotiated a KEM scheme during algorithm negotiation, then this field shall be the Requester's KEM encapsulation key `ek`. The value of D is specified in Table 30 — KEMAlg structure.
40 + `D`	OpaqueDataLength	2	Shall be the size of the `OpaqueData` field that follows in bytes. The value should not be greater than 1024 bytes. Shall be `0` if no `OpaqueData` is provided.
42 + `D`	OpaqueData	`OpaqueDataLength`	If present, shall be the `OpaqueData` sent by the Requester. Used to indicate any parameters that the Requester wishes to pass to the Responder as part of key exchange. If present, this field shall conform to the selected opaque data format in `OtherParamsSelection`.

600 **Table 75 — Session policy**

Bit offset	Field	Description
0	TerminationPolicy	This field specifies the behavior of the Responder when the Responder completes a runtime code or configuration update that affects the hardware or firmware measurement of the Responder. The Requester selects the value. If not set, the Responder shall terminate the session when the runtime update has taken effect. If set, the Responder shall decide whether to terminate or continue with the session based on its own policy. A policy example is one where the Responder terminates the session whenever an update to configuration or runtime code changes the security version of the firmware that manages SPDM sessions. The policy of the Responder is outside the scope of this specification. To terminate a session, the Responder shall either respond with an `ERROR` message of `ErrorCode=RequestResynch` to any SPDM request received within the session or silently discard any request received within the session until a `GET_VERSION` request is received.
1	EventAllPolicy	If set, the Responder shall subscribe the Requester to all events the Responder supports. Upon successfully entering the application phase of a session, the Responder may immediately send events. If set and `EVENT_CAP` is not set in `CAPABILITIES`, the Responder shall either respond with an `ERROR` message of `ErrorCode=InvalidRequest` or silently discard the request.
[7:2]	Reserved	Reserved

601 **Table 76 — Successful KEY_EXCHANGE_RSP response message format**

Byte offset	Field	Size (bytes)	Description
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	RequestResponseCode	1	Shall be `0x64` = `KEY_EXCHANGE_RSP`. See Table 5 — SPDM response codes.
2	Param1	1	Shall be HeartbeatPeriod. The value of this field shall be zero if Heartbeat is not supported by one of the endpoints. Otherwise, the value shall be in units of seconds. Zero is a legal value if Heartbeat is supported, and this means that a heartbeat is not desired on this session.
3	Param2	1	Reserved.

Byte offset	Field	Size (bytes)	Description
4	RspSessionID	2	Shall be the two-byte Responder contribution to allow construction of a unique four-byte session ID between a Requester-Responder pair. The final session ID = Concatenate(ReqSessionID, RspSessionID).
6	MutAuthRequested	1	Bit 0. If set, the Responder is requesting to authenticate the Requester (Session-based mutual authentication) without using the encapsulated request flow. Bit 1. If set, Responder is requesting Session-based mutual authentication with the encapsulated request flow. Bit 2. If set, Responder is requesting Session-based mutual authentication with an implicit `GET_DIGESTS` request. The Responder and Requester shall follow the optimized encapsulated request flow. Bit [7:3]. Reserved. At most one bit of Bit 0, Bit 1, or Bit 2 shall be set. For encapsulated request flow and the optimized encapsulated request flow details, see the GET_ENCAPSULATED_REQUEST request and ENCAPSULATED_REQUEST response messages clause.
7	SlotIDParam	1	Bit [7:4]. Reserved. Bit [3:0]. Shall be the `SlotID`. Slot number of the Requester certificate chain that shall be used for mutual authentication, if `MutAuthRequested` Bit 0 is set. If the public key of the Requester was provisioned to the Responder through other means, the value in this field shall be `0xF`; otherwise it shall be between 0 and 7 inclusive. All other values reserved. For any other value of `MutAuthRequested`, this field shall be set to `0` and ignored by the Requester.
8	RandomData	32	Shall be the Responder-provided random data.

Byte offset	Field	Size (bytes)	Description
40	ExchangeData	D'	If the Requester and Responder negotiated a DHE scheme during algorithm negotiation, then this field shall be the DHE public information generated by the Responder. If the DHE group selected in the most recent `ALGORITHMS` response is finite-field-based (FFDHE), the `ExchangeData` represents the computed public value. If the selected DHE group is elliptic-curve-based (ECDHE), the `ExchangeData` represents the X and Y values in network byte order. Specifically, X is [0: C' - 1] and Y is [C' : D' - 1]. In both cases the size of D' (and C' for ECDHE) is derived from the selected DHE group, as described in Table 25 — DHE structure. If the Requester and Responder negotiated a KEM scheme during algorithm negotiation, then this field shall be the ciphertext `c`. The value of D' is specified in Table 30 — KEMAlg structure.

Byte offset	Field	Size (bytes)	Description
40 + `D'`	MeasurementSummaryHash	`MSHLength` = `H` or 0	If the Responder does not support measurements (`MEAS_CAP=00b` in its `CAPABILITIES` response) or requested `Param1` = `0x0` , this field shall be absent. If the requested `Param1` = `0x1` , this field shall be the combined hash of measurements of all measurable components considered to be in the TCB required to generate this response, computed as `hash(Concatenate(MeasurementBlock[0], MeasurementBlock[1], ...))` , where `MeasurementBlock[x]` denotes a measurement of an element in the TCB and `hash` is the negotiated base hashing algorithm. Measurements are concatenated in ascending order based on their measurement index as Table 57 — Measurement block format describes. If the requested `Param1` = `0x1` and if there are no measurable components in the TCB required to generate this response, this field shall be `0` . If requested `Param1` = `0xFF` , this field shall be computed as `hash(Concatenate(MeasurementBlock[0], MeasurementBlock[1], ..., MeasurementBlock[n]))` of all supported measurements available in the measurement index range `0x01` - `0xFE` , concatenated in ascending index order. Indices with no associated measurements shall not be included in the hash calculation. See the Measurement index assignments clause. If the Responder supports both raw bit stream and digest representations for a given measurement index, the Responder shall use the digest form. This field shall be in hash byte order.
40 + `D'` + `MSHLength`	OpaqueDataLength	2	Shall be the size of the `OpaqueData` field that follows in bytes. The value should not be greater than 1024 bytes. Shall be `0` if no `OpaqueData` is provided.
42 + `D'` + `MSHLength`	OpaqueData	`OpaqueDataLength`	If present, shall be the `OpaqueData` sent by the Responder. Used to indicate any parameters that the Responder wishes to pass to the Requester as part of key exchange. If present, this field shall conform to the selected opaque data format in `OtherParamsSelection` .

Byte offset	Field	Size (bytes)	Description
42 + `D'` + `MSHLength` + `OpaqueDataLength`	Signature	`SigLen`	Shall be the `Signature` over the transcript. `SigLen` is the size of the asymmetric signing algorithm output the Responder selected via the last `ALGORITHMS` response message to the Requester. The Transcript for KEY_EXCHANGE_RSP signature defines the construction of the transcript.
42 + `D'` + `MSHLength` + `OpaqueDataLength` + `SigLen`	ResponderVerifyData	`H` or 0	Conditional field. If the Session Handshake Phase is encrypted and/or message authenticated, this field shall be of length H and shall equal the HMAC of the transcript hash, using `finished_key` as the secret key and using the negotiated hash algorithm as the hash function. The transcript hash shall be the hash of the transcript for `KEY_EXCHANGE_RSP` HMAC as Transcript for KEY_EXCHANGE_RSP HMAC shows. The `finished_key` shall be derived from the Response Direction Handshake Secret and is described in Finished_key derivation. HMAC is described in RFC 2104. If both the Requester and Responder set `HANDSHAKE_IN_THE_CLEAR_CAP` to 1, this field shall be absent.

602 **10.17.4 Session-based mutual authentication**

603 Mutual authentication for `KEY_EXCHANGE` occurs in the session handshake phase of a session.

604 To perform authentication of a Requester, the Responder sets the appropriate bit in the `MutAuthRequested` field of the `KEY_EXCHANGE_RSP` message. When either Bit 1 or Bit 2 of `MutAuthRequested` are set, the encapsulated request flow or the optimized encapsulated request flow shall be used accordingly to enable the Responder to obtain the certificate chains and certificate chain digests of the Requester. For flow details and illustrations, see GET_ENCAPSULATED_REQUEST request and ENCAPSULATED_REQUEST response messages.

605 When either bit 1 or bit 2 of `MutAuthRequested` are set, the only allowed messages in this phase of the session shall be `GET_DIGESTS` , `DIGESTS` , `GET_CERTIFICATE` , `CERTIFICATE` , and `ERROR` . If the Requester receives other requests during this flow, the Requester can respond with an `ERROR` message of `ErrorCode=UnexpectedRequest` and shall terminate the session.

606 If Bit 0 of `MutAuthRequested` is set, then mutual authentication shall be performed without exchanging any messages between `KEY_EXCHANGE_RSP` and `FINISH` request. This is useful for Responders that have obtained a Requester's certificate chains in a previous interaction.

607 **10.17.4.1 Specify Requester certificate for session-based mutual authentication**

608 The SPDM key exchange protocol is optimized to perform key exchange with the least number of messages exchanged. For Responder-only authentication and for mutual authentication where the Responder has obtained the certificate chains of the Requester in a previous interaction, key exchange is carried out with two request/response message pairs (`KEY_EXCHANGE` and `KEY_EXCHANGE_RSP` ; `FINISH` and `FINISH_RSP`). In other cases where mutual authentication is desired, additional encapsulated messages are exchanged between `KEY_EXCHANGE_RSP` and `FINISH` to enable the Responder to obtain the certificate chains and certificate chain digests of the Requester. However, in all cases the certificate chain (or raw public key) the Requester should authenticate against is specified by the Responder via the `SlotID` field in `KEY_EXCHANGE_RSP` , which precedes the aforementioned encapsulated messages. This means that a Responder has no way of knowing in advance which `SlotID` value to use when authenticating a Requester whose certificates it has not obtained in a previous interaction, other than the default (Slot 0).

609 To address this case, the Responder explicitly designates the certificate chain to be used via the final `ENCAPSULATED_RESPONSE_ACK` request issued inside the encapsulated request flow. Specifically, if either Bit 1 or 2 in `MutAuthRequested` is set to `1` , the Responder shall use an `ENCAPSULATED_RESPONSE_ACK` request with `Param2` = `0x02` and a 1-byte-long `Encapsulated Request` field containing the `SlotID` value. The Requester shall use the certificate chain corresponding to the slot specified in the `Encapsulated Request` field.

610 If Bit 0 of `MutAuthRequested` is set, then no encapsulated messages are exchanged after `KEY_EXCHANGE_RSP` and the certificate chain of the Requester is determined by the value of `SlotIDParam` in `KEY_EXCHANGE_RSP` .

611 10.18 FINISH request and FINISH_RSP response messages

612 This request message shall complete the handshake between Requester and Responder initiated by a `KEY_EXCHANGE` request. The purpose of the `FINISH` request and `FINISH_RSP` response messages is to provide key confirmation, bind the identity of each party to the exchanged keys and protect the entire handshake against manipulation by an active attacker. Upon receiving a `FINISH` request, the Responder shall ensure the session and the corresponding session ID were created through a `KEY_EXCHANGE` request and corresponding `KEY_EXCHANGE_RSP` response. Table 77 — FINISH request message format shows the `FINISH` request message format and Table 78 — Successful FINISH_RSP response message format shows the `FINISH_RSP` response message format.

613 **Table 77 — FINISH request message format**

Byte offset	Field	Size (bytes)	Description
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	RequestResponseCode	1	Shall be `0xE5` = `FINISH` . See Table 4 — SPDM request codes.
2	Param1	1	Bit 0. If set, the Signature field is included. This bit shall be set when Session-based mutual authentication occurs. All other bits reserved.

Byte offset	Field	Size (bytes)	Description
3	Param2	1	Shall be the `SlotID`. Only valid if `Param1` = `0x01`, otherwise reserved. Slot number of the Requester certificate chain that shall be used for authentication. If the public key of the Requester was provisioned to the Responder in a trusted environment, the value in this field shall be `0xFF`; otherwise it shall be between 0 and 7 inclusive.
4	Signature	SigLen	Shall be the `Signature` over the transcript. `SigLen` is the size of the asymmetric signing algorithm (`ReqBaseAsymAlg` or `ReqPqcAsymAlg`) output the Responder selected via the last `ALGORITHMS` response message to the Requester. If `Param1` = `0x00`, `SigLen` is zero and this field shall be absent. Transcript for FINISH signature, mutual authentication defines the construction of the transcript, signature generation, and verification.
4 + `SigLen`	RequesterVerifyData	H	Shall be an HMAC of the transcript hash using the `finished_key` as the secret key and using the negotiated hash algorithm as the hash function. For mutual authentication, the transcript hash shall be the hash of the transcript for `FINISH` HMAC, mutual authentication as the transcript for FINISH HMAC, mutual authentication shows. Otherwise, it shall be the hash of the transcript for `FINISH` HMAC, Responder-only authentication as the transcript for FINISH HMAC, Responder-only authentication shows. The `finished_key` shall be derived from Request Direction Handshake Secret and is described in Finished_key derivation. HMAC is described in RFC 2104.

614 If the handshake is performed in the clear (that is, if `HANDSHAKE_IN_THE_CLEAR_CAP` = `1` for both Requester and Responder), and if either Bit 1 or Bit 2 in `KEY_EXCHANGE_RSP` . `MutAuthRequested` is set, then upon receiving `FINISH` the Responder shall confirm that the value in `FINISH` . `Param2` matches the value that the Responder specified in the final `ENCAPSULATED_RESPONSE_ACK` . `EncapsulatedRequest` .

615 **Table 78 — Successful FINISH_RSP response message format**

Byte offset	Field	Size (bytes)	Description
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	RequestResponseCode	1	Shall be `0x65` = `FINISH_RSP`. See Table 5 — SPDM response codes.
2	Param1	1	Reserved.

Byte offset	Field	Size (bytes)	Description
3	Param2	1	Reserved.
4	ResponderVerifyData	`H` or 0	Conditional field. If the Session Handshake Phase is encrypted and/or message authenticated (that is, if either the Requester or the Responder set `HANDSHAKE_IN_THE_CLEAR_CAP` to 0), this field shall be absent. If both the Requester and Responder support `HANDSHAKE_IN_THE_CLEAR_CAP` field, this field shall be of length H and shall equal the HMAC of the transcript hash using `finished_key` as the secret key and using the negotiated hash algorithm as the hash function. For Session-based mutual authentication, the transcript hash shall be the hash of the transcript for `FINISH_RSP` HMAC, as the transcript for FINISH_RSP HMAC, mutual authentication shows. Otherwise, the transcript hash shall be the hash of the transcript for `FINISH_RSP` HMAC, Responder-only authentication as the transcript for FINISH_RSP HMAC, Responder-only authentication shows. The `finished_key` shall be derived from Response Direction Handshake Secret and is described in Finished_key derivation. HMAC is described in RFC 2104.

616 **10.18.1 Transcript and transcript hash calculation rules for KEY_EXCHANGE**

617 Transcript for KEY_EXCHANGE_RSP signature shows the transcript for the `KEY_EXCHANGE_RSP` signature:

618 **Transcript for KEY_EXCHANGE_RSP signature**

1. `VCA`
2. `[DIGESTS].*` (if issued and `MULTI_KEY_CONN_RSP` is true).
3. Hash of the specified certificate chain in DER format (that is, `Param2` of `KEY_EXCHANGE`) or hash of the public key in its provisioned format, if a certificate is not used.
4. `[KEY_EXCHANGE]`.`*`
5. `[KEY_EXCHANGE_RSP]`.`*` except the `Signature` and `ResponderVerifyData` fields.

619 The Responder shall generate the `KEY_EXCHANGE_RSP` signature from:

```
SPDMsign(PrivKey, transcript, "key_exchange_rsp signing");
```

620 where

- `SPDMsign` is described by the Signature generation clause.
- `PrivKey` shall be the private key of the Responder associated with the leaf certificate stored in `SlotID` in `KEY_EXCHANGE`. If the public key of the Responder was provisioned to the Requester, then `PrivKey` shall be the associated private key.
- `transcript` shall be the concatenation of the messages for a `KEY_EXCHANGE_RSP` signature.

621 The leaf certificate of the Responder shall be the one indicated by `SlotID` in `Param2` of `KEY_EXCHANGE` request.

622 Likewise, the Requester shall verify the `KEY_EXCHANGE_RSP` signature using `SPDMsignatureVerify(PubKey, signature, transcript, "key_exchange_rsp signing")`, where `transcript` is the concatenation of the messages for a `KEY_EXCHANGE_RSP` signature and `PubKey` is the public key of the leaf certificate of the Responder. The leaf certificate of the Responder shall be the one indicated by `SlotID` in `Param2` of `KEY_EXCHANGE` request. `SPDMsignatureVerify` is described in Signature verification. A successful verification shall be when `SPDMsignatureVerify` returns `success`.

623 Transcript for KEY_EXCHANGE_RSP HMAC shows the transcript for `KEY_EXCHANGE_RSP` HMAC:

624 **Transcript for `KEY_EXCHANGE_RSP` HMAC**

1. `VCA`
2. `[DIGESTS].*` (if issued and `MULTI_KEY_CONN_RSP` is true).
3. Hash of the specified certificate chain in DER format (that is, `Param2` of `KEY_EXCHANGE`) or hash of the public key in its provisioned format, if a certificate is not used.
4. `[KEY_EXCHANGE].*`
5. `[KEY_EXCHANGE_RSP].*` except the `ResponderVerifyData` field.

625 Transcript for FINISH signature, mutual authentication shows the transcript for the `FINISH` signature with mutual authentication:

626 **Transcript for `FINISH` signature, mutual authentication**

1. `VCA`
2. `[DIGESTS].*` (if issued and `MULTI_KEY_CONN_RSP` is true).
3. Hash of the specified certificate chain in DER format (that is, `Param2` of `KEY_EXCHANGE`) or hash of the public key in its provisioned format, if a certificate is not used.
4. `[KEY_EXCHANGE].*`
5. `[KEY_EXCHANGE_RSP].*`
6. `[DIGESTS].*` (if encapsulated `DIGESTS` is issued and `MULTI_KEY_CONN_REQ` is true).
7. Hash of the specified certificate chain in DER format (that is, `Param2` of `FINISH`) or hash of the public key in its provisioned format, if a certificate is not used.
8. `[FINISH].SPDM Header Fields`

627 The Requester shall generate the `FINISH` signature from `SPDMsign(PrivKey, transcript, "finish signing")`, where `transcript` is the concatenation of the messages for `FINISH` signature and the `PrivKey` is the private key of the leaf certificate of the Requester. The leaf certificate of the Requester shall be the one indicated in `SlotID` in `Param2` of `FINISH` request. `SPDMsign` is described in Signature generation.

628 Likewise, the Responder shall verify the `FINISH` signature using `SPDMsignatureVerify(PubKey, signature,`

`transcript, "finish signing")`, where `transcript` is the concatenation of the messages for a `FINISH` signature and the `PubKey` is the public key of the leaf certificate of the Requester. The leaf certificate of the Requester shall be the one indicated in `SlotID` in `Param2` of the `FINISH` request. `SPDMsignatureVerify` is described in Signature verification. A successful verification is when `SPDMsignatureVerify` returns `success`.

629 Transcript for FINISH HMAC, Responder-only authentication shows the transcript for `FINISH` HMAC with Responder-only authentication:

630 **Transcript for `FINISH` HMAC, Responder-only authentication**

1. `VCA`
2. `[DIGESTS].*` (if issued and `MULTI_KEY_CONN_RSP` is true).
3. Hash of the specified certificate chain in DER format (that is, `Param2` of `KEY_EXCHANGE`) or hash of the public key in its provisioned format, if a certificate is not used.
4. `[KEY_EXCHANGE]`.`*`
5. `[KEY_EXCHANGE_RSP]`.`*`
6. `[FINISH]`.`SPDM Header Fields`

631 Transcript for FINISH HMAC, mutual authentication shows the transcript for `FINISH` HMAC with mutual authentication:

632 **Transcript for `FINISH` HMAC, mutual authentication**

1. `VCA`
2. `[DIGESTS].*` (if issued and `MULTI_KEY_CONN_RSP` is true).
3. Hash of the specified certificate chain in DER format (that is, `Param2` of `KEY_EXCHANGE`) or hash of the public key in its provisioned format, if a certificate is not used.
4. `[KEY_EXCHANGE]`.`*`
5. `[KEY_EXCHANGE_RSP]`.`*`
6. `[DIGESTS].*` (if encapsulated `DIGESTS` is issued and `MULTI_KEY_CONN_REQ` is true).
7. Hash of the specified certificate chain in DER format (that is, `Param2` of `FINISH`) or hash of the public key in its provisioned format, if a certificate is not used.
8. `[FINISH]`.`SPDM Header Fields`
9. `[FINISH]`.`Signature`

633 Transcript for FINISH_RSP HMAC, Responder-only authentication shows the transcript for `FINISH_RSP` HMAC with Responder-only authentication:

634 **Transcript for `FINISH_RSP` HMAC, Responder-only authentication**

1. `VCA`
2. `[DIGESTS].*` (if issued and `MULTI_KEY_CONN_RSP` is true).
3. Hash of the specified certificate chain in DER format (that is, `Param2` of `KEY_EXCHANGE`) or hash of the public key in its provisioned format, if a certificate is not used.
4. `[KEY_EXCHANGE]`.`*`
5. `[KEY_EXCHANGE_RSP]`.`*`

6. `[FINISH]` . `*`
7. `[FINISH_RSP]` . `SPDM Header fields`

635 Transcript for FINISH_RSP HMAC, mutual authentication shows the transcript for `FINISH_RSP` HMAC with mutual authentication:

636 **Transcript for `FINISH_RSP` HMAC, mutual authentication**

1. `VCA`
2. `[DIGESTS].*` (if issued and `MULTI_KEY_CONN_RSP` is true).
3. Hash of the specified certificate chain in DER format (that is, `Param2` of `KEY_EXCHANGE`) or hash of the public key in its provisioned format, if a certificate is not used.
4. `[KEY_EXCHANGE]` . `*`
5. `[KEY_EXCHANGE_RSP]` . `*`
6. `[DIGESTS].*` (if encapsulated `DIGESTS` is issued and `MULTI_KEY_CONN_REQ` is true).
7. Hash of the specified certificate chain in DER format (that is, `Param2` of `FINISH`) or hash of the public key in its provisioned format, if a certificate is not used.
8. `[FINISH]` . `*`
9. `[FINISH_RSP]` . `SPDM Header fields`

637 When multiple session keys are being established between the same Requester-Responder pair, the `Signature` over the transcript during `FINISH` request is computed using only the corresponding `KEY_EXCHANGE` , `KEY_EXCHANGE_RSP` , and `FINISH` request parameters.

638 For additional rules, see general ordering rules.

639 10.19 PSK_EXCHANGE request and PSK_EXCHANGE_RSP response messages

640 The Pre-Shared Key (PSK) key exchange scheme provides an option for a Requester and a Responder to perform session key establishment with symmetric-key cryptography. This option is especially useful for endpoints that do not support asymmetric-key cryptography or certificate processing. This option can also be leveraged to expedite session key establishment even if asymmetric-key cryptography is supported.

641 This option requires the Requester and Responder to have prior knowledge of a common PSK before the handshake. Essentially, the PSK serves as a mutual authentication credential and as the base of session key establishment. As such, only the two endpoints and potentially a trusted third party that provisions the PSK to the two endpoints know the value of the PSK. For these same reasons, the `HANDSHAKE_IN_THE_CLEAR_CAP` is not applicable in a PSK key exchange. Thus, for PSK-based session establishment, both the Responder and the Requester shall ignore the `HANDSHAKE_IN_THE_CLEAR_CAP` bit.

642 A Requester can pair with multiple Responders. Likewise, a Responder can pair with multiple Requesters. A Requester-Responder pair can be provisioned with one or more PSKs. An endpoint can act as a Requester to one device and simultaneously a Responder to another device. If both endpoints can act as Requester or Responder,

then the endpoints shall use different PSKs for each role. It is the responsibility of the transport layer to identify the peer and establish communication between the two endpoints before the PSK-based session key exchange starts.

643 The PSK can be provisioned in a trusted environment, for example, during the secure manufacturing process. In an untrusted environment, the PSK can be agreed upon between the two endpoints using a secure protocol. The mechanism for PSK provisioning is outside the scope of this specification. The size of the provisioned PSK is determined by the security strength requirements of the application, but it should be at least 128 bits. It is recommended to be at least 256 bits in order to resist dictionary attacks, particularly when the Requester and Responder cannot both contribute sufficient entropy during the exchange.

644 Two message pairs are defined for this option:

- `PSK_EXCHANGE` / `PSK_EXCHANGE_RSP`
- `PSK_FINISH` / `PSK_FINISH_RSP`

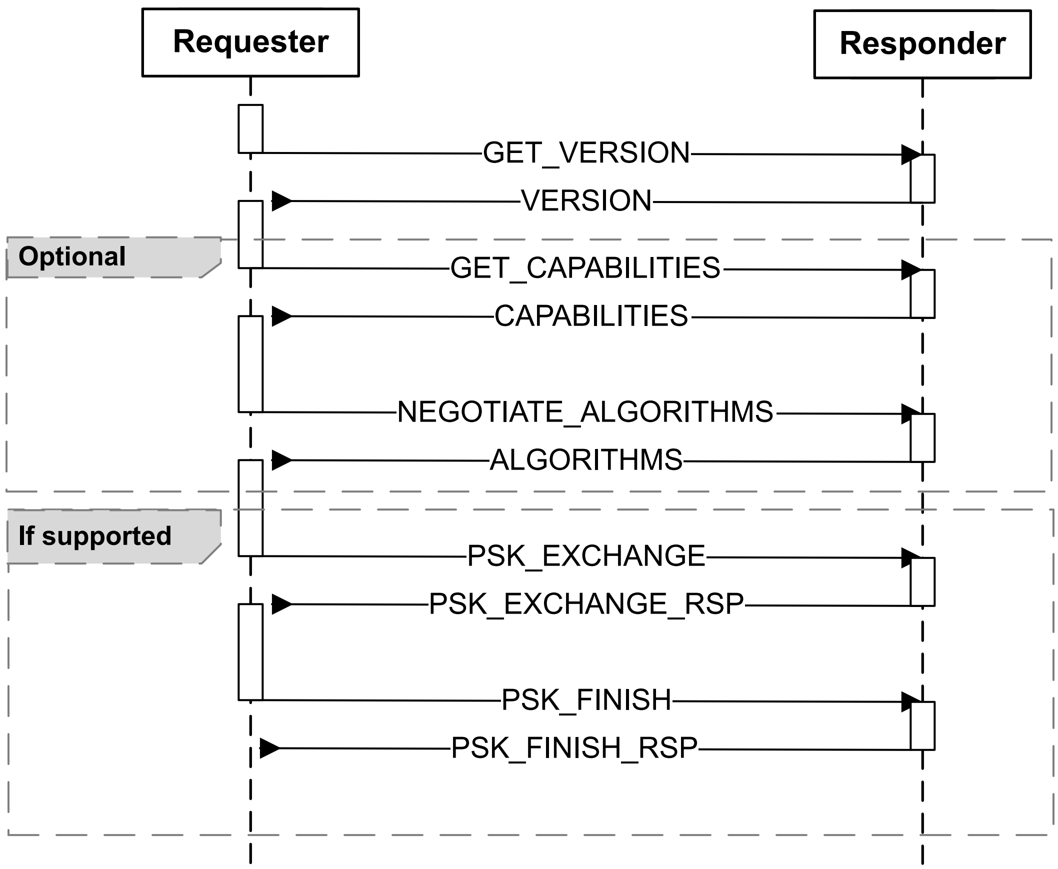
645 The `PSK_EXCHANGE` message carries three responsibilities:

1. Prompts the Responder to retrieve the specific PSK.
2. Exchanges contextual information between the Requester and the Responder.
3. Proves to the Requester that the Responder knows the correct PSK and has derived the correct session keys.

646 Figure 17 — PSK_EXCHANGE: Example shows an example of the `PSK_EXCHANGE` message:

647 **Figure 17 — PSK_EXCHANGE: Example**

648



649 **Table 79 — PSK_EXCHANGE request message format**

Byte offset	Field	Size (bytes)	Description
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	RequestResponseCode	1	Shall be `0xE6` = `PSK_EXCHANGE`. See Table 4 — SPDM request codes.

Byte offset	Field	Size (bytes)	Description
2	Param1	1	Shall be the type of measurement summary hash requested: `0x0` : No measurement summary hash requested. `0x1` : TCB measurements only. `0xFF` : All measurements. All other values reserved. If a Responder does not support measurements (`MEAS_CAP=00b` in its `CAPABILITIES` response), the Requester shall set this value to `0x0` .
3	Param2	1	Shall be the session policy. See Table 75 — Session policy.
4	ReqSessionID	2	Shall be the two-byte Requester contribution to allow construction of a unique four-byte session ID between a Requester-Responder pair. The final session ID = Concatenate(ReqSessionID, RspSessionID).
6	P	2	Shall be the length of `PSKHint` in bytes.
8	R	2	Shall be the length of `RequesterContext` in bytes.
10	OpaqueDataLength	2	Shall be the size of the `OpaqueData` field that follows in bytes. The value should not be greater than 1024 bytes. Shall be `0` if no `OpaqueData` is provided.
12	PSKHint	`P`	Shall be the information required by the Responder to retrieve the PSK. Optional.
12 + `P`	RequesterContext	`R`	Shall be the context of the Requester. Shall include a nonce or non-repeating counter of at least 32 bytes and, optionally, relevant information contributed by the Requester.
12 + `P` + `R`	OpaqueData	`OpaqueDataLength`	Optional. If present, the `OpaqueData` sent by the Requester is used to indicate any parameters that the Requester wishes to pass to the Responder as part of PSK-based key exchange. If present, this field shall conform to the selected opaque data format in `OtherParamsSelection` .

650 The field `PSKHint` is optional. It is absent if P is set to 0. It is introduced to address two scenarios:

- The Responder is provisioned with multiple PSKs and stores them in secure storage. The Requester uses `PSKHint` as an identifier to specify which PSK will be used in this particular session.

- The Responder does not store the actual value of the PSK but can derive the PSK using `PSKHint`. For example, if the Responder has an immutable UDS (Unique Device Secret) in fuses, then during provisioning a PSK can be derived from the UDS (or a derivative value) and a non-secret salt known by the Requester. During session key establishment, the salt value is sent to the Responder in `PSKHint` of `PSK_EXCHANGE`. This mechanism allows the Responder to support any number of PSKs without consuming secure storage.

651 The `RequesterContext` is the contribution of the Requester to session key derivation. It shall contain a nonce or non-repeating counter to ensure that the derived session keys are ephemeral to mitigate against replay attacks. If a non-repeating counter is used, the counter shall not be reset for the lifetime of the device. The `RequesterContext` can also contain other information from the Requester.

652 Upon receiving a `PSK_EXCHANGE` request, the Responder:

1. Generates PSK from `PSKHint`, if necessary.
2. Generates `ResponderContext`, if supported.
3. Derives the `finished_key` of the Responder by following the key schedule.
4. Constructs the `PSK_EXCHANGE_RSP` response message and sends it to the Requester.

653 **Table 80 — PSK_EXCHANGE_RSP response message format**

Byte offset	Field	Size (bytes)	Description
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	RequestResponseCode	1	Shall be `0x66` = `PSK_EXCHANGE_RSP`. See Table 5 — SPDM response codes.
2	Param1	1	Shall be HeartbeatPeriod. The value of this field shall be zero if Heartbeat is not supported by one of the endpoints. Otherwise, the value shall be in units of seconds. Zero is a legal value if Heartbeat is supported, and this means that a heartbeat is not desired on this session.
3	Param2	1	Reserved.
4	RspSessionID	2	Shall be the two-byte Responder contribution to allow construction of a unique four-byte session ID between a Requester-Responder pair. The final session ID (SessionID) = Concatenate(ReqSessionID, RspSessionID).
6	Reserved	2	Reserved.
8	Q	2	Shall be the length of ResponderContext in bytes.
10	OpaqueDataLength	2	Shall be the size of the `OpaqueData` field that follows in bytes. The value should not be greater than 1024 bytes. Shall be `0` if no `OpaqueData` is provided.

Byte offset	Field	Size (bytes)	Description
12	MeasurementSummaryHash	`MSHLength` = `H` or 0	If the Responder does not support measurements (`MEAS_CAP=00b` in its `CAPABILITIES` response) or requested `Param1` = `0x0` , this field shall be absent. If the requested `Param1` = `0x1` , this field shall be the combined hash of measurements of all measurable components considered to be in the TCB required to generate this response, computed as `hash(Concatenate(MeasurementBlock[0], MeasurementBlock[1], ...))` , where `MeasurementBlock[x]` denotes a measurement of an element in the TCB and `hash` is the negotiated base hashing algorithm. Measurements are concatenated in ascending order based on their measurement index as Table 57 — Measurement block format describes. If the requested `Param1` = `0x1` and if there are no measurable components in the TCB required to generate this response, this field shall be `0` . If requested `Param1` = `0xFF` , this field shall be computed as `hash(Concatenate(MeasurementBlock[0], MeasurementBlock[1], ..., MeasurementBlock[n]))` of all supported measurements available in the measurement index range `0x01` - `0xFE` , concatenated in ascending index order. Indices with no associated measurements shall not be included in the hash calculation. See the Measurement index assignments clause. If the Responder supports both raw bit stream and digest representations for a given measurement index, the Responder shall use the digest form. This field shall be in hash byte order.
12 + `MSHLength`	ResponderContext	`Q`	Shall be the context of the Responder. Optional. If present, shall include a nonce and/or information contributed by the Responder.
12 + `MSHLength` + `Q`	OpaqueData	`OpaqueDataLength`	Optional. If present, the `OpaqueData` sent by the Responder is used to indicate any parameters that the Responder wishes to pass to the Requester as part of PSK-based key exchange. If present, this field shall conform to the selected opaque data format in `OtherParamsSelection` .

Byte offset	Field	Size (bytes)	Description
12 + `MSHLength` + `Q` + `OpaqueDataLength`	ResponderVerifyData	`H`	Shall be the data to be verified by the Requester using the `finished_key` of the Responder.

654 The `ResponderContext` is the contribution of the Responder to session key derivation. It should contain a nonce or non-repeating counter and other information from the Responder. If a non-repeating counter is used, the counter shall not be reset for the lifetime of the device. Because the Responder can be a constrained device that cannot generate a nonce, `ResponderContext` is optional. However, the Responder is required to use `ResponderContext` if it can generate a nonce.

655 Note that the nonce in `ResponderContext` is critical for anti-replay. If a nonce is not present in `ResponderContext`, then the Responder is not challenging the Requester for real-time knowledge of the PSK. Such a session is subject to replay attacks—that is, a person-in-the-middle attacker could record and replay prior `PSK_EXCHANGE` and `PSK_FINISH` messages and set up a session with the Responder. But the bogus session would not leak secrets, so long as the PSK and session keys of the prior replayed session are not compromised.

656 If `ResponderContext` is absent, such as when `PSK_CAP` in the `CAPABILITIES` of the Responder is `01b`, the Requester shall not send `PSK_FINISH`, because the session keys are solely determined by the Requester and the Session immediately enters the Application Phase. If and only if the `ResponderContext` is present in the response, such as when `PSK_CAP` in the `CAPABILITIES` of the Responder is `10b`, the Requester shall send `PSK_FINISH` with `RequesterVerifyData` to prove that it has derived correct session keys.

657 To calculate `ResponderVerifyData`, the Responder calculates an HMAC. The HMAC key is the `finished_key` of the Responder. The data is the hash of the concatenation of all messages sent up to this point between the Requester and the Responder. For messages that are encrypted, the plaintext messages are used in calculating the hash.

```
1. [GET_VERSION].*
2. [VERSION].*
3. [GET_CAPABILITIES].* (if issued)
4. [CAPABILITIES].* (if issued)
5. [NEGOTIATE_ALGORITHMS].* (if issued)
6. [ALGORITHMS].* (if issued)
7. [PSK_EXCHANGE].*
8. [PSK_EXCHANGE_RSP].* except the ResponderVerifyData field
```

658 Note that, even if `CERTIFICATE` and Responder-signed response messages (such as `CHALLENGE_AUTH`) were issued, these messages would not be included in the data for calculating `ResponderVerifyData`. In other words, the identity of the signer of the response messages is not bound to the identity of the sender of `PSK_EXCHANGE_RSP`. Therefore, to mitigate Responder identity impersonation, if the Requester has received a response with a signature and if there is no cryptographic binding between the signer of the Responder-signed response and the sender of `PSK_EXCHANGE_RSP`, then the Requester should not issue `PSK_EXCHANGE`. The method of cryptographic binding between the signer of the Responder-signed response and the sender of `PSK_EXCHANGE_RSP` is outside the scope of this specification.

659 Upon receiving `PSK_EXCHANGE_RSP`, the Requester:

1. Derives the `finished_key` of the Responder by following the key schedule.

2. Verifies `ResponderVerifyData` by calculating the HMAC in the same manner as the Responder. If verification fails, the Requester terminates the session.
3. If the Responder contributes to session key derivation, such as when the `ResponderContext` field is present in the `PSK_EXCHANGE_RSP` response, it constructs the `PSK_FINISH` request and sends it to the Responder.

660 If a successful `PSK_EXCHANGE_RSP` has been received by the Requester, and the `PSK_CAP` of the Responder is `10b`, and the `ResponderContext` field is present in the `PSK_EXCHANGE_RSP` response then, for the session ID created by the `PSK_EXCHANGE` and `PSK_EXCHANGE_RSP` messages, the next request shall be `PSK_FINISH`.

661 10.20 PSK_FINISH request and PSK_FINISH_RSP response messages

662 These messages shall complete the mutually-authenticated handshake between Requester and Responder initiated by a `PSK_EXCHANGE` request. The `PSK_FINISH` request proves to the Responder that the Requester knows the PSK and has derived the correct session keys. This is achieved by an HMAC value calculated with the `finished_key` of the Requester and messages of this session. The Requester shall send `PSK_FINISH` only if `ResponderContext` is present in `PSK_EXCHANGE_RSP`. Upon receiving a `PSK_FINISH` request, the Responder shall ensure the session and the corresponding session ID were created through a `PSK_EXCHANGE` request and corresponding `PSK_EXCHANGE_RSP` response.

663 Table 81 — PSK_FINISH request message format describes the `PSK_FINISH` request message format:

664 **Table 81 — PSK_FINISH request message format**

Byte offset	Field	Size (bytes)	Description
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	RequestResponseCode	1	Shall be `0xE7` = `PSK_FINISH`. See Table 4 — SPDM request codes.
2	Param1	1	Reserved.
3	Param2	1	Reserved.
4	RequesterVerifyData	`H`	Shall be the data to be verified by the Responder using the `finished_key` of the Requester.

665 To calculate `RequesterVerifyData`, the Requester calculates an HMAC. The key is the `finished_key` of the Requester, as described in the Key schedule clause. The data is the hash of the concatenation of all messages sent so far between the Requester and the Responder. For messages that are encrypted, the plaintext messages are used in calculating the hash.

```
1. [GET_VERSION].*
2. [VERSION].*
3. [GET_CAPABILITIES].* (if issued)
```

```
4. [CAPABILITIES].* (if issued)
5. [NEGOTIATE_ALGORITHMS].* (if issued)
6. [ALGORITHMS].* (if issued)
7. [PSK_EXCHANGE].*
8. [PSK_EXCHANGE_RSP].*
9. [PSK_FINISH].* except the RequesterVerifyData field
```

666 For additional rules, see general ordering rules.

667 Upon receiving the `PSK_FINISH` request, the Responder derives the `finished_key` of the Requester and calculates the HMAC independently in the same manner and verifies that the result matches `RequesterVerifyData`. If verification is successful, the Responder constructs the `PSK_FINISH_RSP` response and sends it to the Requester. Otherwise, the Responder sends the Requester an `ERROR` message of `ErrorCode=InvalidRequest`.

668 Table 82 — Successful PSK_FINISH_RSP response message format describes the successful `PSK_FINISH_RSP` response message format:

669 **Table 82 — Successful PSK_FINISH_RSP response message format**

Byte offset	Field	Size (bytes)	Description
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	RequestResponseCode	1	Shall be `0x67` = `PSK_FINISH_RSP`. See Table 5 — SPDM response codes.
2	Param1	1	Reserved.
3	Param2	1	Reserved.

670 10.21 HEARTBEAT request and HEARTBEAT_ACK response messages

671 This request shall keep a session alive if `HEARTBEAT` is supported by both the Requester and Responder. The `HEARTBEAT` request shall be sent periodically as indicated in `HeartbeatPeriod` in either the `KEY_EXCHANGE_RSP` or `PSK_EXCHANGE_RSP` response messages if no other messages are received in this secure session in the `HeartbeatPeriod`. The Responder shall terminate the session if session traffic is not received for two successive `HeartbeatPeriod` s. Likewise, the Requester shall terminate the session if session traffic, including `ERROR` responses, is not received for two successive `HeartbeatPeriod` s. Session traffic includes encrypted data at the transport layer. How an SPDM endpoint is informed of encrypted data at the transport layer is outside the scope of this specification. The Requester can retry `HEARTBEAT` requests.

672 The timer for the Heartbeat period shall start at either the transmission (for Responders) or the reception (for Requesters) of the appropriate `FINISH_RSP`, `PSK_FINISH_RSP` (`PSK_CAP` of Responder is `10b`), or `PSK_EXCHANGE_RSP` (`PSK_CAP` of Responder is `01b`) response messages. When determining the value of `HeartbeatPeriod`, the Responder should ensure this value is sufficiently greater than `T1`.

673 Each secure session shall track the heartbeat period independently of other sessions within the same SPDM Connection.

674 For session termination details, see session termination phase.

675 Table 83 — HEARTBEAT request message format describes the message format.

676 **Table 83 — HEARTBEAT request message format**

Byte offset	Field	Size (bytes)	Description
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	RequestResponseCode	1	Shall be `0xE8` = `HEARTBEAT` request. See Table 4 — SPDM request codes.
2	Param1	1	Reserved.
3	Param2	1	Reserved.

677 Table 84 — HEARTBEAT_ACK response message format describes the format for the Heartbeat Response.

678 **Table 84 — HEARTBEAT_ACK response message format**

Byte offset	Field	Size (bytes)	Description
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	RequestResponseCode	1	Shall be `0x68` = `HEARTBEAT_ACK` response. See Table 5 — SPDM response codes.
2	Param1	1	Reserved.
3	Param2	1	Reserved.

679 10.21.1 Heartbeat additional information

680 The transport layer might allow the `HEARTBEAT` request to be sent from the Responder to the Requester. This is recommended for transports capable of asynchronous bidirectional communication.

681 10.22 KEY_UPDATE request and KEY_UPDATE_ACK response messages

682 This request shall be used to update session keys. There are many reasons for doing this, but an important one is when the per-record nonce will soon reach its maximum value and roll over. The `KEY_UPDATE` request can also be issued by the Responder using the `GET_ENCAPSULATED_REQUEST` mechanism. A `KEY_UPDATE` request shall perform the operation given in `Param1` and defined in Table 87 — KEY_UPDATE operations. Because the Responder can also send this request, it is possible that two simultaneous key updates, one for each direction, can occur. However, only one `KEY_UPDATE` request for a single direction shall occur at a time. Until the session key update synchronization

successfully completes, subsequent `KEY_UPDATE` requests for the same direction shall be considered a retry of the original `KEY_UPDATE` request.

683 Table 85 — KEY_UPDATE request message format describes the `KEY_UPDATE` request message format:

684 **Table 85 — KEY_UPDATE request message format**

Byte offset	Field	Size (bytes)	Description
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	RequestResponseCode	1	Shall be `0xE9` = `KEY_UPDATE` Request. See Table 4 — SPDM request codes.
2	Param1	1	Shall indicate the key operation. See Table 87 — KEY_UPDATE operations.
3	Param2	1	Shall be the requesting SPDM endpoint assigned tag. This field shall contain a unique number to aid the responding SPDM endpoint in differentiating between the original and any retry requests. A retry request shall contain the same tag number as the original.

685 Table 86 — KEY_UPDATE_ACK response message format describes the `KEY_UPDATE_ACK` response message format:

686 **Table 86 — KEY_UPDATE_ACK response message format**

Byte offset	Field	Size (bytes)	Description
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	RequestResponseCode	1	Shall be `0x69` = `KEY_UPDATE_ACK` response. See Table 5 — SPDM response codes.
2	Param1	1	Shall indicate the key operation. This field shall reflect the Key Operation field of the request. See Table 87 — KEY_UPDATE operations
3	Param2	1	Shall be the tag. This field shall reflect the Tag number (`Param2`) from the `KEY_UPDATE` request.

687 Table 87 — KEY_UPDATE operations describes the `KEY_UPDATE` operations:

688 **Table 87 — KEY_UPDATE operations**

Value	Operation	Description
0	Reserved	Reserved.

Value	Operation	Description
1	UpdateKey	Shall update only the single-direction key associated with the direction of the request.
2	UpdateAllKeys	Shall update the keys for both directions.
3	VerifyNewKey	Shall ensure that the key update is successful and that old keys can be safely discarded.
4 - 255	Reserved	Reserved.

689 ### 10.22.1 Session key update synchronization

690 In the key update process, to clarify, the term "sender" means the SPDM endpoint that issued the `KEY_UPDATE` request, and the term "receiver" means the SPDM endpoint that received the `KEY_UPDATE` request. To ensure the key update process is seamless while still allowing the transmission and reception of records, both sender and receiver shall follow the prescribed method described in this clause.

691 The data transport layer shall ensure that data transfer during key updates is managed in such a way that the correct keys are used before, during, and after the key update operation. How this is accomplished by the data transport layer is outside the scope of this specification.

692 Both the sender and the receiver shall derive the new keys as detailed in Major secrets update.

693 The sender shall not use the new transmit key until after reception of the `KEY_UPDATE_ACK` response.

694 The sender and receiver shall use the new key on the `KEY_UPDATE` request with the `VerifyNewKey` command and all subsequent commands until another key update is performed.

695 In the case of a `KEY_UPDATE` request with `UpdateAllKeys`, the receiver shall use the new transmit key for the `KEY_UPDATE_ACK` response. The `KEY_UPDATE` request with `UpdateAllKeys` should only be used with physical transports that are single master to ensure that simultaneous `UpdateAllKeys` requests do not occur.

696 If the sender has not received `KEY_UPDATE_ACK`, the sender can retry or end the session. The sender shall not proceed to the next step until successfully receiving the corresponding `KEY_UPDATE_ACK`.

697 Upon the successful reception of the `KEY_UPDATE_ACK`, the sender shall transmit a `KEY_UPDATE` request with the `VerifyNewKey` operation using the new session keys. The sender can retry until the corresponding `KEY_UPDATE_ACK` response is received. However, the sender shall be prohibited, at this point, from restarting this process or going back to a previous step. Its only recourse in error handling is either to retry the same request or to terminate the session.

698 For `UpdateKey`, upon successful reception and verification of the `KEY_UPDATE` with the `VerifyNewKey` operation, the receiver can discard the old session keys. For `UpdateAllKeys`, upon successful reception and verification of the `KEY_UPDATE_ACK` with the `UpdateAllKeys` operation, the sender can discard the old session keys that protect receiver-sent messages. Upon successful reception and verification of the `KEY_UPDATE` with the `VerifyNewKey` operation, the receiver can discard the old session keys that protect sender-sent messages.

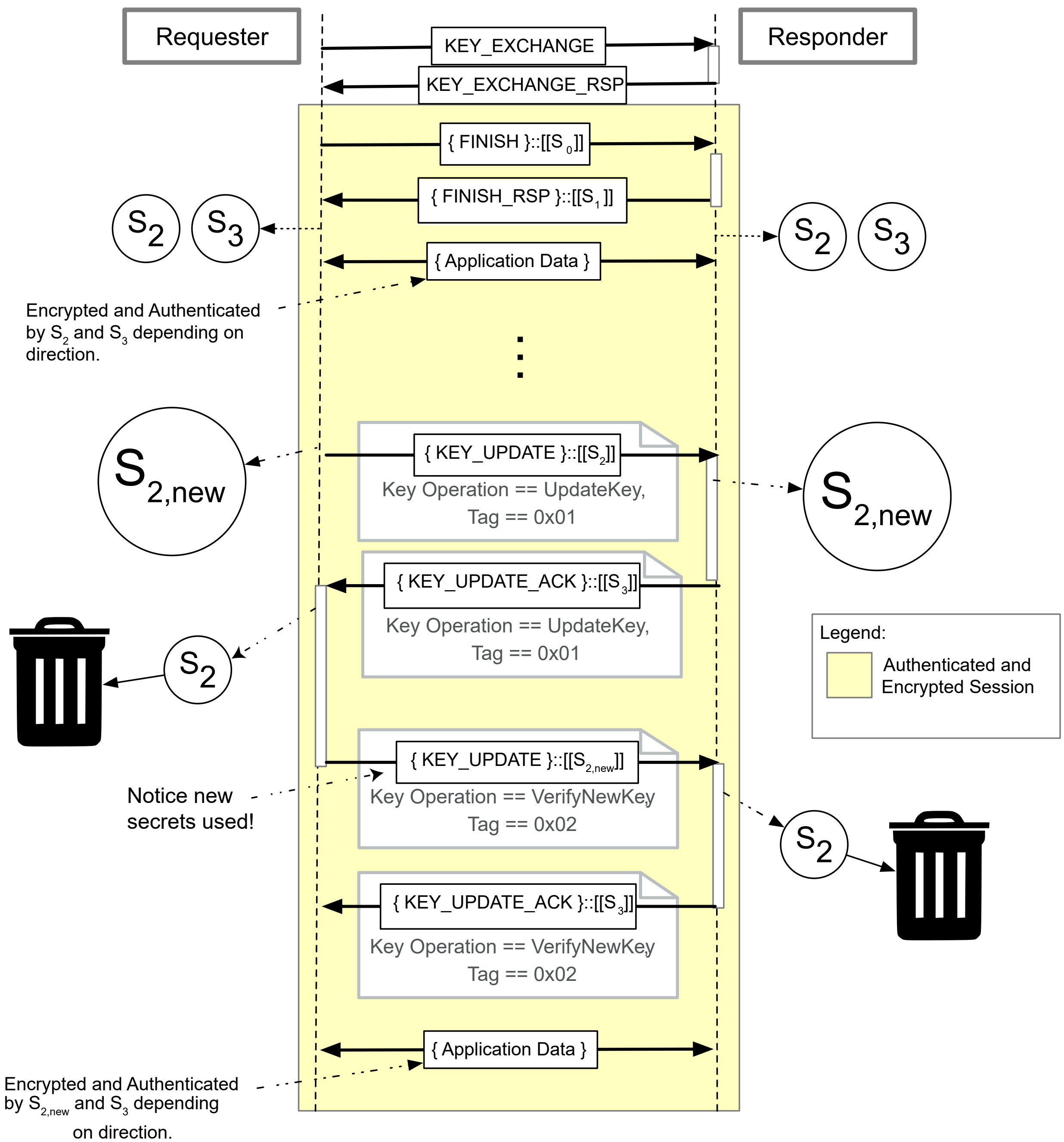
699 In certain scenarios, the receiver might need additional time to process the `KEY_UPDATE` request such as when

processing data already in its buffer. Thus, the receiver can reply with an `ERROR` message of `ErrorCode=Busy`. The sender should retry the request after a reasonable period of time and with a reasonable number of retries to prevent premature session termination.

700 Finally, it bears repeating that a key update in one direction can happen simultaneously with a key update in the opposite direction. In this case, the aforementioned synchronization process occurs independently but simultaneously for each direction.

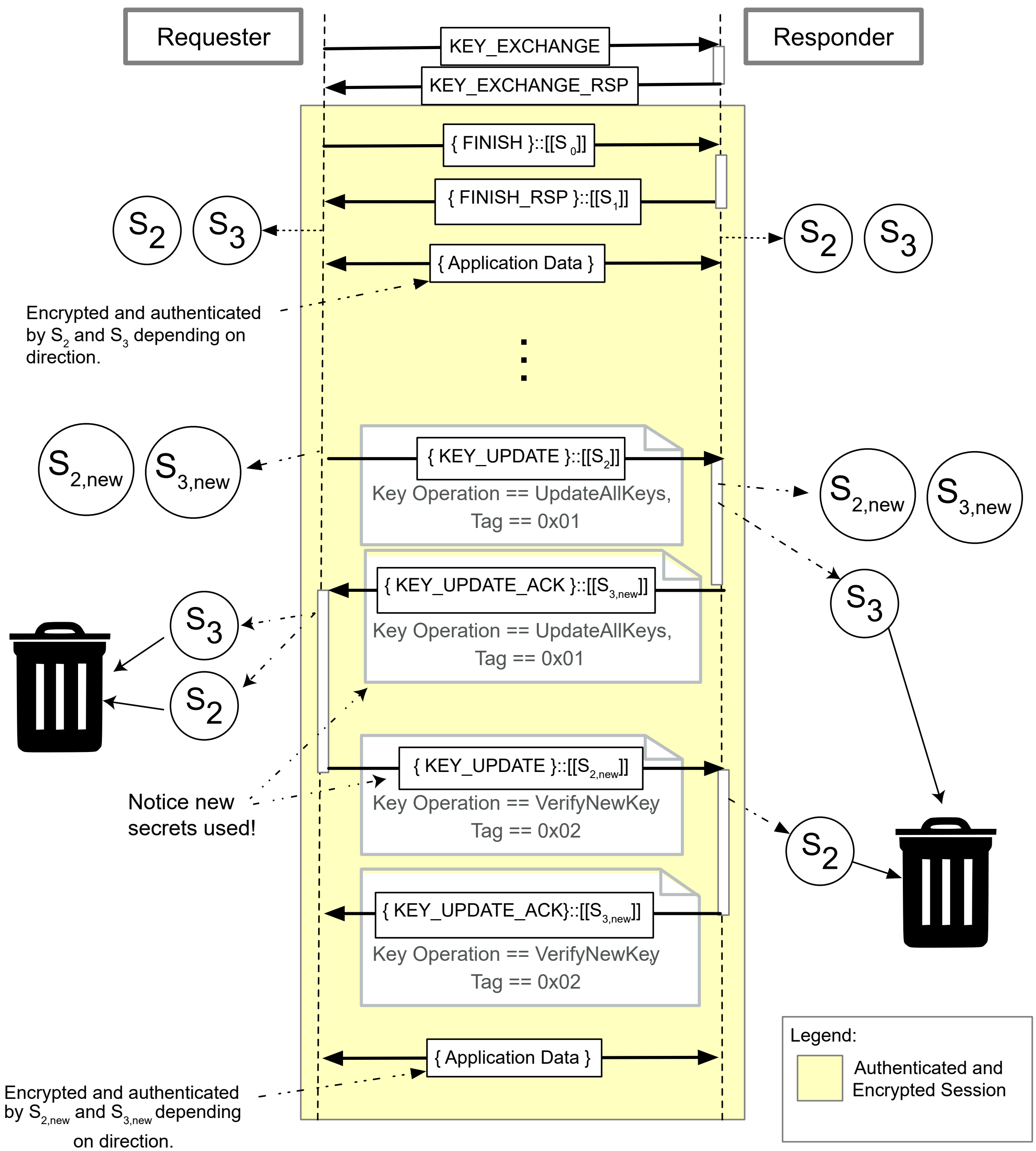
701 Figure 18 — KEY_UPDATE protocol example flow illustrates a typical key update initiated by the Requester to update its secret. In this example, the Responder and Requester are both capable of message authentication and encryption.

702 **Figure 18 — KEY_UPDATE protocol example flow**

703 

704 Figure 19 — KEY_UPDATE protocol change all keys example flow illustrates a typical key update initiated by the Requester to update all secrets. In this example, the Responder and Requester are both capable of message authentication and encryption.

705 **Figure 19 — KEY_UPDATE protocol change all keys example flow**

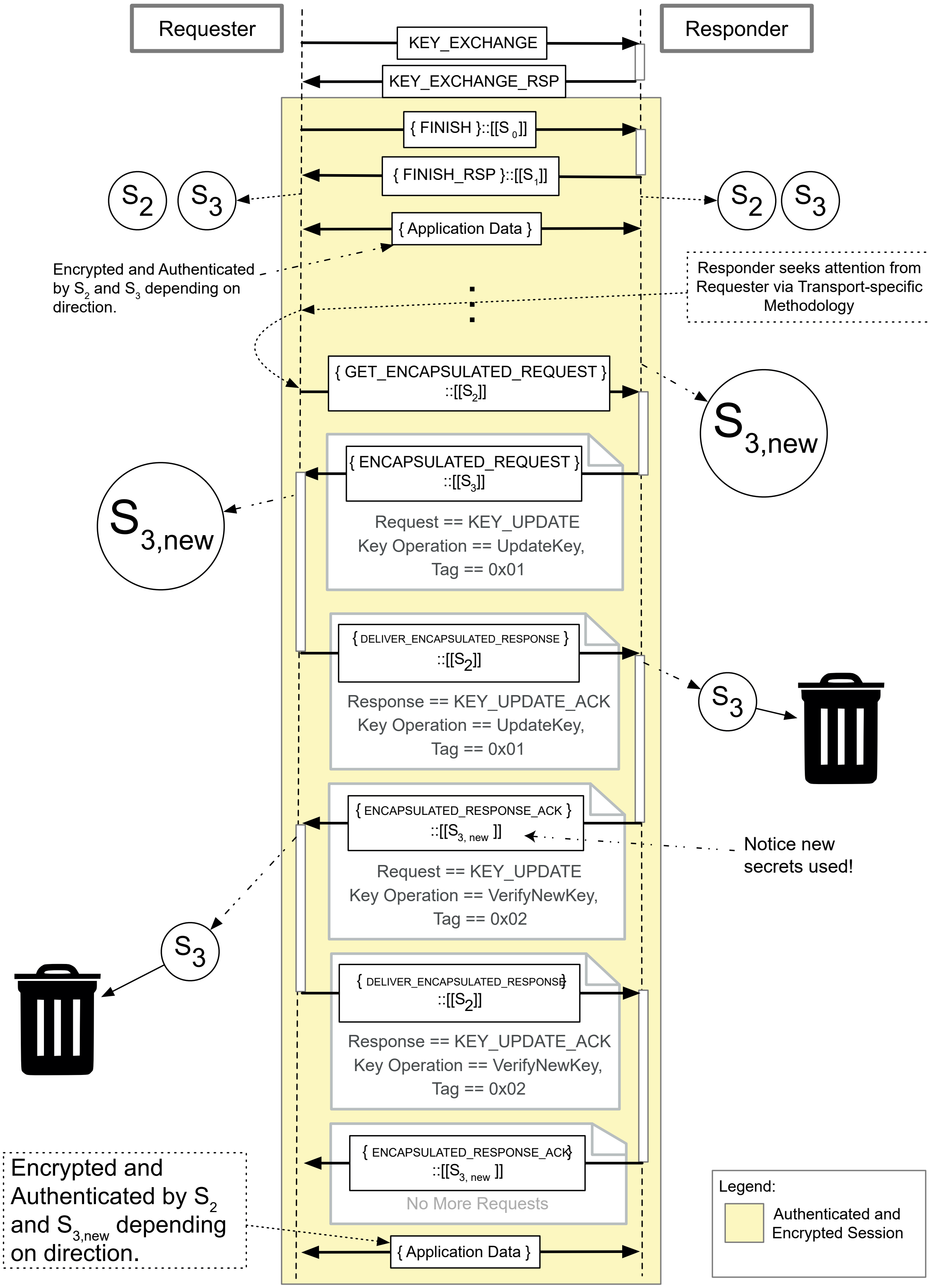
706 

707 10.22.2 KEY_UPDATE transport allowances

708 On some transports, bidirectional communication can occur asynchronously. On such transports, the transport can allow or disallow the `KEY_UPDATE` to be sent asynchronously without using the `GET_ENCAPSULATED_REQUEST` mechanism. The transport should define the actual method to use. Such a definition is outside the scope of this specification.

709 Figure 20 — KEY_UPDATE protocol example flow 2 illustrates a key update over a physical transport that has a limitation whereby only a single device (often called the "primary") is allowed to initiate all transactions on that bus. This physical transport specifies that a Responder shall alert the Requester through a side-band mechanism and to utilize the `GET_ENCAPSULATED_REQUEST` mechanism to fulfill SPDM-related requirements. Note also in this example that the Requester and Responder are both capable of encryption and message authentication.

710 **Figure 20 — KEY_UPDATE protocol example flow 2**


Requester
Responder
KEY_EXCHANGE
KEY_EXCHANGE_RSP
{ FINISH }::[[S0]]
{ FINISH_RSP }::[[S1]]
S2
S3
S2
S3
{ Application Data }
Encrypted and Authenticated by S2 and S3 depending on direction.
Responder seeks attention from Requester via Transport-specific Methodology
{ GET_ENCAPSULATED_REQUEST } ::[[S2]]
S3,new
{ ENCAPSULATED_REQUEST } ::[[S3]]
S3,new
Request == KEY_UPDATE
Key Operation == UpdateKey,
Tag == 0x01
{ DELIVER_ENCAPSULATED_RESPONSE } ::[[S2]]
S3
Response == KEY_UPDATE_ACK
Key Operation == UpdateKey,
Tag == 0x01
{ ENCAPSULATED_RESPONSE_ACK } ::[[S3, new]]
Notice new secrets used!
Request == KEY_UPDATE
Key Operation == VerifyNewKey,
Tag == 0x02
S3
{ DELIVER_ENCAPSULATED_RESPONSE } ::[[S2]]
Response == KEY_UPDATE_ACK
Key Operation == VerifyNewKey,
Tag == 0x02
{ ENCAPSULATED_RESPONSE_ACK } ::[[S3, new]]
No More Requests
Encrypted and Authenticated by S2 and S3,new depending on direction.
{ Application Data }
Legend:
Authenticated and Encrypted Session

712 10.23 GET_ENCAPSULATED_REQUEST request and ENCAPSULATED_REQUEST response messages

713 In certain use cases, such as mutual authentication, the Responder needs the ability to issue its own SPDM request messages to the Requester. Certain transports prohibit the Responder from asynchronously sending out data on that transport. Cases like these are addressed through message encapsulation, which preserves the roles of Requester and Responder as far as the transport is concerned but enables the Responder to issue its own requests to the Requester. Message encapsulation is only allowed in certain scenarios, as described in various clauses in other parts of this specification. For example, Figure 21 — Session-based mutual authentication example and Figure 22 — Optimized session-based mutual authentication example illustrate the use of this scheme.

714 A Requester issues a `GET_ENCAPSULATED_REQUEST` request message to retrieve an encapsulated SPDM request message from the Responder. The response to this message is an `ENCAPSULATED_REQUEST` that encapsulates the SPDM request message as if the Responder were acting as a Requester. Table 88 — GET_ENCAPSULATED_REQUEST request message format describes the request message format. The Responder shall use the same SPDM version the Requester used.

715 10.23.1 Encapsulated request flow

716 The encapsulated request flow starts with the Requester sending a `GET_ENCAPSULATED_REQUEST` message and ends with an `ENCAPSULATED_RESPONSE_ACK` that carries no more encapsulated requests. The `GET_ENCAPSULATED_REQUEST` shall only be issued once, with the exception of retries. This is also illustrated in Figure 21 — Session-based mutual authentication example.

717 When the Requester issues a `GET_ENCAPSULATED_REQUEST`, the encapsulated request flow shall start. Upon the successful reception of the `ENCAPSULATED_REQUEST` and when the encapsulated response is ready, the Requester shall continue the flow by issuing the `DELIVER_ENCAPSULATED_RESPONSE`. During this period, the Requester shall not issue any other message, with the exception of `GET_VERSION`, `RESPOND_IF_READY`, or `DELIVER_ENCAPSULATED_RESPONSE`. If a Responder receives a request other than `DELIVER_ENCAPSULATED_RESPONSE`, `RESPOND_IF_READY`, or `GET_VERSION`, the Responder should respond with an `ERROR` message of `ErrorCode=RequestInFlight`.

718 10.23.2 Optimized encapsulated request flow

719 The optimized encapsulated request flow is similar to the encapsulated request flow but without the need of a `GET_ENCAPSULATED_REQUEST`. This is because the encapsulated request accompanies one of the Session-Secrets-Exchange responses; thereby removing the obligation on the Requester to issue a `GET_ENCAPSULATED_REQUEST`. When the Responder includes an encapsulated request with a Session-Secrets-Exchange response, the optimized encapsulated request flow shall start. See Figure 22 — Optimized session-based mutual authentication example.

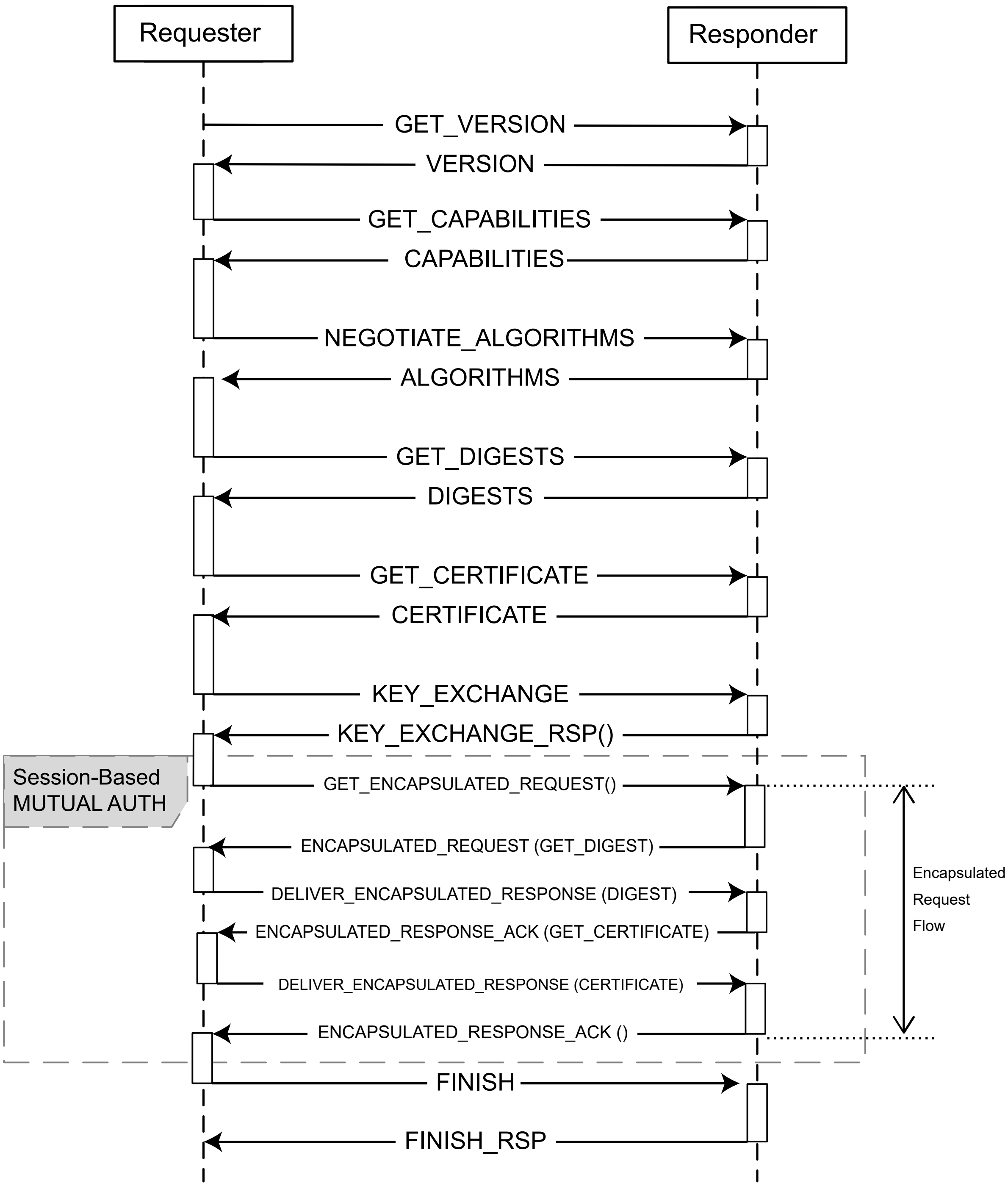
720 When the Requester successfully receives a Session-Secrets-Exchange response with an included encapsulated request, the Requester shall send a `DELIVER_ENCAPSULATED_RESPONSE` after processing the encapsulated request. The Requester shall not issue any other requests except for `DELIVER_ENCAPSULATED_RESPONSE`, `RESPOND_IF_READY`, and

`GET_VERSION`. If a Responder receives a request other than `DELIVER_ENCAPSULATED_RESPONSE`, `RESPOND_IF_READY`, or `GET_VERSION`, the Responder should respond with an `ERROR` message of `ErrorCode=RequestInFlight`.

721 Figure 21 — Session-based mutual authentication example shows an example of session-based mutual authentication:

722 **Figure 21 — Session-based mutual authentication example**

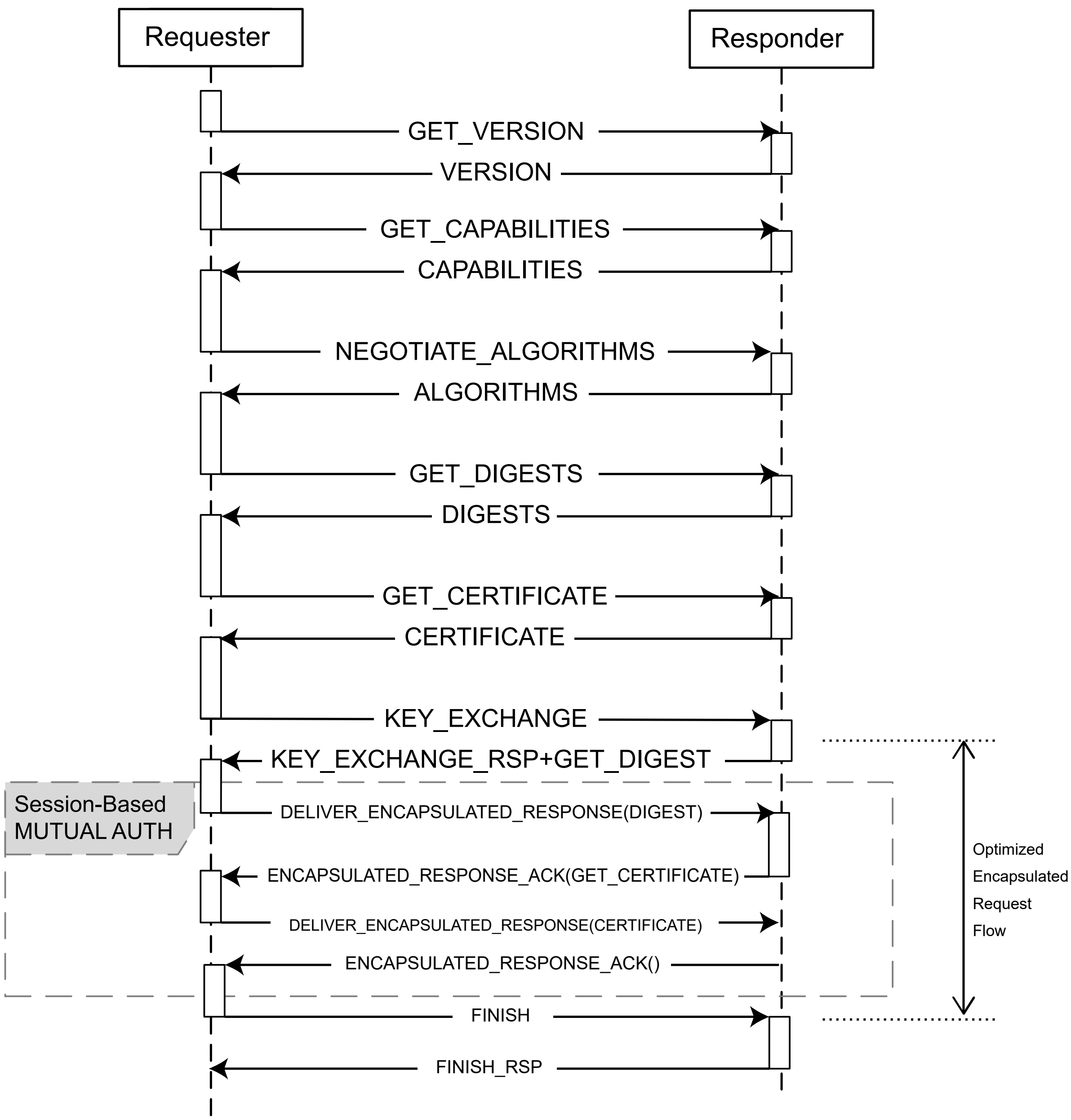
723



724 Figure 22 — Optimized session-based mutual authentication example shows an example of optimized session-based mutual authentication:

725 **Figure 22 — Optimized session-based mutual authentication example**

726



727 **Table 88 — GET_ENCAPSULATED_REQUEST request message format**

Byte offset	Field	Size (bytes)	Description
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.

Byte offset	Field	Size (bytes)	Description
1	RequestResponseCode	1	Shall be `0xEA` = `GET_ENCAPSULATED_REQUEST`. See Table 4 — SPDM request codes.
2	Param1	1	Reserved.
3	Param2	1	Reserved.

728 Table 89 — ENCAPSULATED_REQUEST response message format describes the format of this response.

729 **Table 89 — ENCAPSULATED_REQUEST response message format**

Byte offset	Field	Size (bytes)	Description
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	RequestResponseCode	1	Shall be `0x6A` = `ENCAPSULATED_REQUEST` response. See Table 5 — SPDM response codes.
2	Param1	1	Shall be the Responder-allocated Request ID. This field should be unique to help the Responder match response to request.
3	Param2	1	Reserved.
4	EncapsulatedRequest	Variable	Shall be the SPDM Request Message. The value of this field shall represent a valid SPDM request message. The length of this field is dependent on the SPDM Request message. The field shall start with the `SPDMVersion` field. The `SPDMVersion` field of the `Encapsulated Request` shall be the same as the `SPDMVersion` of the `ENCAPSULATED_REQUEST` response. Both `GET_ENCAPSULATED_REQUEST` and `DELIVER_ENCAPSULATED_RESPONSE` shall be invalid requests, and the Requester should respond with an `ERROR` message of `ErrorCode=UnexpectedRequest` if these requests are encapsulated.

730 10.23.3 Triggering `GET_ENCAPSULATED_REQUEST`

731 Once a session has been established, the Responder might wish to send a request asynchronously, such as a `KEY_UPDATE` request, but cannot due to the limitations of the physical bus or transport protocol. In such a scenario, the transport and/or physical layer is responsible for defining an alerting mechanism for the Requester. Upon receiving the alert, the Requester shall issue a `GET_ENCAPSULATED_REQUEST` to the Responder.

732 If the physical transport cannot define an alerting mechanism to the Requester, the Requester can still use the

encapsulated request flow as a polling mechanism by periodically sending the `GET_ENCAPSULATED_REQUEST` message. If the Responder receives a `GET_ENCAPSULATED_REQUEST` and has no request pending, the Responder should respond with an `ERROR` message of `ErrorCode=NoPendingRequests`.

733 ## 10.24 DELIVER_ENCAPSULATED_RESPONSE request and ENCAPSULATED_RESPONSE_ACK response messages

734 As a Requester processes an encapsulated request, it needs a mechanism to deliver back the corresponding response. That mechanism shall be the `DELIVER_ENCAPSULATED_RESPONSE` and `ENCAPSULATED_RESPONSE_ACK` messages. The `DELIVER_ENCAPSULATED_RESPONSE`, which is an SPDM request, encapsulates the response and delivers it to the Responder. The `ENCAPSULATED_RESPONSE_ACK`, which is an SPDM response, acknowledges the reception of the encapsulated response.

735 Furthermore, if there are additional requests from the Responder, the Responder shall provide the next request in the `ENCAPSULATED_RESPONSE_ACK` response message.

736 In an encapsulated request flow, the Requester shall not send any other requests after the successful reception of the first encapsulated request, with the exception of `DELIVER_ENCAPSULATED_RESPONSE`, `RESPOND_IF_READY`, or `GET_VERSION`. If a Responder receives a request other than `DELIVER_ENCAPSULATED_RESPONSE`, `RESPOND_IF_READY`, or `GET_VERSION` after the successful reception of the first `DELIVER_ENCAPSULATED_RESPONSE`, the Responder should respond with an `ERROR` message of `ErrorCode=RequestInFlight`.

737 If `Param2` of `ENCAPSULATED_RESPONSE_ACK` is set to `0x00` or `0x02`, then this shall be the final encapsulated flow message that the Responder shall issue and the encapsulated flow shall be completed.

738 The timing parameters for the response shall depend on the encapsulated request. This enables the Responder to process the response before delivering the next request. See Additional information.

739 Table 90 — DELIVER_ENCAPSULATED_RESPONSE request message format describes the request message format.

740 **Table 90 — DELIVER_ENCAPSULATED_RESPONSE request message format**

Byte offset	Field	Size (bytes)	Description
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	RequestResponseCode	1	Shall be `0xEB` = `DELIVER_ENCAPSULATED_RESPONSE` Request. See Table 4 — SPDM request codes.

Byte offset	Field	Size (bytes)	Description
2	Param1	1	Shall be the Request ID. The Requester shall use the same `Request ID` (that is, `Param1`) that was provided by the Responder in the corresponding `ENCAPSULATED_REQUEST` or `ENCAPSULATED_RESPONSE_ACK` . If the value was not provided by the Responder (for example, in the first message of an optimized encapsulated request flow), `Request ID` shall be 0.
3	Param2	1	Reserved.
4	EncapsulatedResponse	Variable	Shall be the SPDM Response Message. The value of this field shall represent a valid SPDM response message. The length of this field is dependent on the SPDM Response message. The field shall start with the `SPDMVersion` field. The `SPDMVersion` field of the `Encapsulated Response` shall be the same as the `SPDMVersion` of the `DELIVER_ENCAPSULATED_RESPONSE` request. Both `ENCAPSULATED_REQUEST` and `ENCAPSULATED_RESPONSE_ACK` shall be invalid responses, and the Responder should respond with an `ERROR` message of `ErrorCode=InvalidResponseCode` if these responses are encapsulated.

741 Table 91 — ENCAPSULATED_RESPONSE_ACK response message format describes the response message format.

742 **Table 91 — ENCAPSULATED_RESPONSE_ACK response message format**

Byte offset	Field	Size (bytes)	Description
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	RequestResponseCode	1	Shall be `0x6B` = `ENCAPSULATED_RESPONSE_ACK` . See Table 5 — SPDM response codes.
2	Param1	1	Shall be the Request ID. If `EncapsulatedRequest` is present and if `Param2` = `0x01` , this field should contain a unique non-zero number to help the Responder match response to request. Otherwise, this field shall be `0x00` .

Byte offset	Field	Size (bytes)	Description
3	Param2	1	Shall indicate the payload Type. If set to `0x00`, no request message is encapsulated and the `EncapsulatedRequest` field is absent. If set to `0x01`, the `EncapsulatedRequest` field follows. If set to `0x02`, a 1-byte `EncapsulatedRequest` field follows containing the `SlotID` of the Requester's certificate chain used for mutual authentication. The value in this field shall be between 0 and 7 inclusive. All other values reserved.
4	AckRequestID	1	Shall be the same as `Param1` of the `DELIVER_ENCAPSULATED_RESPONSE` request message. The purpose of this field is to help the Requester distinguish between new requests and retries.
5	Reserved	3	Reserved.
8	EncapsulatedRequest	Variable	If `Param2` = `0x01`, the value of this field shall represent a valid SPDM request message. The length of this field is dependent on the SPDM Request message. The field shall start with the `SPDMVersion` field. The `SPDMVersion` field of the `EncapsulatedRequest` shall be the same as the `SPDMVersion` of the `ENCAPSULATED_REQUEST` response. Both `GET_ENCAPSULATED_REQUEST` and `DELIVER_ENCAPSULATED_RESPONSE` shall be invalid requests, and the Requester shall respond with an `ERROR` message of `ErrorCode=UnexpectedRequest` if these requests are encapsulated. If `Param2` = `0x02`, the value of this field shall contain the `SlotID` corresponding to the certificate chain the Requester shall use for mutual authentication. The field size shall be 1 byte. If `Param2` = `0x00`, this field shall be absent.

743 **10.24.1 Additional information**

744 Using unique `Request ID`s is highly recommended to aid the Responder in differentiating between retries and new `DELIVER_ENCAPSULATED_RESPONSE` messages. For example, if the Responder sent an `ENCAPSULATED_RESPONSE_ACK` message with a new encapsulated request and the message failed in transmission over the wire, the Requester would send a retry but that retry would still contain the response to the previous encapsulated request. Without a

different `Request ID`, the Responder might mistake the retried `DELIVER_ENCAPSULATED_RESPONSE` for a new request. This mistake might cause further mistakes to occur.

745 The response timing for `ENCAPSULATED_RESPONSE_ACK` shall have the same timing constraints as the encapsulated request. For example, if the encapsulated request is `CHALLENGE_AUTH`, the Responder, too, would adhere to `CT` timing rules when it has a subsequent request. If necessary, the Requester can return an `ERROR` message of `ErrorCode=ResponseNotReady`.

746 The `DELIVER_ENCAPSULATED_RESPONSE` and `ENCAPSULATED_RESPONSE_ACK` messages shall only be allowed to encapsulate certain requests in certain scenarios. For details about these constraints, see the Session, Basic mutual authentication, and KEY_UPDATE request and KEY_UPDATE_ACK response messages clauses.

747 10.24.2 Allowance for encapsulated requests

748 Only certain requests can be encapsulated in any encapsulated request flow. Their corresponding responses, including `ERROR`, can also be encapsulated. Additionally, these requests are only allowed in certain flows as described in various parts of this specification. This consolidated list shall be the requests that are allowed to be encapsulated:

- `CHALLENGE`
- `GET_CERTIFICATE`
- `GET_DIGESTS`
- `KEY_UPDATE`
- `END_SESSION`
- `SUBSCRIBE_EVENT_TYPES`
- `SEND_EVENT`
- `GET_SUPPORTED_EVENT_TYPES`
- `GET_ENDPOINT_INFO`

749 If a request is not in this list, the request and its corresponding response shall be prohibited from being encapsulated.

750 10.24.3 Certain error handling in encapsulated flows

751 These clauses describe special error scenarios and their handling requirements.

752 10.24.3.1 Response not ready

753 In an encapsulated request flow, a Responder can issue an encapsulated request that can take up to `CT` time to fulfill. When the Requester delivers an `ERROR` message of `ErrorCode=ResponseNotReady`, the Responder shall not encapsulate another request by setting `Param2` in `ENCAPSULATED_RESPONSE_ACK` to a value of zero. This effectively and naturally terminates the encapsulated request flow.

754 The Responder should wait the amount of time indicated in the `ERROR` message for the particular error code.

755 When the timeout is near expiration, the Responder should perform the following:

1. Trigger its transport-defined alert mechanism to initiate the Encapsulated request flow.
2. When the Requester issues a `GET_ENCAPSULATED_REQUEST`, the Responder should encapsulate the `RESPOND_IF_READY` request populated with the information from the previous `ERROR` with `ResponseNotReady` message.
   - If the Responder does not do this, the Requester can drop the original response.

756 **10.24.3.2 Timeouts**

757 If the Responder is not receiving a response to its encapsulated request, the Responder can trigger its transport-defined alert mechanism. When this occurs, if the Requester is in the middle of an existing encapsulated request flow with the same Responder, then the existing flow shall terminate and the Requester shall restart the encapsulated request flow.

758 Both Responder and Requester should comply with the timing requirements prescribed in Timing requirements.

759 10.25 END_SESSION request and END_SESSION_ACK response messages

760 This request shall terminate a session. See the Session termination phase clause.

761 Table 92 — END_SESSION request message format and Table 93 — End session request attributes describe this format.

762 **Table 92 — END_SESSION request message format**

Byte offset	Field	Size (bytes)	Description
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	RequestResponseCode	1	Shall be `0xEC` = `END_SESSION`. See Table 4 — SPDM request codes.
2	Param1	1	See Table 93 — End session request attributes.
3	Param2	1	Reserved.

763 **Table 93 — End session request attributes**

Bit offset	Value	Field	Description
0	0	Negotiated State Clearing Indicator	If the Responder does not support Negotiated State caching (`CACHE_CAP=0`), this field shall be ignored. For an `END_SESSION` request sent from the Requester to the Responder, the Responder shall preserve the cached Negotiated State. For an `END_SESSION` request sent from the Responder to the Requester, the value of 0 notifies the Requester that the Responder has preserved the cached Negotiated State.
0	1	Negotiated State Clearing Indicator	If the Responder does not support Negotiated State caching (`CACHE_CAP=0`), this field shall be ignored. For an `END_SESSION` request sent from the Requester to the Responder, the Responder shall also clear the cached Negotiated State as part of session termination. If there is no cached Negotiated State to be cleared due to a previous `END_SESSION` request message with this field set to 1, this field shall be ignored. For an `END_SESSION` request sent from the Responder to the Requester, the value of 1 notifies the Requester that the Responder has cleared the cached Negotiated State, and the Requester shall send `GET_VERSION` , `GET_CAPABILITIES` , and `NEGOTIATE_ALGORITHMS` at the next handshake.
[7:1]	Reserved	Reserved	Reserved.

764 Table 94 — END_SESSION_ACK response message format describes the response message.

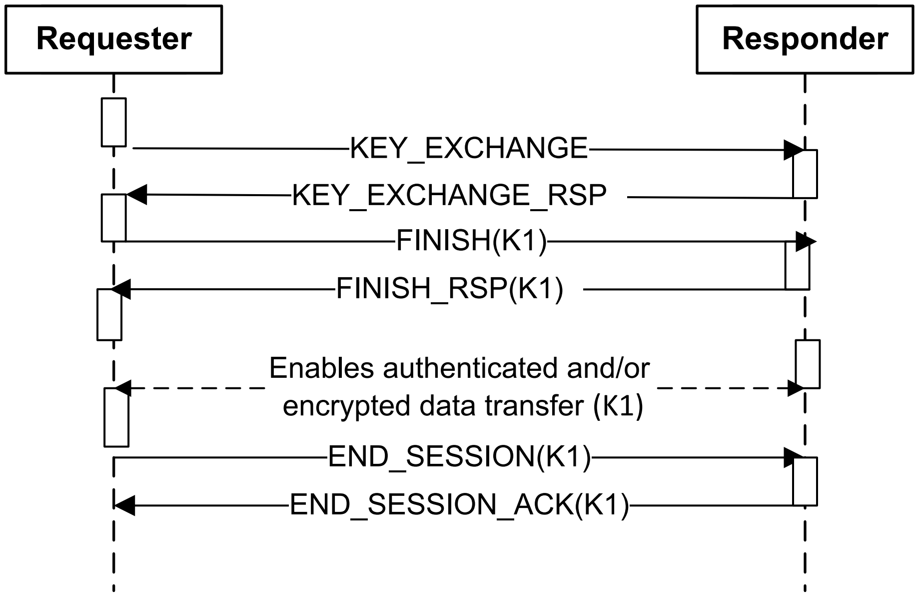
765 **Table 94 — END_SESSION_ACK response message format**

Byte offset	Field	Size (bytes)	Description
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	RequestResponseCode	1	Shall be `0x6C` = `END_SESSION_ACK` . See Table 5 — SPDM response codes.
2	Param1	1	Reserved.
3	Param2	1	Reserved.

766 Figure 23 — END_SESSION protocol flow shows the `END_SESSION` protocol flow:

767 **Figure 23 — END_SESSION protocol flow**

768



769 10.25.1 END_SESSION additional information

770 The transport layer might allow the `END_SESSION` request to be sent from the Responder to the Requester. This is recommended for transports capable of asynchronous bidirectional communication.

771 10.26 Certificate provisioning

772 These clauses describe the request and response messages used for provisioning a device with certificate chains. Provisioning of Slot 0 should only be done in a trusted environment (such as a secure manufacturing environment).

773 10.26.1 GET_CSR request and CSR response messages

774 The `GET_CSR` request message shall retrieve a Certificate Signing Request (CSR) from the Responder.

775 A Responder shall only process a `GET_CSR` request if it already possesses an appropriate asymmetric key pair for the signature suite (that is, the algorithms and associated parameters) required by the request. If more than one signature suite are supported, selection of the appropriate signature suite (and, thus, the key pair) shall be determined via the most recent `ALGORITHMS` response. Upon receiving a `GET_CSR` request, a Responder shall generate and sign a CSR for the corresponding public key. The CSR shall be populated with a combination of attributes provided by the Requester via the `RequesterInfo` field and other attributes contributed by the Responder itself. The `RequesterInfo` format shall comply with the PKCS #10 specification in RFC 2986, specifically the `CertificationRequestInfo` format. Vendor-defined extensions shall be encoded using the `Attributes` type. The

Responder may alter the value of requested `CertificationRequestInfo` fields in `RequesterInfo` when generating `CSRdata`. The Responder shall return an `ERROR` message with error code `InvalidRequest` if it cannot support a field included in the `RequesterInfo`, or if the value of a requested field is not supported and the Responder cannot alter the value of the field. If the Responder receives a new `GET_CSR` request (`CSRTrackingTag` = `0`) while another `GET_CSR` request is outstanding and if `Overwrite` is not specified (that is, Bit 7 of `Param2` is set to `0b`), the Responder can either overwrite the existing request and process the new `GET_CSR` request or respond with an `ERROR` message of `ErrorCode=Busy`. If the Responder receives a `GET_CSR` request while another `GET_CSR` request is outstanding and if `Overwrite` is specified (that is, Bit 7 of `Param2` is set to `1b`), the Responder shall overwrite the existing request and process the new `GET_CSR` request.

776 If the device requires a reset to complete the `GET_CSR` request, the device shall respond with an `ERROR` message of `ErrorCode=ResetRequired` with `Bit[2:0]` of the `Error Data` field set to a Responder-assigned `CSRTrackingTag` in the range of `1` to `7`, inclusive. `CSRTrackingTag`s are allocated and managed by the Responder. If a Requester is sending a new `GET_CSR` request, then the `CSRTrackingTag` field shall be set to `0`. If the Responder requires a reset to process a `GET_CSR` request, but does not have any available `CSRTrackingTag`s, it shall respond with an `ERROR` message of `ErrorCode=Busy`. After the Responder has processed the reset, the Requester sends a `GET_CSR` request with `Bit[5:3]` in `Param2` set to the `CSRTrackingTag` that the Responder provided in the corresponding `ERROR` response, which signals to the Responder to send the `CSR` response associated with the previous request. After a Requester has retrieved a `CSR` response from a previous `GET_CSR` request, the Responder can discard any associated CSR data and reuse the `CSRTrackingTag`. If the Requester sends a `GET_CSR` request with a non-zero `CSRTrackingTag` that the Responder did not generate, the Responder shall either respond with an `ERROR` message of `ErrorCode=UnexpectedRequest` or drop the request.

777 The attributes of the resulting CSR and their values shall comply with the clauses presented in SPDM certificate requirements and recommendations. If the `GET_CSR` request conforms to the `DeviceCert` model, the resulting CSR shall be for a Device Certificate. If the `GET_CSR` request conforms to the `AliasCert` model, the resulting CSR shall be for a Device Certificate CA. If the `GET_CSR` request conforms to the `GenericCert` model, the resulting CSR shall be for a Generic Leaf Certificate. See Identity provisioning for more details.

778 Table 95 — GET_CSR request message format shows the `GET_CSR` request message format.

779 Table 97 — CSR response message format shows the `CSR` response message format.

780 Fields from `CSRdata` contained in a successful `CSR` response are assembled into a certificate and should be signed by an appropriate Certificate Authority. The details of the Public Key Infrastructure used to verify and sign the CSR and make the final certificate available for provisioning are outside the scope of this specification.

781 **Table 95 — GET_CSR request message format**

Byte offset	Field	Size (bytes)	Description
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	RequestResponseCode	1	Shall be `0xED` = `GET_CSR`. See Table 4 — SPDM request codes.

Byte offset	Field	Size (bytes)	Description
2	Param1	1	KeyPairID. The value of this field shall be the key pair ID identifying the desired asymmetric key pair to use in generating the CSR. If `MULTI_KEY_CONN_RSP` is false, the value shall be zero; otherwise, the value shall be non-zero.
3	Param2	1	Request Attributes. Shall be the format as Get CSR request attributes defines.
4	RequesterInfoLength	2	Shall be the length of the `RequesterInfo` field in bytes provided by the Requester. This field can be `0` .
6	OpaqueDataLength	2	Shall be the size of the `OpaqueData` field that follows in bytes. The value should not be greater than 1024 bytes. Shall be `0` if no `OpaqueData` is provided.
8	RequesterInfo	`RequesterInfoLength`	Shall be the optional information provided by the Requester. This field shall be DER-encoded.
8 + `RequesterInfoLength`	OpaqueData	`OpaqueDataLength`	The Requester can include vendor-specific information for the Responder to generate the CSR. This field is optional. If present, this field shall conform to the selected opaque data format in `OtherParamsSelection` .

782 **Table 96 — Get CSR request attributes**

Bit offset	Field	Description
[2:0]	CSRCertModel	This field indicates the desired certificate model of the CSR. The value and format of this field shall be the same as `CertModel` in Certificate info. If `MULTI_KEY_CONN_RSP` is true the value shall not be zero.
[5:3]	CSRTrackingTag	If the Requester is requesting a previously requested `GET_CSR` after a reset has completed, this field shall contain the `CSRTrackingTag` of the associated `GET_CSR` request.
6	Reserved	Reserved.
7	Overwrite	If set, the Responder shall stop processing any existing `GET_CSR` request and overwrite it with this request, and the Responder shall discard all previously generated `CSRTrackingTag` s.

783 The `CSRCertModel` field in GET CSR request attributes helps the Responder determine the content of the CSR. For example, if the `CSRCertModel` indicates a device certificate model, the Responder may add additional OIDs such as those OIDs defined in this specification. If the `CSRCertModel` indicates an alias certificate model, the Responder sets the `CA` constraint to `TRUE` in the CSR.

784 **Table 97 — CSR response message format**

Byte offset	Field	Size (bytes)	Description
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	RequestResponseCode	1	Shall be `0x6D` = `CSR`. See Table 5 — SPDM response codes.
2	Param1	1	Reserved.
3	Param2	1	Reserved.
4	CSRLength	2	Shall be the length of the `CSRdata` in bytes.
6	Reserved	2	Reserved.
8	CSRdata	`CSRLength`	Shall be the requested contents of the CSR. This field shall be DER-encoded.

785 The `CSRdata` format shall comply with the PKCS #10 specification in RFC 2986, specifically the `CertificationRequest` format. When the Responder supports multiple asymmetric keys (`MULTI_KEY_CONN_RSP` is true) in the SPDM connection, the `SubjectPublicKeyInfo` as defined in RFC 5280 shall contain values consistent with the requested asymmetric key pair (`KeyPairID`) in the corresponding request.

786 10.26.2 SET_CERTIFICATE request and SET_CERTIFICATE_RSP response messages

787 For Slot 0 provisioning, the Requester should issue `SET_CERTIFICATE` only in a trusted environment (such as a secure manufacturing environment). For slots 1-7, if the provisioning happens in a trusted environment, the Requester should issue `SET_CERTIFICATE` inside a secure session. If the provisioning for slots 1-7 is done outside of a trusted environment, the Requester shall issue `SET_CERTIFICATE` inside a secure session. Mutual authentication and/or other means for checking the authorization of the Requester that issues the `SET_CERTIFICATE` request should be performed. Requester authorization is outside the scope of this specification. The device might require a reset to complete the `SET_CERTIFICATE` request, potentially so that the device can generate `AliasCert` certificates using lower firmware layers. If the device requires a reset to complete the `SET_CERTIFICATE` request, then the device shall respond with an `ERROR` message of `ErrorCode=ResetRequired` . When `ResetRequired` is pending and the device receives a new `SET_CERTIFICATE` request for the same slot number, the device shall overwrite the existing `CertChain` and process the new `SET_CERTIFICATE` request. If the device temporarily cannot write to a slot, including in the case when it receives overlapping `SET_CERTIFICATE` requests from different Requesters, then the device shall respond with an `ERROR` message of `ErrorCode=Busy` .

788 If `Bit 7` of `SET_CERTIFICATE` . `Param1` is set to `1` , the Responder shall erase the certificate chain present in the slot identified by bits [3:0] of `SET_CERTIFICATE` . `Param1` and report it as unpopulated until it is re-provisioned. The `SET_CERTIFICATE` `Erase` operation does not erase the key associated with the specified slot. If the operation completes successfully, the Responder shall respond with a `SET_CERTIFICATE_RSP` response message with bits [3:0] of `Param1` identifying the `SlotID` of the slot that was erased. If the operation failed, the Responder shall respond with an `ERROR` message with `ErrorCode=OperationFailed` . If the specified slot is unpopulated when the `SET_CERTIFICATE` `Erase` operation is issued, the Responder shall respond with an `ERROR` message with `ErrorCode=InvalidRequest` .

789 When a reset is required for a pending previous `SET_CERTIFICATE` request and the device receives a `GET_CERTIFICATE` request for a pending slot or a `GET_DIGESTS` request, the device shall respond with an `ErrorCode=ResetRequired` response.

790 Table 98 — SET_CERTIFICATE request message format shows the `SET_CERTIFICATE` request message format.

791 Table 100 — Successful SET_CERTIFICATE_RSP response message format shows the `SET_CERTIFICATE_RSP` response message format.

792 **Table 98 — SET_CERTIFICATE request message format**

Byte offset	Field	Size (bytes)	Description
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	RequestResponseCode	1	Shall be `0xEE` = `SET_CERTIFICATE`. See Table 4 — SPDM request codes.
2	Param1	1	Request attributes. Shall be the format that the set certificate request attributes table defines.
3	Param2	1	KeyPairID. The value of this field shall be the unique key pair number identifying the desired asymmetric key pair to associate with `SlotID`. If support for multiple asymmetric keys (`MULTI_KEY_CONN_RSP`) is false, the value of this field shall be zero.
4	CertChain	Variable	Shall be the contents of the target certificate chain, as specified in Certificates and certificate chains, with the additional requirement that it include the root certificate. If the Responder uses the `AliasCert` model (`ALIAS_CERT_CAP=1b` in its `CAPABILITIES` response) and `SetCertModel` is set to `AliasCert`, this field shall contain a partial certificate chain from the root CA to the Device Certificate CA. If the `Request attributes`. `Erase` bit is set, this field shall be absent. If the certificate chain exceeds 64 KB and `LARGE_CERT_CAP=0b` in the `CAPABILITIES` response, the Responder shall send an Error message with `ErrorCode=CertChainTooLarge`.

793 **Table 99 — Set certificate request attributes**

Bit offset	Field	Description
[3:0]	SlotID	The certificate slot where the new certificate is written. The value in this field shall be between 0 and 7 inclusive.

Bit offset	Field	Description
[6:4]	SetCertModel	This field indicates the certificate model of the certificate chain. The value and format of this field shall be the same as `CertModel` in Certificate info. If the certificate chain was formed with information from a `CSR` response, the value in this field shall match the value in the `CSRCertModel` field from the corresponding `GET_CSR` request. If `MULTI_KEY_CONN_RSP` is true and `Erase` is not set, the value shall not be zero.
7	Erase	If set, the certificate chain in the certificate slot identified by bits [3:0] shall be deleted. Additionally, if this bit is set, the `CertChain` field shall be absent and the value of `SetCertModel` shall be zero.

794 The Responder should verify that contents of the certificate chain meet the requirements in this specification for the requested certificate model and key pair. If it does not, the Responder shall retain the current certificate in the requested `SlotID`, if present. If an `Erase` operation occurs on a `SlotID` that does not contain a certificate or the request contains invalid parameters, the Responder shall respond with an `ERROR` message or silently discard the request.

795 **Table 100 — Successful SET_CERTIFICATE_RSP response message format**

Byte offset	Field	Size (bytes)	Description
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	RequestResponseCode	1	Shall be `0x6E` = `SET_CERTIFICATE_RSP`. See Table 5 — SPDM response codes.
2	Param1	1	Bit [7:4]. Reserved. Bit [3:0]. Shall be the `SlotID` where the new certificate is written. The value in this field shall be the same as `SlotID` in corresponding `SET_CERTIFICATE` Request. If the `Erase` bit is set in the `Request attributes` field, this field shall contain the `SlotID` of the slot that was erased. The value in this field shall be between 0 and 7 inclusive.
3	Param2	1	Reserved.

796 10.27 Large SPDM message transfer mechanism

797 A large SPDM message is an SPDM message whose size is either greater than the `DataTransferSize` of the receiving SPDM endpoint or greater than the transmit buffer size of the sending SPDM endpoint. These clauses provide a transport-agnostic mechanism to transfer large SPDM messages. This mechanism will be used only if the size of an SPDM message exceeds either the `DataTransferSize` of the receiving SPDM endpoint or the transmit buffer size of the sending SPDM endpoint. Additionally, the transport may provide an alternative method to transfer

large SPDM messages. For SPDM messages that are less than or equal to both the `DataTransferSize` of the receiving SPDM endpoint and the transmit buffer size of the sending SPDM endpoint, the sending SPDM endpoint shall not utilize this transfer mechanism.

798 This transfer mechanism divides a large SPDM message into smaller fragments called chunks. The chunks shall be numbered and shall be transferred in sequence. The chunks and their sequence of transfer are described thus:

- The first chunk shall be assigned a numeric value of 0, the second chunk shall be assigned a numeric value of 1, the third chunk shall be assigned a numeric value of 2, and this pattern shall continue up to and including the last chunk. Each of these numeric values is called a chunk sequence number.
- The first chunk shall contain the first set of bytes of the large SPDM message, the second chunk shall contain the second set of bytes, the third chunk shall contain the third set of bytes, and this pattern shall continue up to and including the last chunk.
- All chunks shall represent all bytes of the large SPDM message without altering the message in any way.
- The sequence of transfer shall start with chunk sequence number 0 and shall continue with sequentially increasing chunk sequence numbers up to and including the last chunk.
- The chunked transfer shall not be interrupted by any commands that are not part of the chunk transfer sequence, with the exception of `GET_VERSION`. The Responder shall return the error `ErrorCode=UnexpectedRequest` if an unexpected command is received during the chunked transfer. If `CHUNK_GET` is invalid or corrupted, the Requester may receive corresponding error codes (`ErrorCode=InvalidRequest`, `ErrorCode=VersionMismatch`, etc.). These error codes shall not interrupt the chunk transfer sequence, with exception of the error code `ErrorCode=DecryptError`.
- `CHUNK_SEND`, `CHUNK_GET`, and their corresponding Responses shall be used to transfer these chunks.

799 The `ChunkSeqNo` fields indicate the chunk sequence number for a given chunk.

800 The requests and responses, which these clauses define, handle the transfer of each chunk.

801 10.27.1 CHUNK_SEND request and CHUNK_SEND_ACK response message

802 The `CHUNK_SEND` request and the `CHUNK_SEND_ACK` response shall be used to send a request to an SPDM endpoint when the size of the request is greater than either the `DataTransferSize` of the receiving SPDM endpoint or the transmit buffer size of the sending SPDM endpoint.

803 Table 101 — CHUNK_SEND request format describes the format for the request.

804 Table 102 — Chunk sender attributes describes the chunk sender attributes.

805 **Table 101 — CHUNK_SEND request format table**

Byte offset	Field	Size (bytes)	Description
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	RequestResponseCode	1	Shall be `0x85` = `CHUNK_SEND` request. See Table 4 — SPDM request codes.

Byte offset	Field	Size (bytes)	Description
2	Param1	1	Shall be the Request Attributes. See Table 102 — Chunk sender attributes.
3	Param2	1	Shall be the handle. This field should uniquely identify the transfer of a large SPDM message. The value of this field shall be the same for all chunks of the same large SPDM message. The value of this field should either sequentially increase or sequentially decrease with each large SPDM message and with the expectation that it will wrap around after reaching the maximum or minimum value, respectively, of this field.
4	ChunkSeqNo	4	Shall identify the chunk sequence number associated with `SPDMchunk` .
8	ChunkSize	4	Shall indicate the size of `SPDMchunk` . See Additional chunk transfer requirements.
12	LargeMessageSize	`L0` = 0 or 4	Shall indicate the size of the large SPDM message being transferred. This field shall only be present when `ChunkSeqNo` is zero and shall have a non-zero value. The value of this field shall be greater than the `DataTransferSize` of the receiving SPDM endpoint.
12 + `L0`	SPDMchunk	Variable	Shall contain the chunk of the large SPDM request message associated with `ChunkSeqNo` .

806 **Table 102 — Chunk sender attributes**

Bit offset	Field	Description
0	LastChunk	If set, the chunk indicated by `ChunkSeqNo` shall represent the last chunk of the large SPDM message.
[7:1]	Reserved	Reserved.

807 Table 103 — CHUNK_SEND_ACK response message format describes the format for the response.

808 **Table 103 — CHUNK_SEND_ACK response message format**

Byte offset	Field	Size (bytes)	Description
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	RequestResponseCode	1	`0x05` = `CHUNK_SEND_ACK` response. See Table 5 — SPDM response codes.

Byte offset	Field	Size (bytes)	Description
2	Param1	1	Shall be the Response attributes. See Table 104 — Chunk receiver attributes.
3	Param2	1	Shall contain the handle from the corresponding `CHUNK_SEND` request. This field should uniquely identify the transfer of a large SPDM message. The value of this field shall be the same for all chunks of the same SPDM message.
4	ChunkSeqNo	4	Shall be the same as `ChunkSeqNo` in the corresponding request.
6	ResponseToLargeRequest	Variable	Shall be present on the last chunk (that is, when `LastChunk` is set), or when the `EarlyErrorDetected` bit in `Param1` is set. This field shall contain the response to the large SPDM request message. When the `EarlyErrorDetected` bit in `Param1` is set, this field shall contain an `ERROR` message.

809 Table 104 — Chunk receiver attributes describes the chunk receiver attributes:

810 **Table 104 — Chunk receiver attributes**

Bit offset	Field	Description
0	EarlyErrorDetected	If set, the receiver of a large SPDM request message detected an error in the Request before the last chunk was received. If set, the sender of the large SPDM request message shall terminate the transfer of any remaining chunks. After addressing the issue, the sender of the failed large SPDM request message can transfer the fixed large SPDM request message as a new transfer.
[7:1]	Reserved	Reserved.

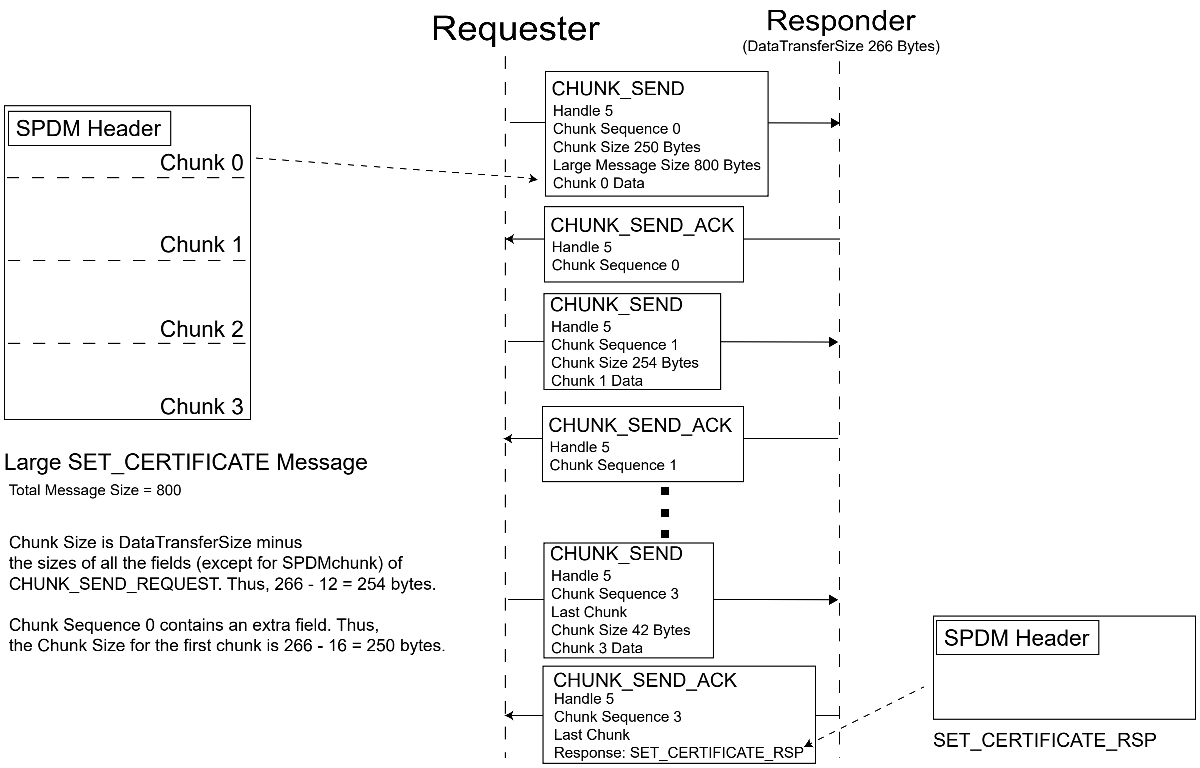
811 Upon reception of the last chunk, the receiving SPDM endpoint shall respond with the response corresponding to the large SPDM request message in `ResponseToLargeRequest`. If placing the response in `ResponseToLargeRequest` causes the size of the `CHUNK_SEND_ACK` to exceed the `DataTransferSize`, the receiving end point shall, instead, respond to `CHUNK_SEND` with an `ERROR` message of `ErrorCode=LargeResponse`. An `ERROR` message of `ErrorCode=LargeResponse` shall not be allowed in `ResponseToLargeRequest`. `ERROR` messages with other error codes can be placed in `ResponseToLargeRequest` to distinguish between an `ERROR` message to the `CHUNK_SEND` request and an `ERROR` message that is a response to the large SPDM request message.

812 In the case where the size of the `CHUNK_SEND_ACK` message is greater than `DataTransferSize` but the size of `ResponseToLargeRequest` is less than `DataTransferSize` the Responder will chunk a message whose size is less than `DataTransferSize`.

813 Figure 24 — Large SET_CERTIFICATE example illustrates the sending of a large SPDM request message to a Responder.

814 **Figure 24 — Large SET_CERTIFICATE example**

815



816 10.27.2 CHUNK_GET request and CHUNK_RESPONSE response message

817 `CHUNK_GET` request and `CHUNK_RESPONSE` response shall be used to retrieve a Large SPDM Response from an SPDM endpoint when the size of the Response is greater than the `DataTransferSize` of the SPDM endpoint receiving the Response or the transmit buffer size of the SPDM endpoint sending the Response.

818 When responding to a Request of any size, if the corresponding response will be a Large SPDM Response, the responding SPDM endpoint shall respond with an `ERROR` message of `ErrorCode=LargeResponse`. This `ERROR` message contains a handle to uniquely identify the given Large SPDM Response. The handle shall be used for all `CHUNK_GET` Requests retrieving the same large SPDM message. The value of the handle is indicated in the `Handle` field of this `ERROR` message.

819 Table 105 — CHUNK_GET request format describes the format for the request.

820 **Table 105 — CHUNK_GET request format**

Byte offset	Field	Size (bytes)	Description
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.

Byte offset	Field	Size (bytes)	Description
1	RequestResponseCode	1	Shall be `0x86` = `CHUNK_GET` request. See Table 4 — SPDM request codes.
2	Param1	1	Reserved.
3	Param2	1	Shall contain a handle. This field shall be the same value as given in the `Handle` field of the `ERROR` message of `ErrorCode=LargeResponse`.
4	ChunkSeqNo	4	Shall indicate the desired chunk sequence number of the Large SPDM Response to retrieve.

821 Table 106 — CHUNK_RESPONSE response format describes the format for the response.

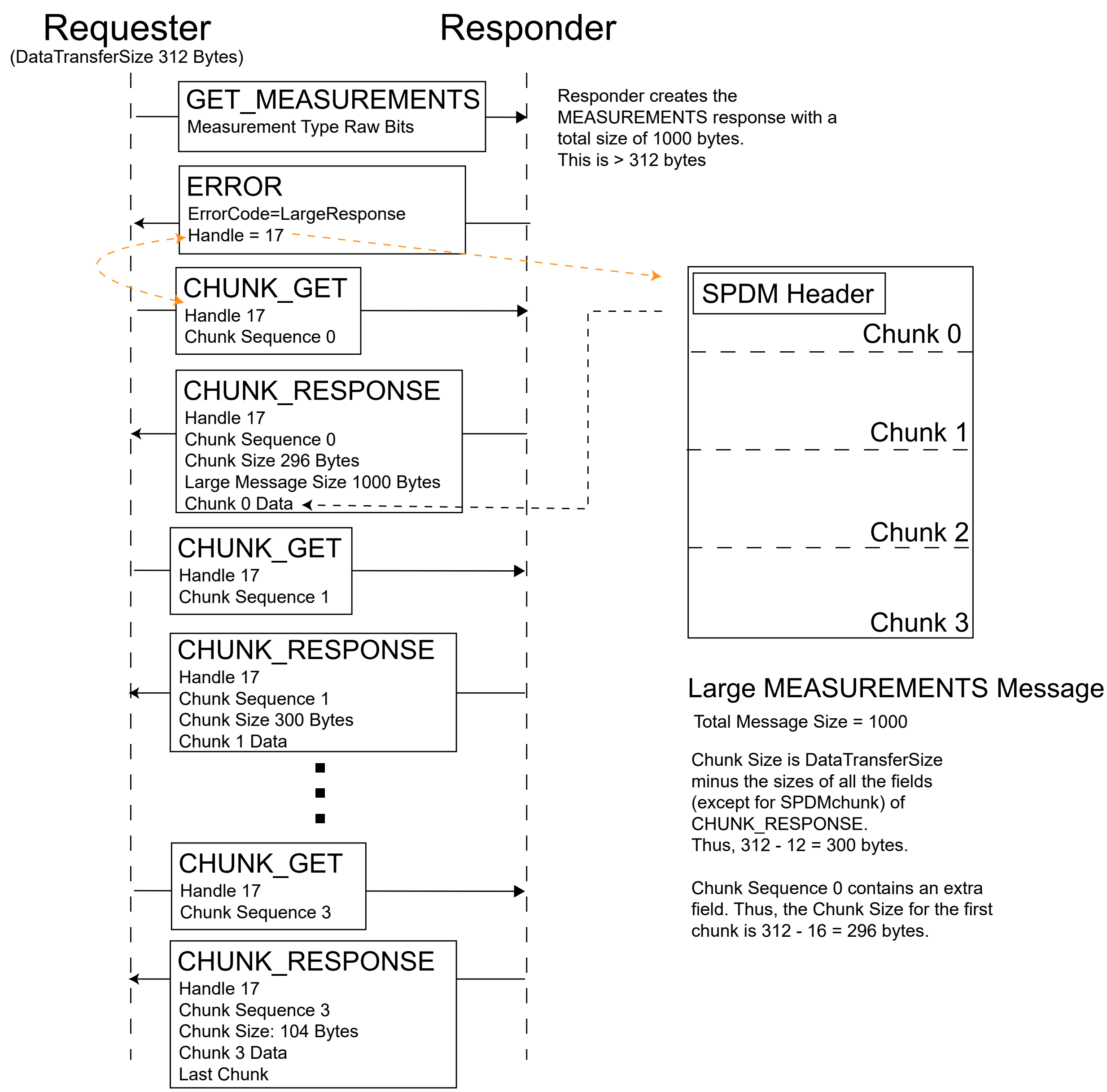
822 **Table 106 — CHUNK_RESPONSE response format**

Byte offset	Field	Size (bytes)	Description
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	RequestResponseCode	1	Shall be `0x06` = `CHUNK_RESPONSE` response. See Table 5 — SPDM response codes.
2	Param1	1	Shall be the Response attributes. See Table 102 — Chunk sender attributes.
3	Param2	1	Shall be the handle. This field shall be the same for all chunks of the same Large SPDM Response. The value of this field shall be the same value as in `Param2` field of `CHUNK_GET`.
4	ChunkSeqNo	4	Shall identify the chunk sequence number associated with `SPDMchunk`. The value of this field shall be the same value as `ChunkSeqNo` in the `CHUNK_GET`.
8	ChunkSize	4	Shall indicate the size of `SPDMchunk`. See Additional chunk transfer requirements.
12	LargeMessageSize	`L0` = 0 or 4	Shall indicate the size of the large SPDM message being transferred. Shall only be present when `ChunkSeqNo` is zero and shall have a non-zero value. The value of this field should be greater than the `DataTransferSize` of the receiving SPDM endpoint.
12 + `L0`	SPDMchunk	Variable	Shall contain the chunk of the large SPDM request message associated with `ChunkSeqNo`.

823 Figure 25 — Large MEASUREMENT example illustrates the sending and retrieval of a Large SPDM Response that was the result of a Requester issuing a `GET_MEASUREMENTS` request.

824 **Figure 25 — Large MEASUREMENT example**

825



826 10.27.3 Additional chunk transfer requirements

827 When transferring a large SPDM message, an SPDM endpoint shall be prohibited from transferring a chunk sequence number (that is, a `ChunkSeqNo`) less than the current chunk sequence number. In other words, an SPDM endpoint cannot go backwards in the transfer or re-send or re-retrieve a chunk sequence number less than the current one in the transfer. However, due to retries, an SPDM endpoint might re-send or re-retrieve the current chunk number in the transfer. Additionally, if the receiving SPDM endpoint receives an out-of-order chunk sequence

number, the receiving SPDM endpoint shall either silently discard the request or respond with an `ERROR` message of `ErrorCode=InvalidRequest`.

828 The value of `ChunkSize` fields shall be one that ensures the total size of `CHUNK_SEND` or `CHUNK_RESPONSE` does not exceed the `DataTransferSize` of the receiving SPDM endpoint. For all chunks that are not the last chunk, `ChunkSize` shall be a value where the total size of `CHUNK_SEND` or `CHUNK_RESPONSE` shall be from `MinDataTransferSize` to the `DataTransferSize` of the receiving SPDM endpoint. For the last chunk, `ChunkSize` shall be a value where the total size of `CHUNK_SEND` or `CHUNK_RESPONSE` shall be equal to or less than the `DataTransferSize` of the receiving SPDM endpoint.

829 While this transfer mechanism can carry any Request or Response, this transfer mechanism shall prohibit `CHUNK_SEND`, `CHUNK_GET`, and their corresponding responses to be transferred as chunks themselves. Additionally to ensure the general interoperability and reliability of this transfer mechanism, these messages shall be prohibited from being transferred in chunks using this transfer mechanism:

- `GET_VERSION`
- `VERSION`
- `GET_CAPABILITIES`
- `CAPABILITIES` with `Param1` in the `GET_CAPABILITIES` request set to 0.
- `ERROR`
  - An `ERROR` message with an `ErrorCode` other than `LargeResponse` can be placed in the `ResponseToLargeRequest` of a `CHUNK_SEND_ACK` response.

830 This transfer mechanism can carry Requests and Responses that are involved in signature generation or verification and other cryptographic computations. However, this transfer mechanism is not part of any signature generation or verification or cryptographic computation. In other words, `CHUNK_SEND`, `CHUNK_GET`, and their corresponding responses shall not become part of any data or bit stream, such as message transcript, transcript, and so on, that are used to verify or generate a signature or other cryptographic information. Signature generation, signature verification, and other cryptographic computations operate on the large SPDM messages, themselves, which other parts of this specification define.

831 The `ERROR` message of `ErrorCode=ResponseNotReady` shall not be used to directly respond to `CHUNK_SEND` and `CHUNK_GET` requests. However, the `ResponseToLargeRequest` can contain an `ERROR` message of `ErrorCode=ResponseNotReady`.

832 While a large SPDM message is being transferred in chunks, this large SPDM message is not considered a complete SPDM message until the last chunk is received. Therefore, as soon as the `CHUNK_SEND` request begins transmission, this large SPDM request message is considered to be outstanding.

833 10.28 Key configuration

834 Key configuration is the ability to retrieve or configure various parameters pertaining to asymmetric keys for a given SPDM endpoint. These clauses describe the requests and responses that provide key-configuration capabilities.

835 SPDM endpoints can contain key pair ID(s) (`KeyPairID`) that are fixed and already provisioned, key pair IDs that are configurable, or an assortment of both types. For configurable key pair IDs, one or more parameters related to the

key pair are configurable. The requests and responses in these clauses provide the details for each `KeyPairID`. An SPDM endpoint shall contain `KeyPairID` s starting from 1 to `TotalKeyPairs` inclusive and without gaps regardless of the value of `MULTI_KEY_CAP`. Additionally, `TotalKeyPairs` is a fixed number and represents the maximum number of key pair IDs in the corresponding SPDM endpoint per SPDM connection.

836 The Responder should authorize the Requester before allowing it to change information related to a key pair. The method of authorization is outside the scope of this specification.

837 In general, if a key pair ID is configurable, the high-level flow for provisioning and configuring a key pair ID to a usable state should follow these steps:

1. Use the `GET_KEY_PAIR_INFO` request and its corresponding response to retrieve information about one or more key pair ID(s).
2. Use the `SET_KEY_PAIR_INFO` request and its corresponding response to configure the key pair ID.
   - Ensure the key pair ID is associated with one or more certificate slots.
3. Use the `GET_CSR` and/or `SET_CERTIFICATE` requests and their corresponding responses to provision a certificate chain to one or more of the certificate slots the key pair ID is associated with.

838 To return a key pair ID to its initial or default values, follow these steps:

1. Use the `GET_KEY_PAIR_INFO` request and its corresponding response to retrieve information about the desired key pair ID.
   - In particular, note all the certificate slots the key pair ID is associated with.
2. Use the `SET_CERTIFICATE` request and its corresponding response to erase all certificate chains associated with the key pair ID.
3. Use the `SET_KEY_PAIR_INFO` request and its corresponding response to erase the key pair ID.

839 Outside of a session, the Requester and Responder should only issue `GET_KEY_PAIR_INFO`, `SET_KEY_PAIR_INFO`, and their corresponding responses while in a trusted environment.

840 Lastly, when `PUB_KEY_ID_CAP` is set, keys associated with `PUB_KEY_ID_CAP` shall not be associated with any value of `KeyPairID`.

841 10.28.1 GET_KEY_PAIR_INFO request and KEY_PAIR_INFO response

842 The `GET_KEY_PAIR_INFO` request shall retrieve key pair information from the Responder. This request and its response shall report information for all key pairs on the Responder independent of any negotiated parameters of the current SPDM connection. This allows the Requester to inquire about key pair information for all key pair IDs without restarting the SPDM connection.

843 Table 107 — GET_KEY_PAIR_INFO request message format shows the `GET_KEY_PAIR_INFO` request message format.

844 **Table 107 — GET_KEY_PAIR_INFO request message format**

Byte offset	Field	Size (bytes)	Description
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.

Byte offset	Field	Size (bytes)	Description
1	RequestResponseCode	1	`GET_KEY_PAIR_INFO=0xFC` . See Table 4 — SPDM request codes.
2	Param1	1	Reserved.
3	Param2	1	Reserved.
4	KeyPairID	1	The value of this field shall indicate which key pair ID's information to retrieve.

845 The corresponding successful response shall be the `KEY_PAIR_INFO` response as Table 108 — KEY_PAIR_INFO response message format describes.

846 **Table 108 — KEY_PAIR_INFO response message format**

Byte offset	Field	Size (bytes)	Description
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	RequestResponseCode	1	`KEY_PAIR_INFO = 0x7C` . See Table 5 — SPDM response codes.
2	Param1	1	Reserved.
3	Param2	1	Reserved.
4	TotalKeyPairs	1	The value of this field shall indicate the total number of key pairs on the Responder.
5	KeyPairID	1	The value of this field shall be the same value as the `KeyPairID` field in the corresponding request. The remaining fields in this response shall pertain to the requested key pair ID in the corresponding Request.
6	Capabilities	2	This field indicates the capabilities of the requested key pair (`KeyPairID`). The format of this field shall be as Table 109 — Key pair capabilities format defines.
8	KeyUsageCapabilities	2	This field shall indicate the key usages the Responder allows. The format of this field shall be as Key usage bit mask defines. At least one bit shall be set. The Responder shall indicate support for one or more key usages by setting the corresponding bits.

Byte offset	Field	Size (bytes)	Description
10	CurrentKeyUsage	2	This field shall indicate the currently configured key usage for the requested key pair ID. The format of this field shall be as Key usage bit mask defines. If no bits are set, this field shall indicate that the key usage for this key pair ID has not yet been configured. More than one bit can be set. If a bit is set, the Responder shall support cryptographic operations (such as signature generation) for the corresponding key usage.
12	AsymAlgoCapabilities	4	This field shall indicate the asymmetric algorithms the Responder supports for this key pair ID. The format of this field shall be as Table 110 — Asymmetric algorithm capabilities format defines. The Responder shall indicate support for one or more asymmetric algorithms by setting the corresponding bits. At least one bit shall be set in this field or `PqcAsymAlgoCapabilities` .
16	CurrentAsymAlgo	4	This field shall indicate the currently configured asymmetric algorithm for this key pair ID. The format of this field shall be as Table 110 — Asymmetric algorithm capabilities format defines. The total number of bits set in this field and `CurrentPqcAsymAlgo` shall be no more than one. If no bits are set in both fields, it shall indicate that any asymmetric algorithm for this key pair has not yet been selected. The set bit shall indicate that the corresponding asymmetric algorithm is currently configured.
20	PublicKeyInfoLen	2	This field shall indicate the size in bytes of the `PublicKeyInfo` field in this response. A value of zero shall indicate that the actual key pair is absent or has yet to be generated. Otherwise, the value of this field shall be non-zero.
22	AssocCertSlotMask	1	This field is a bit mask representing the currently associated certificate slots. A set bit at position X shall indicate an association between certificate slot X and the requested `KeyPairID` . If `ShareableCap` is not set and `SET_KEY_PAIR_INFO_CAP` is set, no more than one bit shall be set. Otherwise, any number of bits can be set.

Byte offset	Field	Size (bytes)	Description
23	PublicKeyInfo	PublicKeyInfoLen	The field shall contain the public key information for the requested key pair ID. The format of this field shall be the DER encoding of the `AlgorithmIdentifier` structure in an X.509 v3 certificate. See the "4.1.2.7. Subject Public Key Info" clauses in RFC 5280 for additional details. Within the `AlgorithmIdentifier` structure, the `parameters` member shall be present and contain values consistent with the information pertaining to the requested key pair ID.
23 + `PublicKeyInfoLen`	PqcAsymAlgoCapLen	4	This field shall indicate the size in bytes of the `PqcAsymAlgoCapabilities` field in this response.
27 + `PublicKeyInfoLen`	PqcAsymAlgoCapabilities	`PqcAsymAlgoCapLen`	This field shall indicate the PQC asymmetric algorithms the Responder supports for this key pair ID. The format of this field shall be as Table 111 — PQC asymmetric algorithm capabilities format defines. The Responder shall indicate support for one or more PQC asymmetric algorithms by setting the corresponding bits. At least one bit shall be set in this field or `AsymAlgoCapabilities`.
27 + `PublicKeyInfoLen` + `PqcAsymAlgoCapLen`	CurrentPqcAsymAlgoLen	4	This field shall indicate the size in bytes of the `CurrentPqcAsymAlgo` field in this response.
31 + `PublicKeyInfoLen` + `PqcAsymAlgoCapLen`	CurrentPqcAsymAlgo	`CurrentPqcAsymAlgoLen`	This field shall indicate the currently configured PQC asymmetric algorithm for this key pair ID. The format of this field shall be as Table 111 — PQC asymmetric algorithm capabilities format defines. The total number of bits set in this field and `CurrentAsymAlgo` shall be no more than one. If no bits are set in both fields, it shall indicate that the any asymmetric algorithm for this key pair has not yet been selected. The set bit shall indicate that the corresponding PQC asymmetric algorithm is currently configured.

847 Table 109 — Key pair capabilities format defines the format for capabilities associated with a key pair ID.

848 **Table 109 — Key pair capabilities format**

Bit offset	Field	Description
0	GenKeyCap	If set, this key pair identified by the given `KeyPairID` can be generated or regenerated.
1	ErasableCap	If set, this key pair identified by the given `KeyPairID` can be erased.

Bit offset	Field	Description
2	CertAssocCap	If set, the Responder allows a Requester to change the association between the given `KeyPairID` and `CertSlot` .
3	KeyUsageCap	If set, the Responder allows a Requester to change the key usage for the given `KeyPairID` .
4	AsymAlgoCap	If set, the Responder allows a Requester to change the asymmetric algorithm or the PQC asymmetric algorithm for the given `KeyPairID` .
5	ShareableCap	If set, the Responder allows a Requester to associate the given `KeyPairID` with more than one `CertSlot` . This bit shall not be set if `CertAssocCap` is not set.
All other bits	Reserved	Reserved.

849 Table 110 — Asymmetric algorithm capabilities format defines the bit mapping for asymmetric algorithms support. See Table 142 — SPDM Asymmetric Signature Reference Information for references for the asymmetric algorithms.

850 **Table 110 — Asymmetric algorithm capabilities format**

Bit offset	Asymmetric Algorithm
0	RSA 2048
1	RSA 3072
2	RSA 4096
3	ECC NIST P256
4	ECC NIST P384
5	ECC NIST P521
6	SM2 P256
7	Ed25519
8	Ed448
All other bits	Reserved.

851 **Table 111 — PQC asymmetric algorithm capabilities format**

Bit offset	PQC Asymmetric Algorithm
0	ML-DSA-44
1	ML-DSA-65
2	ML-DSA-87
3	SLH-DSA-SHA2-128s
4	SLH-DSA-SHAKE-128s
5	SLH-DSA-SHA2-128f
6	SLH-DSA-SHAKE-128f
7	SLH-DSA-SHA2-192s
8	SLH-DSA-SHAKE-192s
9	SLH-DSA-SHA2-192f
10	SLH-DSA-SHAKE-192f
11	SLH-DSA-SHA2-256s
12	SLH-DSA-SHAKE-256s
13	SLH-DSA-SHA2-256f
14	SLH-DSA-SHAKE-256f
All other bits	Reserved.

852 10.28.2 SET_KEY_PAIR_INFO request and SET_KEY_PAIR_INFO_ACK response

853 The `SET_KEY_PAIR_INFO` request and the corresponding successful `SET_KEY_PAIR_INFO_ACK` response shall configure one or more parameters for one key pair ID (`KeyPairID`).

854 Table 112 — SET_KEY_PAIR_INFO request message format defines the format for the `SET_KEY_PAIR_INFO` request.

855 **Table 112 — SET_KEY_PAIR_INFO request message format**

Byte offset	Field	Size (bytes)	Description
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	RequestResponseCode	1	`SET_KEY_PAIR_INFO` = `0xFD` . See Table 4 — SPDM request codes.

Byte offset	Field	Size (bytes)	Description
2	Param1	1	Operation. This field shall indicate the desired operation. The format of this field shall be the format as Table 113 — Key pair operations defines. If the operation is `KeyPairErase`, all fields after `KeyPairID` field in this request shall be absent.
3	Param2	1	Reserved.
4	KeyPairID	1	The value of this field shall indicate the key pair ID's information to change.
5	Reserved	1	Reserved.
6	DesiredKeyUsage	2	This field shall indicate the desired key usage (`KEY_PAIR_INFO` . `CurrentKeyUsage`) for the requested key pair ID (`KeyPairID`). The format of this field shall be as Key usage bit mask defines. If no bits are set, the Responder shall not change the current key usage. More than one bit can be set. The Requester shall only select from bits that are set in the `KeyUsageCapabilities` field of the `KEY_PAIR_INFO` response for the requested `KeyPairID`. If `KeyUsageCap` is not set for the requested `KeyPairID`, this field shall be zero.
8	DesiredAsymAlgo	4	This field shall indicate the desired asymmetric algorithm (`KEY_PAIR_INFO` . `CurrentAsymAlgo`) for the requested key pair ID. The format of this field shall be as Table 110 — Asymmetric algorithm capabilities format defines. If no bits are set, the Responder shall not change the current configuration for the asymmetric algorithm. The total number of bits set in this field and `DesiredPqcAsymAlgo` shall be no more than one. The Requester shall only select from bits that are set in the `AsymAlgoCapabilities` field of the `KEY_PAIR_INFO` response for the requested `KeyPairID`. If `AsymAlgoCap` is not set for the requested `KeyPairID`, this field shall be zero.
12	DesiredAssocCertSlotMask	1	This field is a bit mask representing the desired certificate slot association. A set bit at position X shall indicate an association between certificate slot X and the requested `KeyPairID`. An unset bit at position X shall indicate no association between certificate slot X and the requested `KeyPairID`. The Responder shall either remove an association or create an association between the corresponding certificate slot and the requested `KeyPairID`, depending on the value of each bit in this field. If `ShareableCap` is not set, no more than one bit shall be set.

Byte offset	Field	Size (bytes)	Description
13	DesiredPqcAsymAlgoLen	4	This field shall indicate the size in bytes of the `DesiredPqcAsymAlgo` field.
17	DesiredPqcAsymAlgo	`DesiredPqcAsymAlgoLen`	This field shall indicate the desired PQC asymmetric algorithm (`KEY_PAIR_INFO` . `CurrentPqcAsymAlgo`) for the requested key pair ID. The format of this field shall be as Table 111 — PQC asymmetric algorithm capabilities format defines. If no bits are set, the Responder shall not change the current configuration for the PQC asymmetric algorithm. The total number of bits set in this field and `DesiredAsymAlgo` shall be no more than one. The Requester shall only select from bits that are set in the `PqcAsymAlgoCapabilities` field of the `KEY_PAIR_INFO` response for the requested `KeyPairID` . If `AsymAlgoCap` is not set for the requested `KeyPairID` , this field shall be zero.

856 Table 113 — Key pair operations defines a numeric mapping to an operation.

857 **Table 113 — Key pair operations**

Value	Operation Name	Description
0	ParameterChange	Shall indicate an operation that modifies one or more key-related parameters. The `DesiredKeyUsage` , `DesiredAsymAlgo` , and `DesiredAssocCertSlotMask` fields shall be present.
1	KeyPairErase	Shall indicate an operation that erases all information relating to a `KeyPairID` . The `DesiredKeyUsage` , `DesiredAsymAlgo` , and `DesiredAssocCertSlotMask` fields shall be absent.
2	GenerateKeyPair	Shall indicate an operation that generates a new key pair for this `KeyPairID` . The `DesiredKeyUsage` , `DesiredAsymAlgo` , and `DesiredAssocCertSlotMask` fields shall be present.

858 Table 114 — SET_KEY_PAIR_INFO_ACK response message format defines the format for `SET_KEY_PAIR_INFO_ACK` response.

859 **Table 114 — SET_KEY_PAIR_INFO_ACK response message format**

Byte offset	Field	Size (bytes)	Description
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	RequestResponseCode	1	`SET_KEY_PAIR_INFO_ACK` = `0x7D`. See Table 5 — SPDM response codes.
2	Param1	1	Reserved.
3	Param2	1	Reserved.

860 ### 10.28.3 Key pair ID modification error handling

861 These clauses describe some basic configuration error scenarios an SPDM endpoint should handle.

862 The first error scenario is a request for key generation (`GenerateKeyPair`) when no asymmetric algorithm has been selected yet. A Responder should respond with an `ERROR` message of `ErrorCode=OperationFailed`.

863 Key usage for a key pair ID does not need to be specified until the key pair ID is associated with a certificate slot, so this information is not needed for a `GenerateKeyPair` operation. The Responder should decide when it needs to know the key usage information for a configurable key usage.

864 For a `KeyPairErase` or `GenerateKeyPair` operation request, the Responder shall ensure that the requested `KeyPairID` has no association with any certificate slot. Otherwise, the Responder should respond with an `ERROR` message of `ErrorCode=OperationFailed`.

865 There is one key pair per `KeyPairID` and the value of a key pair is bound to the selected asymmetric algorithm (`CurrentAsymAlgo` of `KEY_PAIR_INFO`) of the corresponding `KeyPairID`. Once a key pair is generated for a `KeyPairID`, the Responder shall discard the `SET_KEY_PAIR_INFO` request or return an `ERROR` message with `ErrorCode=InvalidRequest` when that request changes parameters (`ParameterChange`) that affect the generated key pair, such as the asymmetric algorithm.

866 ## 10.29 Event mechanism

867 An SPDM endpoint may want to be notified of changes from another SPDM endpoint. These change notifications are called events. The SPDM event mechanism provides a framework for the asynchronous notification of events over a secure session. An Event Notifier is an SPDM endpoint sending an event, and an Event Recipient is an SPDM endpoint receiving an event. An SPDM endpoint can be both an Event Notifier and an Event Recipient in the same secure session. See Session for details on secure sessions. There can be multiple sessions between the same Responder and same Requester. The event mechanism applies to each session individually.

868 An event is identified by its event group, event type, and an event instance ID. An event group is a group of all event types a given standards body or vendor defines. An event type classifies the event by indicating its type. The event instance ID is a unique numeric value that represents that occurrence of the event.

869 An Event Recipient can select the event types that it wants to receive. An event subscription is a list of event types an Event Recipient wants to receive. The Event Notifier manages the event subscription. An Event Notifier shall only

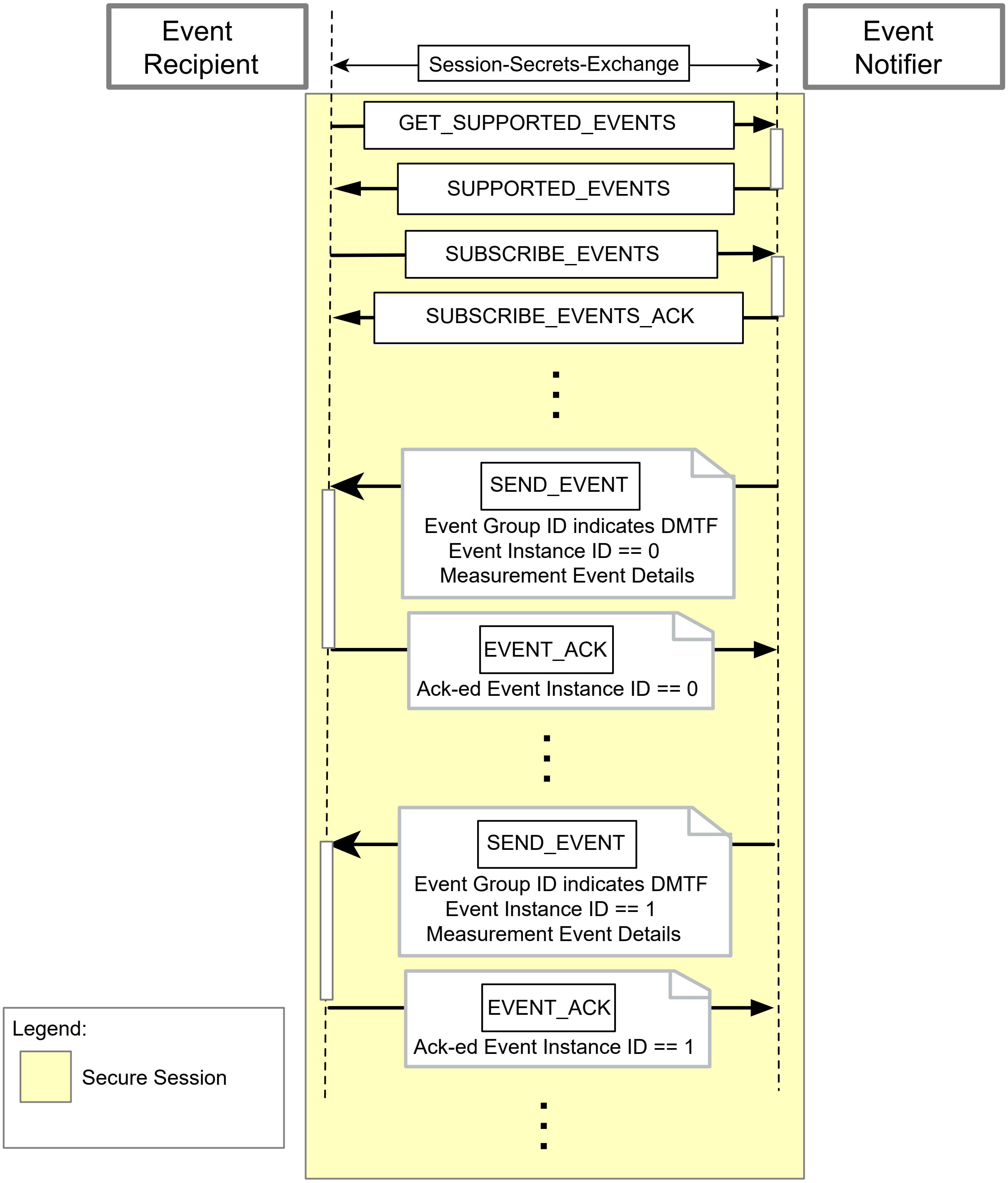
send events of event types that match the event types in the event subscription. See DMTF Event Types for DMTF-defined event types.

870 An Event Notifier shall not send any events in a session until an Event Recipient subscribes to one or more events.

871 The Event Flow diagram illustrates a typical event flow for event subscription and event delivery over a transport capable of asynchronous bidirectional communication.

872 **Figure 26 — Event flow diagram**

873



874 For transports that prohibit a Responder from asynchronously sending out data, the Event Notifier and Event Recipient can use the encapsulated request flow to deliver or receive events. The encapsulated request flow allows for a polling methodology as Triggering `GET_ENCAPSULATED_REQUEST` describes.

875 When `EVENT_CAP` is set, an Event Notifier shall support `SUBSCRIBE_EVENT_TYPES`, `GET_SUPPORTED_EVENT_TYPES`, `SEND_EVENT`, and their corresponding response messages. In addition, an Event Notifier shall support the mandatory DMTF event types.

876 10.29.1 GET_SUPPORTED_EVENT_TYPES request and SUPPORTED_EVENT_TYPES response message

877 These request and response messages retrieve the list of all event types supported by the Event Notifier. Each event type belongs in an event group. An event group contains all event types belonging to the standards body or vendor that defines them. The SVH identifies the event group. Within an event group, an event type ID identifies the event type uniquely within the event group. Both the SVH and the event type ID ensure uniqueness for all event types in this specification.

878 Usually, the Event Notifier does not need to support all event types within an event group or within all event groups. However, the standards body or vendor defines the requirements for the event types they define.

879 Table 115 — GET_SUPPORTED_EVENT_TYPES request message format describes the message format.

880 **Table 115 — GET_SUPPORTED_EVENT_TYPES request message format**

Byte Offset	Field	Size (bytes)	Description
0	`SPDMVersion`	1	Shall be the `SPDMVersion` as described in SPDM version.
1	`RequestResponseCode`	1	`0xE2` = `GET_SUPPORTED_EVENT_TYPES`
2	Param1	1	Reserved.
3	Param2	1	Reserved.

881 Table 116 — SUPPORTED_EVENT_TYPES response message format describes the message format for this response.

882 **Table 116 — SUPPORTED_EVENT_TYPES response message format**

Byte Offset	Field	Size (bytes)	Description
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	`RequestResponseCode`	1	`0x62` = `SUPPORTED_EVENT_TYPES` Response
2	Param1	1	EventGroupCount. Shall be the number of event groups listed in `SupportedEventGroupsList`.
3	Param2	1	Reserved.

Byte Offset	Field	Size (bytes)	Description
4	SupportedEventGroupsListLen	4	The value of this field shall be the size in bytes of the `SupportedEventGroupsList` and shall be greater than zero.
8	SupportedEventGroupsList	`SupportedEventGroupsListLen`	Shall be a list of all event types grouped by event group supported by the Event Notifier. The format of this field shall be a list of Event group. In this format, each event group contains a list of event types the Event Notifier supports. If an event group is present, it shall be present exactly once to avoid duplicates and to minimize the size of this response. The size of this field shall be the value in `SupportedEventGroupsListLen`. See Event group format additional information for additional details.

883 Table 117 — Event group format defines the format for listing event types in a single event group.

884 **Table 117 — Event group format**

Byte Offset	Field	Size (bytes)	Description
0	EventGroupId	2 + `VendorIDLen`	Shall indicate the event group the event type belongs to. The format of this field shall be the SVH format. The size of this field shall be the size of the SVH.
2 + `VendorIDLen`	EventTypeCount	2	Shall be the total number of event types listed in the `EventTypeList` field and belonging to `EventGroupId`. The value of this field shall be greater than zero.

Byte Offset	Field	Size (bytes)	Description
4 + `VendorIDLen`	EventGroupVer	2	Shall be the standards body or vendor-assigned version number that indicates the version of the event types belonging to `EventGroupId`.
6 + `VendorIDLen`	Attributes	4	Attributes. The format of this field shall be defined by the messages using this Event groups format. For the `SUPPORTED_EVENT_TYPES` response message, see Event group format additional information. For the `SUBSCRIBE_EVENT_TYPES` request message, see Additional subscription list information.
10 + `VendorIDLen`	EventTypeList	Variable	Shall be a list of event types in this Event Group (`EventGroupId`). The value in `EventTypeCount` field shall indicate the number of event types in this list. The format of this field shall be a list of Event Type Information. If an event type is present, it shall be present exactly once.

885 Table 118 — Event type information format defines the format for a single event type.

886 **Table 118 — Event type information format**

Byte Offset	Field	Size (bytes)	Description
0	EventTypeId	2	Shall be a numeric value that uniquely identifies this event type within the corresponding event group.

Byte Offset	Field	Size (bytes)	Description
2	Reserved	2	Reserved.

887 The `EventGroupVer` field allows for updates to the event type list such as a new event type. An Event Notifier should add new event types to the end of the list.

888 10.29.1.1 Event group format additional information

889 This clause describes further information for various fields in the Event groups format table. This format is present in more than one SPDM message.

890 Many fields in the Event group format table have different definitions depending on which SPDM message uses this table. For `SUBSCRIBE_EVENT_TYPES`, see Additional subscription list information for requirements on the Event group format.

891 The following requirements shall apply to the Event group format table contained in `SUPPORTED_EVENT_TYPES`.

- The value of `EventTypeCount` field shall be greater than zero.
- The presence of an event type in the `EventTypeList` field shall indicate that the Event Notifier can send events of this type.
- The value of `Attributes` shall be reserved.

892 10.29.2 SUBSCRIBE_EVENT_TYPES request and SUBSCRIBE_EVENT_TYPES_ACK response message

893 The `SUBSCRIBE_EVENT_TYPES` request and `SUBSCRIBE_EVENT_TYPES_ACK` response messages allow an Event Recipient to communicate the list of SPDM event types it is interested in receiving. This request replaces the current subscription list.

894 An event subscription is a list of all event types to which an Event Recipient subscribes. Thus, an Event Notifier shall send events when they occur to an Event Recipient if at least one event type is present in the event subscription of the corresponding Event Recipient.

895 To subscribe or unsubscribe to an event group, an Event Recipient shall send the `SUBSCRIBE_EVENT_TYPES` request message with a complete list of all event types to which the Event Recipient subscribes. An Event Notifier shall replace the current event subscription with the new subscription from the latest `SUBSCRIBE_EVENT_TYPES` message. If the new subscription contains an unsupported or invalid event type, the Responder should respond with an `ERROR` message of `ErrorCode=InvalidRequest`. If an Event Notifier supports multiple Event Recipients, the Event Notifier shall support a unique event subscription list per session for each subscribed Event Recipient. The SUBSCRIBE_EVENT_TYPES request message format describes the message format.

896 **Table 119 — SUBSCRIBE_EVENT_TYPES request message format**

Byte Offset	Field	Size (bytes)	Description
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.

Byte Offset	Field	Size (bytes)	Description
1	`RequestResponseCode`	1	`0xF0` = `SUBSCRIBE_EVENT_TYPES`
2	Param1	1	SubscribeEventGroupCount. Shall be the number of event groups in `SubscribeList`. A value of zero shall indicate that the Event Recipient no longer subscribes to any events. This is the equivalent of an empty event subscription or the removal of all event types in an event subscription. If the value of this field is zero, `SubscribeListLen` and `SubscribeList` fields shall be absent.
3	Param2	1	Reserved.
4	SubscribeListLen	4	The value of this field shall be the size in bytes of `SubscribeList`. The value of this field shall be greater than zero.
8	SubscribeList	`SubscribeListLen`	Shall be a list of event types grouped by event group that the Event Notifier supports and to which the Event Recipient is subscribing. The format of this field shall be a list of Event group. In this format, each event group contains a list of event types to which the Event Recipient subscribes. If an event group is present, it shall be present exactly once. The size of this field shall be the value in `SubscribeListLen` field. See Additional subscription list information for additional requirements.

897 Table 120 — SUBSCRIBE_EVENT_TYPES_ACK response message format describes the response format for the `SUBSCRIBE_EVENT_TYPES` request.

898 **Table 120 — SUBSCRIBE_EVENT_TYPES_ACK response message format**

Byte Offset	Field	Size (bytes)	Description
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	`RequestResponseCode`	1	`0x70` = `SUBSCRIBE_EVENT_TYPES_ACK` Response
2	Param1	1	Reserved.
3	Param2	1	Reserved.

899 For event types defined by this specification, see DMTF event types.

900 **10.29.2.1 Additional subscription list information**

901 These clauses describe further information for various fields in `SubscribeList` whose format is the Event group format.

902 The value of the `EventTypeCount` field shall be greater than or equal to zero. If `EventTypeCount` is zero, then `AllEventTypes` shall also be set.

903 The presence of an event type in the `EventTypeList` field shall subscribe the Event Recipient to that event type. Likewise, the absence of an event type in the `EventTypeList` field shall indicate that the Event Recipient does not or no longer subscribes to this event type. Additionally, the absence of an event group in the `SubscribeList` shall indicate that the Event Recipient does not or no longer subscribes to any event types in this event group.

904 The format of the `Attributes` field shall be as the `SUBSCRIBE_EVENT_TYPES` request attributes format table defines.

905 **Table 121 — SUBSCRIBE_EVENT_TYPES request attributes format**

Byte Offset	Bit Offset	Field	Description
0	0	AllEventTypes	If set, the Event Notifier shall subscribe the Event Recipient to all event types supported by the Event Notifier in the corresponding Event Group and the value of `EventTypeCount` shall be zero.
0	[7:1]	Reserved	Reserved
1	[7:0]	Reserved	Reserved
2	[7:0]	Reserved	Reserved
3	[7:0]	Reserved	Reserved

906 If an Event Recipient sets `AllEventTypes`, it can receive events of event types it does not understand. In this scenario, the Event Recipient shall respond with an `EVENT_ACK` message as SEND_EVENT request and EVENT_ACK response message describes and stop processing the unknown event type.

907 ## 10.29.3 SEND_EVENT request and EVENT_ACK response message

908 To deliver subscribed events to an Event Recipient, the Event Notifier shall use the `SEND_EVENT` request message. This request can contain more than one event.

909 Table 122 — SEND_EVENT request message format describes this request.

910 **Table 122 — SEND_EVENT request message format**

Byte Offset	Field	Size (bytes)	Description
0	`SPDMVersion`	1	Shall be the `SPDMVersion` as described in SPDM version.
1	`RequestResponseCode`	1	`0xF1` = `SEND_EVENT`
2	Param1	1	Reserved.
3	Param2	1	Reserved.
4	EventCount	4	Shall be the number of elements in `EventsList`.
8	EventsList	Variable	Shall be a list of Event Data. The list should be sorted in numerically increasing event instance ID order. The size of this field shall be the size of this list.

911 Table 123 — Event data table describes the format for details of each event.

912 **Table 123 — Event data table**

Byte Offset	Field	Size (bytes)	Description
0	EventInstanceId	4	Shall be the event instance id for the event.
4	Reserved	4	Reserved.
8	EventGroupId	2 + `VendorIDLen`	Shall indicate the event group the event type belongs to. The format of this field shall be SVH format.
10 + `VendorIDLen`	EventTypeId	2	Shall be the numeric value identifying the event type of this event in `EventGroupId`.
12 + `VendorIDLen`	EventDetailLen	2	Shall be the length of `EventDetail`.
14 + `VendorIDLen`	EventDetail	Variable	Shall be the event-specific details of the event indicated by `EventInstanceId`, `EventGroupId` and `EventTypeId`. The format and further definition of this field is specific to the event type indicated by `EventTypeId` in the event group indicated by `EventGroupId`. For the DMTF event group, see Event type details for further information. The size of this field shall be the size of the event-specific details for this event.

913 Table 124 — EVENT_ACK response message format describes the format for the response.

914 **Table 124 — EVENT_ACK response message format**

Byte Offset	Field	Size (bytes)	Description
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	`RequestResponseCode`	1	`0x71` = `EVENT_ACK` Response
2	Param1	1	Reserved.
3	Param2	1	Reserved.

915 The Event Notifier shall only send unacknowledged event instance IDs.

916 The size of `SEND_EVENT` data can exceed the `DataTransferSize` of the Event Recipient, especially if multiple events happen concurrently. While it is possible to use the Large SPDM message transfer mechanism, the Event Notifier should try to divide the events into multiple `SEND_EVENT` requests to ensure efficient delivery of the events instead of combining all events into a single `SEND_EVENT` request.

917 An Event Notifier shall send a `SEND_EVENT` request with only the Event Lost event (`EventTypeId` =EventLost) as an indication that the original event was too big in size under any of these conditions:

- The Event Notifier does not support the Large SPDM message transfer mechanism and the `SEND_EVENT` request with only one event exceeds the `DataTransferSize` of the Event Recipient.
- The size of a `SEND_EVENT` request with only one event is greater than the `MaxSPDMmsgSize` of the Event Recipient.

918 The Event Notifier shall follow the requirements in Timing requirements as a Requester for `SEND_EVENT`. Likewise, the Event Recipient shall follow the timing requirements as a Responder when receiving a `SEND_EVENT` request.

919 10.29.4 Event Instance ID

920 Event Instance ID typically reflects the order of events in the Event Notifier from a chronological perspective. The event instance ID shall start at zero for each secure session and sequentially increase with each occurrence of an event. This method also allows the Event Recipient to determine if an event was lost.

921 When the event instance ID reaches the maximum value, the Event Notifier shall terminate the session after sending a `SEND_EVENT` request containing an event with the maximum value and receiving the corresponding response. An Event Recipient can also terminate the session.

922 ## 10.30 GET_ENDPOINT_INFO request and ENDPOINT_INFO response messages

923 The `GET_ENDPOINT_INFO` request message shall retrieve general information from an endpoint. The `SubCode` parameter is used to differentiate between operations, and a request message shall specify only one `SubCode`. If the Responder does not support the specified `SubCode`, the responder shall return an `ERROR` message of `ErrorCode=UnsupportedRequest`.

924 Table 125 — GET_ENDPOINT_INFO request format shows the format of the `GET_ENDPOINT_INFO` request message.

925 Table 128 — ENDPOINT_INFO response format shows the format of the `ENDPOINT_INFO` response message.

926 **Table 125 — GET_ENDPOINT_INFO request format**

Byte offset	Field	Size (bytes)	Description
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	RequestResponseCode	1	`0x87` = `GET_ENDPOINT_INFO`. See Table 4 — SPDM request codes.
2	Param1	1	Shall be the `GET_ENDPOINT_INFO` SubCode. See GET_ENDPOINT_INFO SubCodes for the list of valid values.
3	Param2	1	Bit [7:4]. Reserved. Bit [3:0]. `SlotID` that identifies the certificate chain whose leaf certificate is used to sign the response. If a signature is not requested (Bit[0] of the `RequestAttributes` field is `0`), this field shall be ignored. If the Responder's public key was provisioned to the Requester previously, this field shall be `0xF`.
4	RequestAttributes	1	Request attributes. See GET_ENDPOINT_INFO request attributes.
5	Reserved	3	Reserved.
8	Nonce	`NL` = 32 or 0	The Requester should choose a random value. This field shall only be present if a signature is requested (`SignatureRequested=1b`).

927 **Table 126 — GET_ENDPOINT_INFO SubCodes**

`SubCode`	Value	Description
Reserved	0x00	Reserved.

SubCode	Value	Description
DeviceClassIdentifier	0x01	The `DeviceClassIdentifier` response returns information that can be used to identify the class of device for the Responder in question. See ENDPOINT_INFO device class identifier list format for the definition of the response data.
Reserved	All other values	SPDM implementations compatible with this version shall not use the reserved `SubCode` s.

928 **Table 127 — GET_ENDPOINT_INFO request attributes**

Bit offset	Field	Description
0	SignatureRequested	If the Responder can generate a signature (`EP_INFO_CAP=10b` in its `CAPABILITIES` response and either `BaseAsymSel` or `ExtAsymSelCount` is non-zero), a value of `1` indicates that a signature on the response is required. When this bit is set to `1` , the Requester shall include the `Nonce` field in the request, and the Responder shall generate a signature and send the signature in the response. A value of `0` indicates that the Requester does not require a signature. The Responder shall not generate a signature in the response. The `Nonce` field shall be absent in the request and response. For Responders that cannot generate a signature (`EP_INFO_CAP=01b` in their `CAPABILITIES` response or both `BaseAsymSel` and `ExtAsymSelCount` are zero), the Requester shall always set this bit to `0` .
[7:1]	Reserved	Reserved.

929 **Table 128 — ENDPOINT_INFO response format**

Byte offset	Field	Size (bytes)	Description
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	RequestResponseCode	1	`0x07` = `ENDPOINT_INFO` . See Table 5 — SPDM response codes.
2	Param1	1	Reserved.
3	Param2	1	Bit [7:4]. Reserved. Bit [3:0]. `SlotID` that identifies the certificate chain whose leaf certificate is used to sign the response. If a signature is not requested (`SignatureRequested=0b`), this field shall be `0` . If the Responder's public key was provisioned to the Requester previously, this field shall be `0xF` .

Byte offset	Field	Size (bytes)	Description
4	Reserved	4	Reserved.
8	Nonce	`NL` = 32 or 0	The Responder should choose a random value. This field shall only be present if Bit[0] of the `RequestAttributes` field is `1`.
8 + `NL`	EPInfoLen	4	Shall contain the length of the `EPInfo` field.
12 + `NL`	EPInfo	`EPInfoLen`	Shall contain endpoint information, as described in the endpoint information format for the specified `SubCode`. The size of this field shall be the size of the returned endpoint information.
12 + `NL` + `EPInfoLen`	Signature	`SigLen`	`Signature` of the endpoint information, excluding the `Signature` field and signed using the private key associated with the leaf certificate. The Responder shall use the asymmetric signing algorithm it selected during the last `ALGORITHMS` response message to the Requester, and `SigLen` is the output size for that asymmetric signing algorithm. This field is conditional and only present in the `ENDPOINT_INFO` response corresponding to a `GET_ENDPOINT_INFO` request with the `SignatureRequested` bit set to 1 in the `RequestAttributes` field. See ENDPOINT_INFO signature generation and ENDPOINT_INFO signature verification for more details.

930 The Device Class Identifier format is an extended form of the standards body or vendor-defined header. For a Device Class Identifier list response, `EPInfoLen` shall have a size of 4 + `IDElemSize`. The `IDElemSize` shall be the sum of the sizes of the `NumIdentifiers` of the Device Class Identifier elements. Each Device Class Identifier shall have a size of 4 + `VendorIDLen` + the sum of the sizes of the subordinate Device Class Identifiers. Each of the subordinate Device Class Identifiers shall have a size of 1 + `SubIDLen`, where `SubIDLen` may be different for each element.

931 **Table 129 — ENDPOINT_INFO device class identifier list format**

Byte offset	Field	Size (bytes)	Description
0	NumIdentifiers	1	Shall be the number of Device Class Identifier elements in this response message. Each identifier shall be unique.
1	Reserved	3	Reserved.
4	IdentifierElements	`IDElemSize`	Shall contain Device Class Identifier elements, as defined in ENDPOINT_INFO device class identifier element format.

932 **Table 130 — ENDPOINT_INFO device class identifier element format**

Byte offset	Field	Size (bytes)	Description
0	IDElemLength	1	Shall be the size of this ID element. The value of `IDElemLength` shall be the number of bytes from the `SVH` . `ID` field through the last `SubordinateID` , inclusive.
1	SVH	2 + `VendorIDLen`	Shall be a standards body or vendor-defined header, as described in Table 69 — Standards body or vendor-defined header (SVH).
3 + `VendorIDLen`	NumSubIDs	1	Shall be the number of subordinate Device Class Identifiers.
4 + `VendorIDLen`	SubordinateID	`NumSubIDs` entries of 1 + `SubIDLen` for a given entry	Shall contain `NumSubIDs` of subordinate Device Class Identifiers, of the format described in Device class identifier subordinate identifier format. If `NumSubIDs` is 0, this field shall be absent.

933 If present, one or more subordinate identifier fields contain identifiers that further identify the device. These identifiers shall be valid in the namespace defined by the standards body specified in the `ID` field and by the vendor ID specified in the `VendorID` field.

934 **Table 131 — Device class identifier subordinate identifier format**

Byte offset	Field	Size (bytes)	Description
0	SubIDLen	1	Shall contain the length in bytes of this subordinate identifier.
1	SubIdentifier	`SubIDLen`	Shall contain one subordinate device identifier that is valid in the namespace of the vendor identified in the `VendorID` field. This field shall be size `SubIDLen` .

935 10.30.1 ENDPOINT_INFO signature generation

936 The signature for an `ENDPOINT_INFO` response is generated per request and response pair. To complete the `ENDPOINT_INFO` signature generation process, the Responder shall complete these steps:

937 1. The Responder shall construct an information log IL1, and the Requester shall construct an information log IL2 over their observed messages:

```
IL1/IL2 = Concatenate(VCA, GET_ENDPOINT_INFO, ENDPOINT_INFO)
```

938 where:

- `Concatenate` is the standard concatenation function.
- `GET_ENDPOINT_INFO` is the entire `GET_ENDPOINT_INFO` request message under consideration where

the Requester has set the `SignatureRequested` bit in the `RequestAttributes` field.

- `ENDPOINT_INFO` is the entire `ENDPOINT_INFO` response message under consideration, except for the signature field.

939 2. The Responder shall generate:

```
Signature = SPDMsign(PrivKey, IL1, "endpoint_info signing")
```

940 where:

- `SPDMsign` is described in Signature generation.
- `PrivKey` shall be the private key of the Responder associated with the leaf certificate stored in `SlotID` of `Param2` in `GET_ENDPOINT_INFO`. If the public key of the Responder was provisioned to the Requester, then `PrivKey` shall be the associated private key.

941 ### 10.30.2 ENDPOINT_INFO signature verification

942 To complete the `ENDPOINT_INFO` signature verification process, the Requester shall complete this step:

943 1. The Requester shall perform:

```
result = SPDMsignatureVerify(PubKey, Signature, IL2, "endpoint_info signing")
```

944 where:

- `SPDMsignatureVerify` is described in Signature verification. A successful verification is when `result` is `success`.
- `PubKey` shall be the public key associated with the leaf certificate stored in `SlotID` of `Param2` in `GET_ENDPOINT_INFO`, and it is extracted from the `CERTIFICATE` response. If the public key of the Responder was provisioned to the Requester, then `PubKey` shall be the provisioned public key.

945 ## 10.31 Measurement extension log mechanism

946 A Responder device may create and maintain a Measurement Extension Log (MEL) to record device information such as measurements of firmware and/or software modules loaded during the boot, firmware and/or software updates, configurations, status of the system, and so on. To construct the MEL, when certain events occur, the Responder appends data associated with the events to the end of the MEL. The events that cause the MEL update are specific to and are determined by individual Responder implementations. For example, the Responder may append the digest and version number of a firmware module to the end of the MEL when the firmware module is loaded. The MEL grows as entries are added. At reset, the Responder may reset the MEL or preserve the MEL. If the Responder preserves the MEL across resets, the reset events themselves may be added as new entries to the

MEL. Accordingly, the corresponding HEM should also be preserved across resets. The Responder should ensure that the MEL will not overrun memory or wrap under normal uses.

947 If the `MEL_CAP` bit in `CAPABILITIES` is set, the Requester may acquire the MEL of the Responder by issuing a `GET_MEASUREMENT_EXTENSION_LOG` request message. The Responder shall respond with the `MEASUREMENT_EXTENSION_LOG` response message. If a Requester acquires the hash-extend measurements outside of a secure session, the Requester should set `SignatureRequested=1` in the `GET_MEASUREMENTS` request or secure the response using other means outside of this specification.

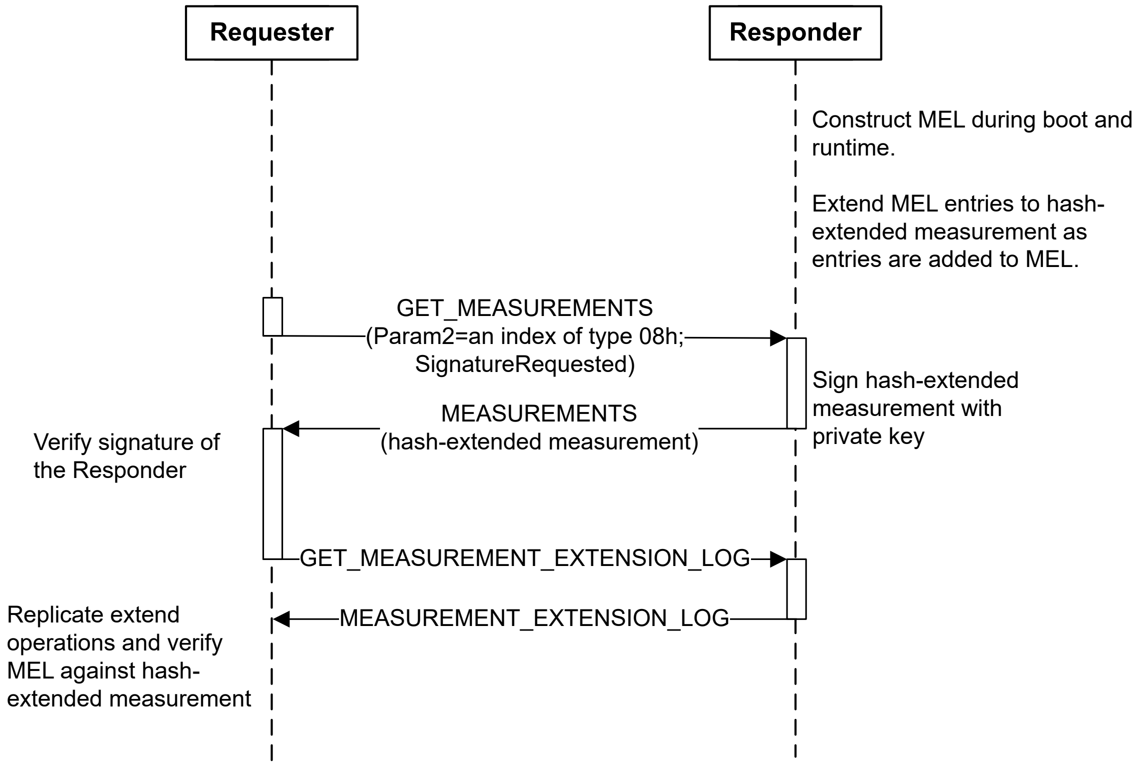
948 The Hash-extend measurements clause introduces a method of constructing a hash value (type `0x8` of `DMTFSpecMeasurementValueType[6:0]`) by extending measurements. The resulting hash guarantees the integrity of the data participating in the extend operations. Leveraging this mechanism can ensure the integrity of the MEL. To do this, an entry of the MEL serves as the `DataToExtend` in calculating `HEM`. After all entries of the MEL are processed, the resulting `HEM` is the hash-extend measurement.

949 To avoid circular dependencies and race conditions, the `DataToExtend` for calculating HEM shall not include the `GET_MEASUREMENTS` request, `MEASUREMENTS` response, `GET_MEASUREMENT_EXTENSION_LOG` request, or `MEASUREMENT_EXTENSION_LOG` response messages.

950 Figure 27 — Flow for acquiring Hash-Extend Measurement and Measurement Extension Log demonstrates an example flow for the Requester to obtain hash-extend measurement and the MEL from the Responder.

951 **Figure 27 — Flow for acquiring Hash-Extend Measurement and Measurement Extension Log**

952



953 As the example flow shows, a Responder that supports MEL would construct the MEL at runtime independently of the Requester. The Requester would first issue `GET_MEASUREMENTS` to obtain the HEM and verify the signature of the Responder, and then it would issue `GET_MEASUREMENT_EXTENSION_LOG` to obtain the MEL from the Responder. With both HEM and MEL, the Requester replicates the extend operations with every entry of the MEL in ascending MEL index order and compares the result of every extend operation with the HEM received in the `MEASUREMENTS` response. If the two values match, then the integrity of the MEL entries used in the extend operations is verified. Note that if the Responder added new entries to the MEL after generating the HEM in the `MEASUREMENTS` response and before responding to `GET_MEASUREMENT_EXTENSION_LOG`, then the result of the extend operations with all MEL entries would not match the HEM. In this case, the Requester should issue `GET_MEASUREMENTS` again to obtain the updated HEM for verifying the integrity of newly added MEL entries. If the result of the extend operations does not match the HEM and the Responder did not add new entries to the MEL between `MEASUREMENTS` and `GET_MEASUREMENT_EXTENSION_LOG`, then the verification is considered to have failed.

954 10.31.1 GET_MEASUREMENT_EXTENSION_LOG request and MEASUREMENT_EXTENSION_LOG response messages

955 Table 132 — GET_MEASUREMENT_EXTENSION_LOG message format shows the `GET_MEASUREMENT_EXTENSION_LOG` request message format.

956 Table 133 — Successful MEASUREMENT_EXTENSION_LOG message format shows the `MEASUREMENT_EXTENSION_LOG` response message format.

957 **Table 132 — GET_MEASUREMENT_EXTENSION_LOG message format**

Byte offset	Field	Size (bytes)	Description
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	RequestResponseCode	1	`0xEF` = `GET_MEASUREMENT_EXTENSION_LOG`. See Table 4 — SPDM request codes.
2	Param1	1	Reserved.
3	Param2	1	Reserved.
4	Offset	4	Shall be the offset in bytes from the start of the MEL to where the read request message begins. The Responder shall send the MEL starting from this offset. Offset 0 shall be the first byte of the MEL.
8	Length	4	Shall be the length of the MEL, in bytes, to be returned in the corresponding response.

958 Note that the large SPDM message transfer mechanism can be used for the MEASUREMENT_EXTENSION_LOG message. Also note that, if the Responder added new entries to MEL between MEASUREMENT_EXTENSION_LOG responses, then RemainderLength of a later MEASUREMENT_EXTENSION_LOG response may be greater than that of the previous MEASUREMENT_EXTENSION_LOG response.

959 **Table 133 — Successful MEASUREMENT_EXTENSION_LOG response message format**

Byte offset	Field	Size (bytes)	Description
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	RequestResponseCode	1	`0x6F` = `MEASUREMENT_EXTENSION_LOG`. See Table 5 — SPDM response codes.
2	Param1	1	Reserved.
3	Param2	1	Reserved.
4	PortionLength	4	Shall be the number of bytes of this portion of the MEL. This shall be less than or equal to the `Length` received as part of the request. For example, the Responder might set this field to a value less than the `Length` received as part of the request due to limitations on the transmit buffer of the Responder.

Byte offset	Field	Size (bytes)	Description
8	RemainderLength	4	Shall be the number of bytes remaining in the MEL from the requested offset + `PortionLength` . A value of 0 shall indicate there are no more bytes beyond the requested offset + `PortionLength` .
12	MEL	`PortionLength`	Requested contents of the MEL. This field shall follow the format negotiated in the most recent ALGORITHMS message.

960 10.31.2 DMTF Measurement Extension Log Format

961 This clause specifies the format of MEL in the `MEASUREMENT_EXTENSION_LOG` response when the MEL specification (`MELspecificationSel`) is "DMTFmelSpec" and the measurement specification (`MeasurementSpecificationSel`) is "DMTFmeasSpec" in the most recent `ALGORITHMS` message (see Table 23 — Successful ALGORITHMS response message format). The MEL format shown in Table 134 — DMTF Measurement Extension Log format leverages the DMTF measurement specification format for its entries.

962 **Table 134 — DMTF Measurement Extension Log Format**

Byte offset	Field	Size (bytes)	Description
0	NumberOfEntries	4	Shall be the number of entries in the MEL.
4	MELEntriesLength	4	Shall be the total number of bytes in all entries of the MEL.
8	Reserved	8	Reserved.
16	MELEntries	`MELEntriesLength`	Shall be the concatenation of all entries of the MEL. The size of this field shall be equal to `MELEntriesLength` .

963 The MELEntries field of the DMTF Measurement Extension Log consists of all entries of the MEL. Each MEL entry shall follow the format that Table 135 — DMTF Measurement Extension Log Entry Format defines. In the calculation of hash-extend measurement, `DataToExtend` shall be one MEL entry at a time.

964 **Table 135 — DMTF Measurement Extension Log Entry Format**

Byte offset	Field	Size (bytes)	Description
0	MELIndex	4	Shall be the index of this entry in the MEL. This field shall be a non-negative integer. The MELIndex shall be in increasing order.

Byte offset	Field	Size (bytes)	Description
4	MeasIndex	1	Shall be the index of the hash-extend measurement which this entry extends, that is, the `Index` of Table 57 — Measurement block format for this hash-extend measurement (`DMTFSpecMeasurementValueType[6:0]` = `0x8`) in the `MEASUREMENTS` response. MeasIndex values of MEL entries can interleave. For example, it is legitimate that a MELIndex of 2 has a MeasIndex of 0x04, but a MELIndex of 1 and a MELIndex of 3 both have a MeasIndex of 0x05. If this entry does not extend to any index, then the Responder shall set this field to `0x00` . In this case, the entry shall not be used in the extend operation for calculating HEM. Some indices are reserved for specific purpose (see Table 55 — Measurement index assigned range).
5	Reserved	3	Reserved.
8	Entry	`DMTFSpecMeasurementValueSize` + 3	Shall be the entry data of the DMTF measurement specification format.

965 ### 10.31.3 Example: Verifying Measurement Extension Log Against Hash-Extend Measurement

966 Figure 28 — Example for Measurement Extension Log illustrates an example of an MEL with 11 entries and two corresponding hash-extend measurements at `MEASUREMENTS` response indices 1 and 2 to which the log entries extend. The MEL in this example is constructed by the Responder during boot. The Responder implements a simple ROM–firmware secure boot architecture.

967 **Figure 28 — Measurement Extension Log Example**

968

SPDM MEASUREMENTS response

Index	DMTFMeasurementValueType	DMTFMeasurementValue Value
1	08h (HEM)	<digest-1>
2	08h (HEM)	<digest-2>

DMTF Measurement Extension Log MELEntries Field

“Entry” following Table“DMTF Measurement SpecificationFormat”

MEL Index	Meas Index	reserved	DMTFMeasurement ValueType	DMTFMeasurementValueSize	DMTFMeasurement Value
1	0	0	89h: raw bits; informational	3	“ROM”
2	1	0	00h: digest; ROM	48	<digest of ROM>
3	1	0	82h: raw bits; hardware config	128	<hardware config data of ROM>
4	0	0	89h: raw bits; informational	7	“Boot FW”
5	2	0	87h: raw bits; security version	8	0x0000000000000002
6	2	0	86h: raw bits; version	4	0x0100030A
7	2	0	01h: digest; firmware	48	<digest of boot firmware>
8	0	0	89h: raw bits; informational	9	“ROM patch”
9	1	0	00h: digest; ROM	48	<digest of ROM patch>
10	0	0	89h: raw bits; informational	14	“Application FW”
11	2	0	01h: digest; firmware	48	<digest of application firmware>

969 The MEL entries of indices 1, 4, 8, and 10 have a value type of `0x9` (informational). Since these are informational and do not apply to any measurement index, they are ignored in calculating HEM.

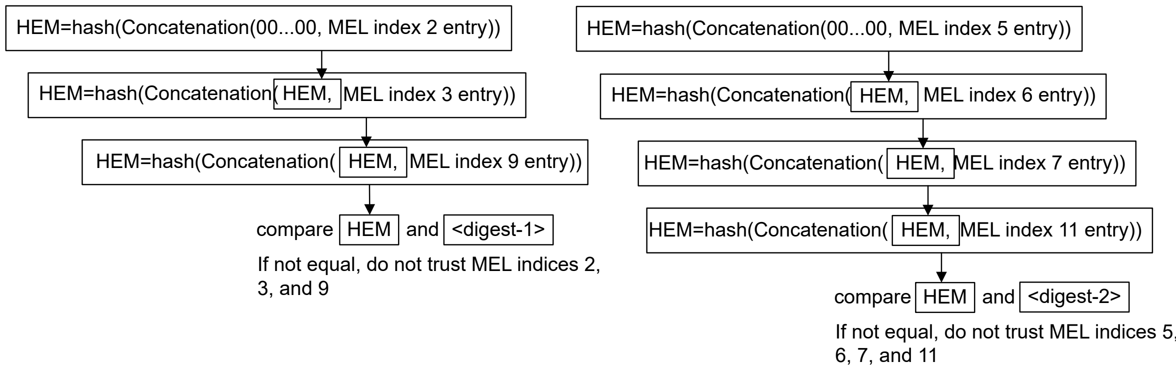
970 The hash-extend measurement at `MEASUREMENTS` index 1 is used for recording digests of ROM, patch, and hardware configuration. The MEL entries with MEL indices 2, 3, and 9 fit in this category and they extend to `MEASUREMENTS` index 1. Note that an extend operation shall consume the entire entry, including `MELIndex`, `MeasIndex`, `Reserved`, and `Entry`.

971 The hash-extend measurement at `MEASUREMENTS` index 2 is used for recording the digest of the firmware, firmware configuration, and version information. The MEL entries with MEL indices 5, 6, 7, and 11 fit in this category, and they extend to `MEASUREMENTS` index 2.

972 The Requester verifies the MEL entries by performing the checks illustrated in Figure 29 — Example for Verifying Measurement Extension Log Entries.

973 **Figure 29 — Example for Verifying Measurement Extension Log Entries**

974


HEM=hash(Concatenation(00...00, MEL index 2 entry))
HEM=hash(Concatenation(HEM, MEL index 3 entry))
HEM=hash(Concatenation(HEM, MEL index 9 entry))
compare HEM and <digest-1>
If not equal, do not trust MEL indices 2, 3, and 9
HEM=hash(Concatenation(00...00, MEL index 5 entry))
HEM=hash(Concatenation(HEM, MEL index 6 entry))
HEM=hash(Concatenation(HEM, MEL index 7 entry))
HEM=hash(Concatenation(HEM, MEL index 11 entry))
compare HEM and <digest-2>
If not equal, do not trust MEL indices 5, 6, 7, and 11

975 11 Session

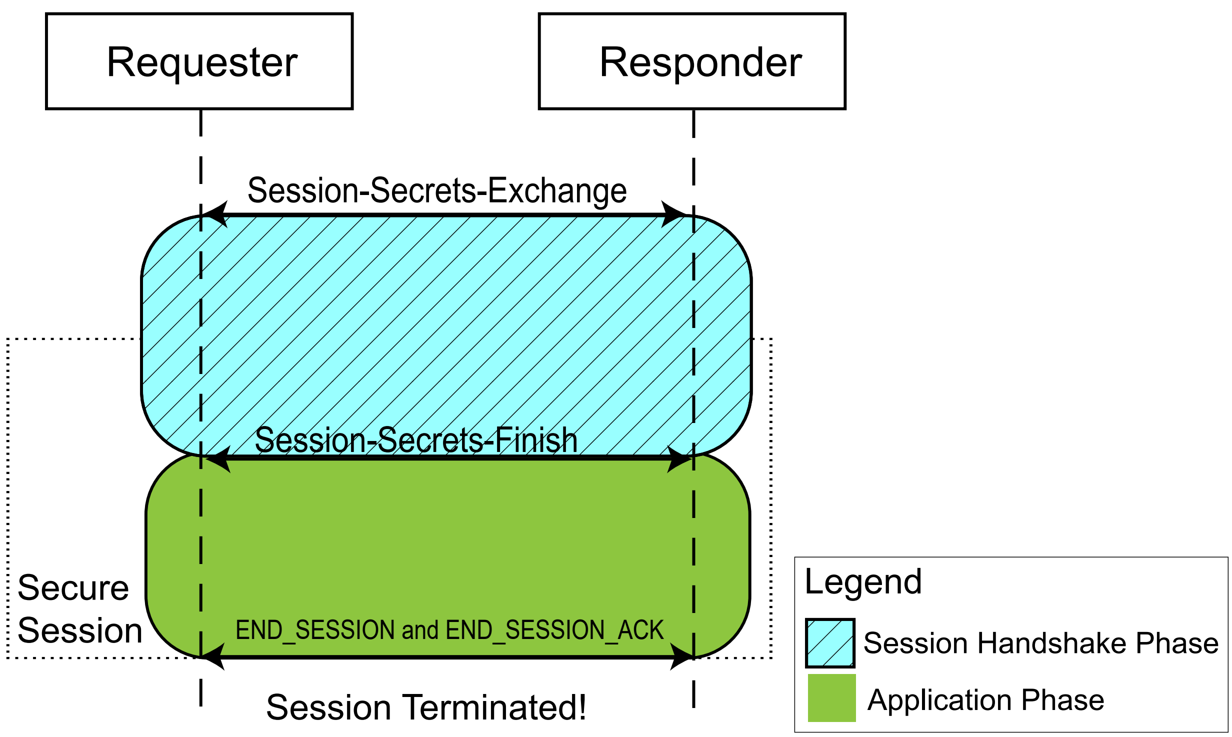
976 Sessions enable a Requester and Responder to have multiple channels of communication. More importantly, it enables a Requester and Responder to build a secure communication channel with cryptographic information that is bound ephemerally. Specifically, an SPDM session provides either encryption or message authentication or both.

977 A session has three phases, as Figure 30 — Session phases shows:

- The handshake
- The application
- Termination

978 **Figure 30 — Session phases**

979



980 11.1 Session handshake phase

981 The session handshake phase begins with either `KEY_EXCHANGE` or `PSK_EXCHANGE`. This phase also allows for the authentication of the Requester if the Responder indicated this earlier in its `ALGORITHMS` response. Furthermore, this phase of the session uses the handshake secrets to secure the communication as described in the Key schedule clause.

982 The purpose of this phase is to first build trust between the Responder and Requester before either side sends

application data. Additionally, it also ensures the integrity of the handshake and, to a certain degree, synchronicity with the derived handshake secrets.

983 In this phase of the session, `GET_ENCAPSULATED_REQUEST` and `DELIVER_ENCAPSULATED_RESPONSE` shall be used to obtain requests from the Responder to complete the authentication of the Requester, if the Responder indicated this in its `ALGORITHMS` response. During this phase, the Responder shall not asynchronously send requests to the Requester. The only requests allowed to be encapsulated shall be `GET_DIGESTS` and `GET_CERTIFICATE`. The Requester shall provide a signature in the `FINISH` request, as the FINISH request and FINISH_RSP response messages clause describes.

984 If an `ERROR` message of `ErrorCode=DecryptError` occurs in this phase, the session shall immediately terminate and proceed to session termination.

985 A successful handshake ends with either `FINISH_RSP` or `PSK_FINISH_RSP` and the application phase begins.

986 11.2 Application phase

987 Once the handshake completes and all validation passes, the session reaches the application phase where either the Responder or the Requester can send application data.

988 During this phase, a Requester can send SPDM messages such as `GET_MEASUREMENTS`. These messages might involve transcript calculations. If such calculations are required, they shall be calculated on a per session basis. Once a session has been established, subsequent messages sent outside of a session shall not contribute to the transcript within a session.

989 The application phase ends when the `HEARTBEAT` requirements fail, or with an `END_SESSION` message, or with an `ERROR` message of `ErrorCode=DecryptError`. The next phase is the session termination phase.

990 11.3 Session termination phase

991 This phase signals the end of the application phase and the enactment of internal clean-up procedures by the endpoints. Requesters and Responders can have various reasons for terminating a session, which are outside the scope of this specification.

992 SPDM provides the `END_SESSION` / `END_SESSION_ACK` message pair to explicitly trigger the session termination phase if needed but, depending on the transport, it might simply be an internal phase with no explicit SPDM messages sent or received.

993 When a session terminates, both Requester and Responder shall destroy or clean up all session secrets such as derived major secrets, DHE secrets and encryption keys. Endpoints might have other internal data associated with a session that they should also clean up.

994 11.4 Simultaneous active sessions

995 At least one session per connection shall be supported if both Requester and Responder advertise `KEY_EXCHANGE` or

`PSK_EXCHANGE` capabilities in this connection. If a `KEY_EXCHANGE` or `PSK_EXCHANGE` request would cause the Responder's number of simultaneous active sessions to exceed this maximum, the Responder shall respond with an `ERROR` message of `ErrorCode=SessionLimitExceeded`.

996 This specification does not prohibit concurrent sessions in which the same Requester and Responder reverse roles. For example, SPDM endpoint ABC, acting as a Requester, can establish a session to SPDM endpoint XYZ, which is acting as a Responder. At the same time, SPDM endpoint XYZ, now acting as a Requester, can establish a session to SPDM endpoint ABC, now acting as a Responder. Because these two sessions are distinct and separate, the two endpoints would ensure they do not mix sessions. To ensure proper session handling, each endpoint would ensure that their portion of the session IDs are unique at the time of Session-Secrets-Exchange. This would form a final unique session ID for that new session. Additionally, the endpoints can use information at the transport layer to further ensure proper handling of sessions.

997 11.5 Records and session ID

998 When the session starts, the communication of secured data is done using records. A record represents a chunk or unit of data that is either encrypted or authenticated or both. This data can be either an SPDM message or application data. Usually, the record contains the session ID resulting from one of the Session-Secrets-Exchange messages to aid both the Responder and Requester in binding the record to the respective derived session secrets.

999 The actual format and other details of a record are outside the scope of this specification. It is generally assumed that the transport protocol will define the format and other details of the record.

1000 12 Key schedule

1001 A key schedule describes how the various keys such as encryption keys used by a session are derived and when each key is used. The default SPDM key schedule makes heavy use of `HKDF-Extract` and `HKDF-Expand`, which RFC 5869 describes. SPDM defines this additional function:

```
BinConcat(Length, Version, Label, Context)
```

1002 where

- `BinConcat` shall be the concatenation of binary data in the order that Table 136 — BinConcat details shows:

1003 **Table 136 — BinConcat details**

Order	Data	Type	Endianness	Size
1	Length	Binary	Little	16 bits
2	Version	Text	Text	8 bytes
3	Label	Text	Text	Variable
4	Context	Binary	Hash byte order	`Hash` . `Length`

1004 If `Context` is `null`, `BinConcat` is the concatenation of the first three components only.

1005 Table 137 — Value of Version Text shows values of the 8-byte version text for different SPDM versions. Hexadecimal equivalents are shown in parentheses for clarity.

1006 **Table 137 — Value of Version Text**

SPDM version	byte 0	byte 1	byte 2	byte 3	byte 4	byte 5	byte 6	byte 7
SPDM 1.1	's' (`0x73`)	'p' (`0x70`)	'd' (`0x64`)	'm' (`0x6D`)	'1' (`0x31`)	'.' (`0x2E`)	'1' (`0x31`)	space (`0x20`)
SPDM 1.2	's' (`0x73`)	'p' (`0x70`)	'd' (`0x64`)	'm' (`0x6D`)	'1' (`0x31`)	'.' (`0x2E`)	'2' (`0x32`)	space (`0x20`)
SPDM 1.3	's' (`0x73`)	'p' (`0x70`)	'd' (`0x64`)	'm' (`0x6D`)	'1' (`0x31`)	'.' (`0x2E`)	'3' (`0x33`)	space (`0x20`)

1007 Note that the eighth byte of the version text is a space (`0x20`).

1008 The `HKDF-Expand` function prototype as used by the default SPDM key schedule is as follows:

```
HKDF-Expand(secret, context, Hash.Length)
```

1009 The `HKDF-Extract` function prototype is described as follows:

```
HKDF-Extract(salt, IKM);
```

1010 where

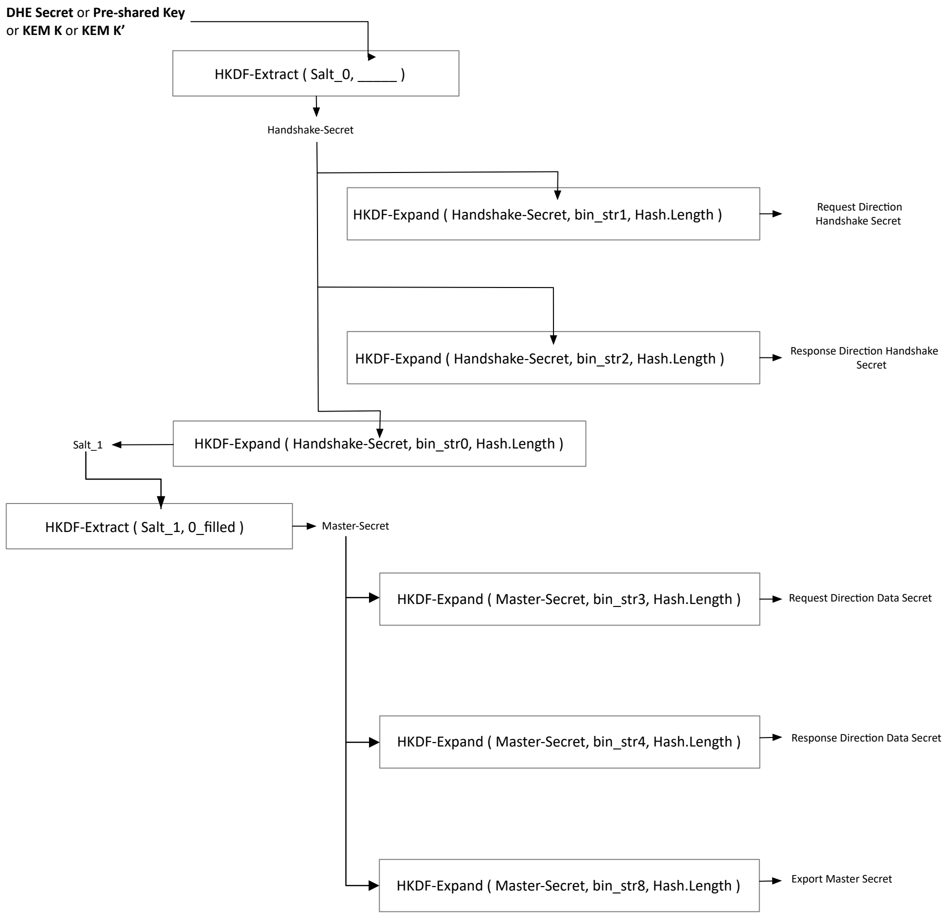
- IKM is the Input Keying Material.

1011 For `HKDF-Expand` and `HKDF-Extract`, the hash function shall be the selected hash function in the `ALGORITHMS` response. `Hash`.`Length` shall be the length of the output of the hash function selected by the `ALGORITHMS` response.

1012 Both Responder and Requester shall use the key schedule that Figure 31 — Key schedule shows.

1013 **Figure 31 — Key schedule**

1014



1015 In the figure, arrows going out of the box are outputs of that box. Arrows going into the box are inputs into the box and point to the specific input parameter they are used in. All boxes represent a single function producing a single output and are given names for clarity.

1016 Table 138 — Key schedule accompanies the figure to complete the key schedule. The Responder and Requester shall also adhere to the definition of this table.

1017 **Table 138 — Key schedule**

Variable	Definition	Value is secret?
Salt_0	A zero-filled array of `Hash`.`Length` length for `KEY_EXCHANGE` session. A 0xFF-filled array of `Hash`.`Length` length for `PSK_EXCHANGE` session.	No
Salt_1	Used to generate the Master-Secret.	Yes
0_filled	A zero-filled array of length `Hash`.`Length`.	No
bin_str0	`BinConcat(Hash.Length, Version, "derived", null)`	No

Variable	Definition	Value is secret?
bin_str1	`BinConcat(Hash.Length, Version, "req hs data", TH1)`	No
bin_str2	`BinConcat(Hash.Length, Version, "rsp hs data", TH1)`	No
bin_str3	`BinConcat(Hash.Length, Version, "req app data", TH2)`	No
bin_str4	`BinConcat(Hash.Length, Version, "rsp app data", TH2)`	No
DHE Secret	This shall be the secret derived from `KEY_EXCHANGE/KEY_EXCHANGE_RSP`.	Yes
KEM K	This shall be the shared secret key K output from the KEM's encapsulation function.	Yes
KEM K'	This shall be the shared secret key K' output from the KEM's decapsulation function.	Yes
Pre-Shared Key	PSK	Yes

1018 > **Note:** With common hash functions, any label longer than 12 characters requires an additional iteration of the hash function to compute. As in RFC 8446, the previously defined labels have all been chosen to fit within this limit.

1019 ## 12.1 DHE secret computation

1020 The DHE secret is a shared secret, and its computation is different per algorithm or algorithm class. These clauses define the format and computation for DHE algorithms.

1021 For `ffdhe2048`, `ffdhe3072`, `ffdhe4096`, `secp256r1`, `secp384r1`, and `secp521r1`, the format and computation of the DHE secret shall be the shared secret, which section 7.4 of RFC 8446 defines.

1022 For `SM2_P256`, the parameters of this curve are defined in the TCG Algorithm Registry. The DHE secret shall be K_A and K_B as defined in GB/T 32918.3-2016. The Requester shall compute K_A, and the Responder shall compute K_B to arrive at the same secret value. K_A and K_B are the results of a KDF. This specification shall use the KDF as defined by GB/T 32918.3-2016. The size of the DHE secret, referred to as `klen` in the KDF of GB/T 32918.3 specification, shall be the key size of the selected AEAD algorithm in `RespAlgStruct`. Lastly, GB/T 32918.3 allows for a flexible hash algorithm. The hash algorithm shall be the selected hash algorithm in `BaseHashSel` or `ExtHashSel`.

1023 ## 12.2 KEM K and K' computation

1024 If the Requester and Responder negotiated an ML-KEM scheme during algorithm negotiation, the Responder calculates the shared secret K by executing the encapsulation function defined in Algorithm 20 of the Module-Lattice-Based Key-Encapsulation Mechanism Standard; the Requester calculates the shared secret K' by executing the decapsulation function defined in Algorithm 21 of the Module-Lattice-Based Key-Encapsulation Mechanism Standard.

1025 Next, the two endpoints derive session keys from `K` and `K'` using the key schedule, respectively. Note that in rare cases, the `K'` output from the Requester's decapsulation function may not be equal to `K`. The Module-Lattice-Based Key Encapsulation Standard explicitly disallows the indicator of "`K` not equal `K'`" to be returned from the decapsulation function. If `K'` is not equal to `K`, the Requester and the Responder will calculate different session keys from the key schedule. As a result, the first message protected by a session key (that is, the `FINISH` request) will fail integrity verification at the receiver end. The Requester and the Responder should handle such rare failures due to mismatched `K` and `K'` the same way as other session message failures.

1026 12.3 Transcript hash in key derivation

1027 The key schedule uses two transcript hashes:

- **TH1**
- **TH2**

1028 12.4 TH1 definition

1029 If the Requester and Responder used `KEY_EXCHANGE` / `KEY_EXCHANGE_RSP` to exchange initial keying information, **TH1** shall be the output of applying the negotiated hash function to the concatenation of the following:

1. `VCA`
2. `[DIGESTS].*` (if issued and if `MULTI_KEY_CONN_RSP` is true).
3. Hash of the specified certificate chain in DER format (that is, `Param2` of `KEY_EXCHANGE`) or hash of the public key in its provisioned format, if a certificate is not used.
4. `[KEY_EXCHANGE]`.`*`
5. `[KEY_EXCHANGE_RSP]`.`*` except for the `ResponderVerifyData` field

1030 If the Requester and Responder used `PSK_EXCHANGE` / `PSK_EXCHANGE_RSP` to exchange initial keying information, **TH1** shall be the output of applying the negotiated hash function to the concatenation of the following:

1. `VCA`
2. `[PSK_EXCHANGE]`.`*`
3. `[PSK_EXCHANGE_RSP]`.`*` except for the `ResponderVerifyData` field

1031 12.5 TH2 definition

1032 If the Requester and Responder used `KEY_EXCHANGE` / `KEY_EXCHANGE_RSP` to exchange initial keying information, **TH2** shall be the output of applying the negotiated hash function to the concatenation of the following:

1. `VCA`
2. `[DIGESTS].*` (if issued and if `MULTI_KEY_CONN_RSP` is true).
3. Hash of the specified certificate chain in DER format (that is, `Param2` of `KEY_EXCHANGE`) or hash of the public key in its provisioned format, if a certificate is not used.

4. `[KEY_EXCHANGE]` . `*`
5. `[KEY_EXCHANGE_RSP]` . `*`
6. `[DIGESTS].*` (if encapsulated `DIGESTS` is issued and if `MULTI_KEY_CONN_REQ` is true).
7. Hash of the specified certificate chain in DER format (that is, `Param2` of `FINISH`) or hash of the public key in its provisioned format, if a certificate is not used. (Valid only in mutual authentication)
8. `[FINISH]` . `*`
9. `[FINISH_RSP]` . `*`

1033 If the Requester and Responder used `PSK_EXCHANGE` / `PSK_EXCHANGE_RSP` to exchange initial keying information, **TH2** shall be the output of applying the negotiated hash function to the concatenation of the following:

1. `VCA`
2. `[PSK_EXCHANGE]` . `*`
3. `[PSK_EXCHANGE_RSP]` . `*`
4. `[PSK_FINISH]` . `*` (if issued)
5. `[PSK_FINISH_RSP]` . `*` (if issued)

1034 12.6 Key schedule major secrets

1035 The key schedule produces four major secrets:

- Request-direction handshake secret (S_0)
- Response-direction handshake secret (S_1)
- Request-direction data secret (S_2)
- Response-direction data secret (S_3)

1036 Each secret applies in a certain direction of transmission and is only valid during a certain time frame. Each of these four major secrets will be used to derive their respective encryption keys and IV values to be used in the AEAD function as selected in the `ALGORITHMS` response.

1037 12.6.1 Request-direction handshake secret

1038 This secret shall only be used during the session handshake phase and shall be applied to all requests after `KEY_EXCHANGE` or `PSK_EXCHANGE` up to and including `FINISH` or `PSK_FINISH`.

1039 12.6.2 Response-direction handshake secret

1040 This secret shall only be used during the session handshake phase and shall be applied to all responses after `KEY_EXCHANGE_RSP` or `PSK_EXCHANGE_RSP` up to and including `FINISH_RSP` or `PSK_FINISH_RSP`.

1041 **12.6.3 Request-direction data secret**

1042 This secret shall be used for any data transmitted during the application phase of the session. This secret shall only be applied for all data traveling from the Requester to the Responder.

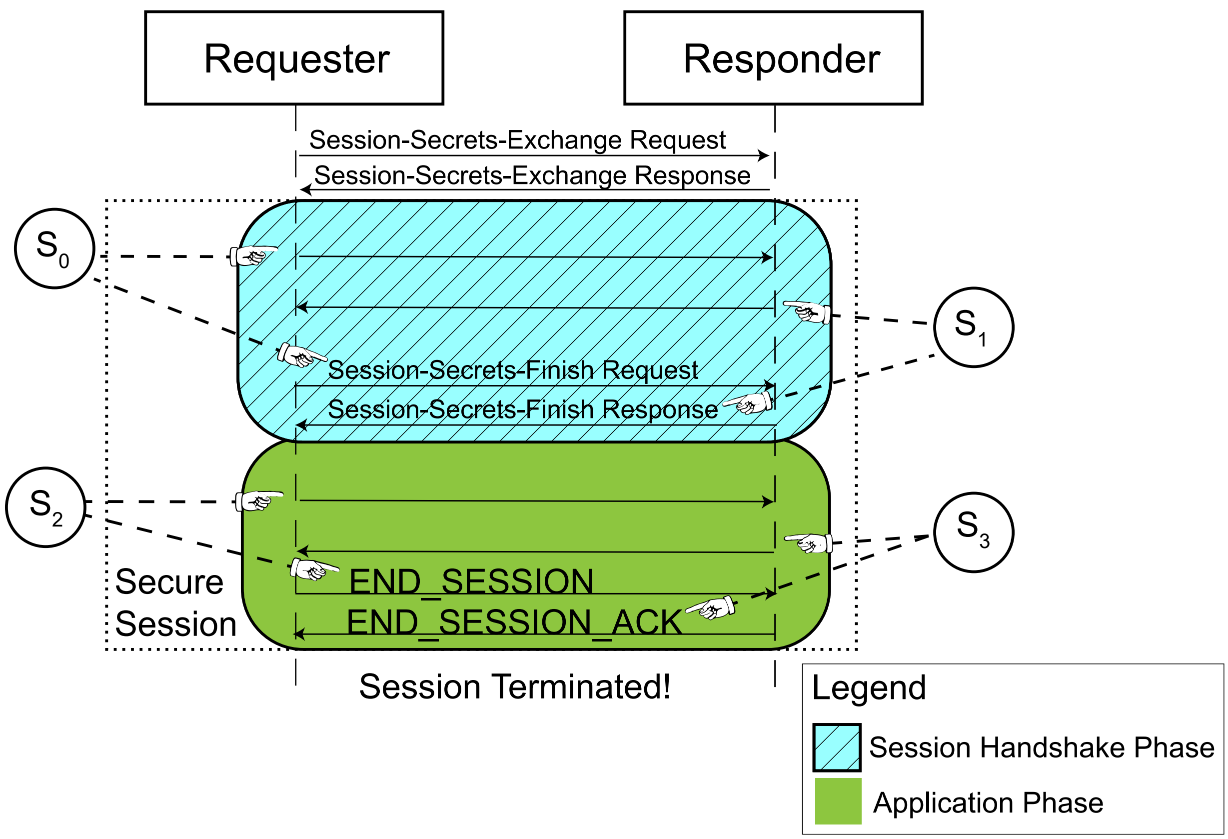
1043 **12.6.4 Response-direction data secret**

1044 This secret shall be used for any data transmitted during the application phase of the session. This secret shall only be applied for all data traveling from the Responder to the Requester.

1045 Figure 32 — Secrets usage illustrates where each of the major secrets are used, as described previously.

1046 **Figure 32 — Secrets usage**

1047



1048 ## 12.7 Encryption key and IV derivation

1049 For each key schedule major secret, the following function shall be applied to obtain the encryption key and IV value.

```
EncryptionKey = HKDF-Expand(major-secret, bin_str5, key_length);
IV = HKDF-Expand(major-secret, bin_str6, iv_length);

bin_str5 = BinConcat(key_length, Version, "key", null);
bin_str6 = BinConcat(iv_length, Version, "iv", null);
```

1050 Both `key_length` and `iv_length` shall be the lengths associated with the selected AEAD algorithm in the `ALGORITHMS` message.

1051

12.8 finished_key derivation

1052 This key shall be used to compute the `RequesterVerifyData` and `ResponderVerifyData` fields used in various SPDM messages. The key, `finished_key`, is defined as follows:

```
finished_key = HKDF-Expand(handshake-secret, bin_str7, Hash.Length);
bin_str7 = BinConcat(Hash.Length, Version, "finished", null);
```

1053 The handshake-secret shall be either a request-direction handshake secret or a response-direction handshake secret.

1054

12.9 Deriving additional keys from the Export Master Secret

1055 After a successful SPDM key exchange, additional keys can be derived from the Export Master Secret. How keys are derived from this secret is outside the scope of this specification. The Export Master Secret is not a major secret and is not updated through a major secrets update. How the Export Master Secret is updated, if required, is outside the scope of this specification.

```
Export Master Secret = HKDF-Expand(Master-Secret, bin_str8, Hash.Length);
bin_str8 = BinConcat(Hash.Length, Version, "exp master", TH2);
```

1056

12.10 Major secrets update

1057 The major secrets can be updated during an active session to avoid the overhead of closing down a session and recreating the session. This is achieved by issuing the `KEY_UPDATE` request.

1058 The major secrets shall be re-keyed as a result of this request. To compute the new secret for each new major data secret, the following algorithm shall be applied.

```
new_secret = HKDF-Expand(current_secret, bin_str9, Hash.Length);
bin_str9 = BinConcat(Hash.Length, Version, "traffic upd", null);
```

1059 In computing the new secret, `current_secret` shall be either the current Request-Direction Data Secret or the Response-Direction Data Secret. As a consequence of updating these secrets, new encryption keys and salts shall be derived from the new secrets and used immediately.

1060 # 13 Application data

1061 SPDM utilizes authenticated encryption with associated data (AEAD) cipher algorithms in much the same way that TLS 1.3 does to protect the confidentiality and integrity of data that shall remain secret as well as to protect the integrity of data that needs to be transmitted in the clear but shall still be protected from manipulation, as is the case for protocol headers. AEAD algorithms provide both encryption and message authentication. Each algorithm specifies details such as the size of the nonce, the position and length of the MAC, and many other factors to ensure a strong cryptographic algorithm.

1062 AEAD functions shall provide the following functions and comply with the requirements defined in RFC 5116:

```
AEAD_Encrypt(encryption_key, nonce, associated_data, plaintext);
AEAD_Decrypt(encryption_key, nonce, associated_data, ciphertext);
```

1063 where

- `AEAD_Encrypt` is the function that fully encrypts the `plaintext`, computes the MAC across both the `associated_data` and `plaintext`, and produces the `ciphertext`, which includes the MAC.
- `AEAD_Decrypt` is the function that verifies the MAC and, if validation is successful, fully decrypts the `ciphertext` and produces the original `plaintext`.
- `encryption_key` is the derived encryption key for the respective direction. See the Key schedule clause.
- `nonce` is the nonce computation. See the Nonce derivation clause.
- `associated_data` is the associated data.
- `plaintext` is the data to encrypt.
- `ciphertext` is the data to decrypt.

1064 ## 13.1 Nonce derivation

1065 Certain AEAD ciphers have specific requirements for nonce construction because their security properties can be compromised by the accidental reuse of a nonce value. Implementations should follow the requirements, such as those provided in RFC 5116 for nonce derivation.

1066 14 General opaque data format

1067 The general opaque data format allows for a variety of data defined by an assortment of vendors, standards bodies, and transport mechanisms to accompany an SPDM message without namespace collisions.

1068 If the `OpaqueDataFmt1` bit is selected in `OtherParamsSelection` of `ALGORITHMS`, then all opaque data fields in SPDM messages shall use the format that Table 139 — General opaque data format defines.

1069 **Table 139 — General opaque data format**

Byte offset	Field	Size (bytes)	Description
0	TotalElements	1	Shall be the total number of elements in `OpaqueList`.
1	Reserved	3	Reserved.
4	OpaqueList	Variable	Shall be a list of opaque elements. See Table 140 — Opaque element.

1070 Table 140 — Opaque element defines the format for each element in `OpaqueList`.

1071 **Table 140 — Opaque element**

Byte offset	Field	Size (bytes)	Description
0	ID	1	Shall be one of the values in the `ID` column of Table 65 — Registry or standards body ID.
1	VendorIDLen	1	Shall be the length in bytes of the `VendorID` field. If the data in `OpaqueElementData` belongs to a standards body, this field shall be 0. Otherwise, the data in `OpaqueElementData` belongs to the vendor and therefore, this field shall be the length indicated in the "Vendor ID length" column of Table 65 — Registry or standards body ID for the respective `ID`.
2	VendorID	`VendorIDLen`	If `VendorIDLen` is greater than zero, this field shall be the ID of the vendor corresponding to the `ID` field. Otherwise, this field shall be absent.

Byte offset	Field	Size (bytes)	Description
2 + `VendorIDLen`	OpaqueElementDataLen	2	Shall be the length of `OpaqueElementData`.
4 + `VendorIDLen`	OpaqueElementData	`OpaqueElementDataLen`	Shall be the data defined by the vendor or standards body.
4 + `VendorIDLen` + `OpaqueElementDataLen`	AlignPadding	`AlignPaddingSize` = 0, 1, 2, or 3	If 4 + `VendorIDLen` + `OpaqueElementDataLen` does not fall on a 4-byte boundary, this field shall be present and of the correct length to ensure that 4 + `VendorIDLen` + `OpaqueElementDataLen` + `AlignPaddingSize` is a multiple of 4. The value of this field shall be all zeros, and the size of this field shall be 0, 1, 2, or 3.

1072

15 Signature generation

1073 The `SPDMsign` function used in various part of this specification defines the signature generation algorithm while accounting for the differences in the various supported cryptographic signing algorithms in the `ALGORITHMS` message.

1074 The signature generation function takes this form:

```
signature = SPDMsign(PrivKey, data_to_be_signed, context);
```

1075 The `SPDMsign` function shall take these input parameters:

- `PrivKey` : a secret key
- `data_to_be_signed` : a bit stream of the data that will be signed
- `context` : a string

1076 The function shall output a signature using `PrivKey` and a selected cryptographic signing algorithm.

1077 The signing function shall follow these steps to create `spdm_prefix` and `spdm_context` (See Text or string encoding for encoding rules):

1. Create `spdm_prefix` . The `spdm_prefix` shall be the repetition, four times, of the concatenation of "dmtf-spdm-v", `SPDMversionString` and ".*". This will form a 64-character string.
2. Create `spdm_context` . If the Requester is generating the signature, `spdm_context` shall be the concatenation of "requester-" and `context` . If the Responder is generating the signature, the `spdm_context` shall be the concatenation of "responder-" and `context` .

1078 Now follows an example, designated Example 1, of creating a `combined_spdm_prefix` .

1079 The version of this specification for this example is 1.4.3, the Responder is generating a signature, and the `context` is "my example context". Thus, the `spdm_prefix` is "dmtf-spdm-v1.4.*dmtf-spdm-v1.4.*dmtf-spdm-v1.4.*dmtf-spdm-v1.4.*". The `spdm_context` is "responder-my example context".

1080 Next, the `combined_spdm_prefix` is formed. The `combined_spdm_prefix` shall be the concatenation of four elements: `spdm_prefix` , a byte with a value of zero, `zero_pad` , and `spdm_context` . The size of `zero_pad` shall be the number of bytes needed to ensure that the length of `combined_spdm_prefix` is 100 bytes. The size of `zero_pad` can be zero. The value of `zero_pad` shall be zero.

1081 Continuing Example 1, Table 141 — Combined SPDM prefix shows the `combined_spdm_prefix` with offsets. Offsets increase from left to right and top to bottom. As shown, the length of `combined_spdm_prefix` is 100 bytes. Hexadecimal equivalents are shown in parentheses for clarity. See Text or string encoding for encoding rules. Table 135 concludes Example 1.

1082 **Table 141 — Combined SPDM prefix**

Offset	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xA	0xB	0xC	0xD	0xE	0xF
0	'd' (`0x64`)	'm' (`0x6D`)	't' (`0x74`)	'f' (`0x66`)	'-' (`0x2D`)	's' (`0x73`)	'p' (`0x70`)	'd' (`0x64`)	'm' (`0x6D`)	'-' (`0x2D`)	'v' (`0x76`)	'1' (`0x31`)	'.' (`0x2E`)	'4' (`0x34`)	'.' (`0x2E`)	'*' (`0x2A`)
0x10	'd' (`0x64`)	'm' (`0x6D`)	't' (`0x74`)	'f' (`0x66`)	'-' (`0x2D`)	's' (`0x73`)	'p' (`0x70`)	'd' (`0x64`)	'm' (`0x6D`)	'-' (`0x2D`)	'v' (`0x76`)	'1' (`0x31`)	'.' (`0x2E`)	'4' (`0x34`)	'.' (`0x2E`)	'*' (`0x2A`)
0x20	'd' (`0x64`)	'm' (`0x6D`)	't' (`0x74`)	'f' (`0x66`)	'-' (`0x2D`)	's' (`0x73`)	'p' (`0x70`)	'd' (`0x64`)	'm' (`0x6D`)	'-' (`0x2D`)	'v' (`0x76`)	'1' (`0x31`)	'.' (`0x2E`)	'4' (`0x34`)	'.' (`0x2E`)	'*' (`0x2A`)
0x30	'd' (`0x64`)	'm' (`0x6D`)	't' (`0x74`)	'f' (`0x66`)	'-' (`0x2D`)	's' (`0x73`)	'p' (`0x70`)	'd' (`0x64`)	'm' (`0x6D`)	'-' (`0x2D`)	'v' (`0x76`)	'1' (`0x31`)	'.' (`0x2E`)	'4' (`0x34`)	'.' (`0x2E`)	'*' (`0x2A`)
0x40	0x0	0x0	0x0	0x0	0x0	0x0	0x0	0x0	'r' (`0x72`)	'e' (`0x65`)	's' (`0x73`)	'p' (`0x70`)	'o' (`0x6F`)	'n' (`0x6E`)	'd' (`0x64`)	'e' (`0x65`)
0x50	'r' (`0x72`)	'-' (`0x2D`)	'm' (`0x6D`)	'y' (`0x79`)	space (`0x20`)	'e' (`0x65`)	'x' (`0x78`)	'a' (`0x61`)	'm' (`0x6D`)	'p' (`0x70`)	'l' (`0x6C`)	'e' (`0x65`)	space (`0x20`)	'c' (`0x63`)	'o' (`0x6F`)	'n' (`0x6E`)
0x60	't' (`0x74`)	'e' (`0x65`)	'x' (`0x78`)	't' (`0x74`)												

1083 The next step is to form the `message_hash`. The `message_hash` shall be the hash of `data_to_be_signed` using the selected hash function in either `BaseHashSel` or `ExtHashSel`. Many hash algorithms allow implementations to compute an intermediate hash, sometimes called a running hash. An intermediate hash allows for the updating of the hash as each byte of the ordered data of the message becomes known. Consequently, the ability to compute an intermediate hash allows for memory utilization optimizations where an SPDM endpoint can discard bytes of the message that are already covered by the intermediate hash while waiting for more bytes of the message to be received.

1084 If the Responder is generating the signature, the selected cryptographic signing algorithm is indicated in `BaseAsymSel`, `ExtAsymSel`, or `PqcAsymAlgo` in the `ALGORITHMS` message. If the Requester is generating the signature, the selected cryptographic signing algorithm is indicated in `ReqBaseAsymAlg` or `ReqPqcAsymAlg` of `RespAlgStruct` in the `ALGORITHMS` message.

1085 Because each cryptographic signing algorithm is vastly different, these clauses define the binding of `SPDMsign` to those algorithms.

1086 15.1 Signing algorithms in extensions

1087 If an algorithm is selected in either the `ExtAsymSel` or `AlgExternal` of `ReqBaseAsymAlg` of `RespAlgStruct` in the `ALGORITHMS` response, its binding is outside the scope of this specification.

1088 15.2 RSA and ECDSA signing algorithms

1089 All RSA and ECDSA specifications do not define a specific hash function. Thus, the hash function to use shall be the hash function selected by the Responder in `BaseHashSel` or `ExtHashSel`.

1090 The private key, defined by the specification for these algorithms, shall be `PrivKey`.

1091 In the specification for these algorithms, the letter `M` denotes the message to be signed. `M` shall be the concatenation of `combined_spdm_prefix` and `message_hash`.

1092 RSA and ECDSA algorithms are described in Signature algorithm references.

1093 The FIPS PUB 186-5 supports deterministic ECDSA as a variant of ECDSA. RFC 6979 describes this deterministic digital signature generation procedure. This variant does not impact the signature verification process. How an implementation chooses to support ECDSA or deterministic ECDSA is outside the scope of this specification.

1094 15.3 EdDSA signing algorithms

1095 These algorithms are described in RFC 8032.

1096 The private key, defined by RFC 8032, shall be `PrivKey`.

1097 In the specification for these algorithms, the letter `M` denotes the message to be signed.

1098 15.3.1 Ed25519 sign

1099 This specification only defines Ed25519 usage and not its variants.

1100 `M` shall be the concatenation of `combined_spdm_prefix` and `message_hash`.

1101 15.3.2 Ed448 sign

1102 This specification only defines Ed448 usage and not its variants.

1103 `M` shall be the concatenation of `combined_spdm_prefix` and `message_hash`.

1104 Ed448 defines a context string, `C`. `C` shall be the `spdm_context`.

1105 15.4 SM2 signing algorithm

1106 This algorithm is described in GB/T 32918.2-2016. GB/T 32918.2-2016 also defines the variable `M` and ID_A.

1107 The private key defined by GB/T 32918.2-2016 shall be `PrivKey`.

1108 In the specification for SM2, the letter `M` denotes the message to be signed. `M` shall be the concatenation of `combined_spdm_prefix` and `message_hash`.

1109 The SM2 specification does not define a specific hash function. Thus, the hash function to use shall be the hash function selected by the Responder in `BaseHashSel` or `ExtHashSel`.

1110 Lastly, SM2 expects a distinguishing identifier, which identifies the signer and is indicated by the variable ID_A. If this algorithm is selected, the ID shall be an empty string of size 0.

1111 15.5 ML-DSA signing algorithm

1112 The ML-DSA standard defines a signature generation algorithm in Algorithm 2 (ML-DSA.Sign) and a pre-hash variant in Algorithm 4 (HashML-DSA.Sign). This specification uses Algorithm 2 of ML-DSA and not the pre-hash variant.

1113 The `M` input to ML-DSA.Sign shall be the concatenation of `combined_spdm_prefix` and `message_hash`. The ML-DSA standard does not define a specific hash function. Thus, the hash function to use shall be the hash function selected by the Responder in `BaseHashSel` or `ExtHashSel`. The `sk` input to ML-DSA.Sign shall be `PrivKey`. The `ctx` input to ML-DSA.Sign shall be `spdm_context`.

1114 The ML-DSA standard supports "hedged" and "deterministic" signing. Both variants use the same signature verification process. How an implementation chooses to support hedged or deterministic ML-DSA signing is outside the scope of this specification.

1115 15.6 SLH-DSA signing algorithm

1116 The SLH-DSA standard defines a signature generation algorithm in Algorithm 22 (slh_sign) and a pre-hash variant in Algorithm 23 (hash_slh_sign). This specification uses Algorithm 22 of SLH-DSA and not the pre-hash variant.

1117 The `M` input to slh_sign shall be the concatenation of `combined_spdm_prefix` and `message_hash`. The SLH-DSA standard does not define a specific hash function. Thus, the hash function to use shall be the hash function selected by the Responder in `BaseHashSel` or `ExtHashSel`. The `SK` input to slh_sign shall be `PrivKey`. The `ctx` input to slh_sign shall be `spdm_context`.

1118 The SLH-DSA standard supports "hedged" and "deterministic" signing. Both variants use the same signature verification process. How an implementation chooses to support hedged or deterministic SLH-DSA signing is outside the scope of this specification.

1119 15.7 Signature algorithm references

1120 These clauses provide basic information about each asymmetric algorithms SPDM supports, as Table 142 — SPDM Asymmetric Signature Reference Information shows. SPDM endpoints shall use the references in the **References** column for signature-related operations and the key size as indicated in the **Key Size** columns for the respective algorithm. The byte order for a signature when placing it into an SPDM signature field shall be signature byte order.

1121 **Table 142 — SPDM Asymmetric Signature Reference Information**

Algorithm Name	Private Key Size (bits)	References
TPM_ALG_RSASSA_2048	2048	Section 8.2 of IETF RFC 8017
TPM_ALG_RSASSA_3072	3072	Section 8.2 of IETF RFC 8017
TPM_ALG_RSASSA_4096	4096	Section 8.2 of IETF RFC 8017

Algorithm Name	Private Key Size (bits)	References
TPM_ALG_RSAPSS_2048	2048	Section 8.1 of IETF RFC 8017
TPM_ALG_RSAPSS_3072	3072	Section 8.1 of IETF RFC 8017
TPM_ALG_RSAPSS_4096	4096	Section 8.1 of IETF RFC 8017
TPM_ALG_ECDSA_ECC_NIST_P256	256	Section 6 of FIPS PUB 186-5 using `TPM_ECC_NIST_P256` curve parameters as TCG Algorithm Registry defines.
TPM_ALG_ECDSA_ECC_NIST_P384	384	Section 6 of FIPS PUB 186-5 using `TPM_ECC_NIST_P384` curve parameters as TCG Algorithm Registry defines.
TPM_ALG_ECDSA_ECC_NIST_P521	521	Section 6 of FIPS PUB 186-5 using `TPM_ECC_NIST_P521` curve parameters as TCG Algorithm Registry defines.
TPM_ALG_SM2_ECC_SM2_P256	256	Section 6 of GB/T 32918.2-2016 using `TPM_ECC_SM2_P256` curve parameters as TCG Algorithm Registry defines.
EdDSA ed25519	256	IETF RFC 8032
EdDSA ed448	456	IETF RFC 8032
ML-DSA-44	20480	Algorithm 2 and Algorithm 3 of ML-DSA
ML-DSA-65	32256	Algorithm 2 and Algorithm 3 of ML-DSA
ML-DSA-87	39168	Algorithm 2 and Algorithm 3 of ML-DSA
SLH-DSA-SHA2-128s	512	Algorithm 22 and Algorithm 24 of SLH-DSA
SLH-DSA-SHAKE-128s	512	Algorithm 22 and Algorithm 24 of SLH-DSA
SLH-DSA-SHA2-128f	512	Algorithm 22 and Algorithm 24 of SLH-DSA
SLH-DSA-SHAKE-128f	512	Algorithm 22 and Algorithm 24 of SLH-DSA

Algorithm Name	Private Key Size (bits)	References
SLH-DSA-SHA2-192s	768	Algorithm 22 and Algorithm 24 of SLH-DSA
SLH-DSA-SHAKE-192s	768	Algorithm 22 and Algorithm 24 of SLH-DSA
SLH-DSA-SHA2-192f	768	Algorithm 22 and Algorithm 24 of SLH-DSA
SLH-DSA-SHAKE-192f	768	Algorithm 22 and Algorithm 24 of SLH-DSA
SLH-DSA-SHA2-256s	1024	Algorithm 22 and Algorithm 24 of SLH-DSA
SLH-DSA-SHAKE-256s	1024	Algorithm 22 and Algorithm 24 of SLH-DSA
SLH-DSA-SHA2-256f	1024	Algorithm 22 and Algorithm 24 of SLH-DSA
SLH-DSA-SHAKE-256f	1024	Algorithm 22 and Algorithm 24 of SLH-DSA

1122 16 Signature verification

1123 The `SPDMsignatureVerify` function, used in various part of this specification, defines the signature verification algorithm while accounting for the differences in the various supported cryptographic signing algorithms in the `ALGORITHMS` message.

1124 The signature verification function takes this form:

```
SPDMsignatureVerify(PubKey, signature, unverified_data, context);
```

1125 The `SPDMsignatureVerify` function shall take these input parameters:

- `PubKey` : the public key
- `signature` : a digital signature
- `unverified_data` : a bit stream of data that needs to be verified
- `context` : a string

1126 The function shall verify the `unverified_data` using `signature` , `PubKey` , and a selected cryptographic signing algorithm. `SPDMsignatureVerify` shall return success if the signature verifies correctly and failure otherwise. Each cryptographic signing algorithm states the verification steps or criteria for successful verification.

1127 The verifier of the signature shall create `spdm_prefix` , `spdm_context` , and `combined_spdm_context` as described in Signature generation.

1128 The next step is to form the `unverified_message_hash` . The `unverified_message_hash` shall be the hash of `unverified_data` using the selected hash function in either `BaseHashSel` or `ExtHashSel` .

1129 If the Responder generated the signature, the selected cryptographic signature verification algorithm is indicated in `BaseAsymSel` , `ExtAsymSel` , or `PqcAsymSel` in the `ALGORITHMS` message. If the Requester generated the signature, the selected cryptographic signature verification algorithm is indicated in `ReqBaseAsymAlg` or `ReqPqcAsymAlg` of `RespAlgStruct` in the `ALGORITHMS` message.

1130 Because each cryptographic signature verification algorithm is vastly different, these clauses define the binding of `SPDMsignatureVerify` to those algorithms.

1131 16.1 Signature verification algorithms in extensions

1132 If an algorithm is selected in either the `ExtAsymSel` or `AlgExternal` of `ReqBaseAsymAlg` of `RespAlgStruct` in the `ALGORITHMS` response, its binding is outside the scope of this specification.

1133 16.2 RSA and ECDSA signature verification algorithms

1134 All RSA and ECDSA specifications do not define a specific hash function. Thus, the hash function to use shall be the hash function selected by the Responder in `BaseHashSel` or `ExtHashSel`.

1135 The public key, defined in the specification for these algorithms, shall be `PubKey`.

1136 In the specification for these algorithms, the letter `M` denotes the message that is signed. `M` shall be concatenation of the `combined_spdm_prefix` and `unverified_message_hash`.

1137 For RSA algorithms, `SPDMsignatureVerify` shall return success when the output of the signature verification operation, as defined in the RSA specification, is "valid signature". Otherwise, `SPDMsignatureVerify` shall return a failure.

1138 For ECDSA algorithms, `SPDMsignatureVerify` shall return success when the output of "ECDSA Signature Verification Algorithm" as defined in FIPS PUB 186-5 is `"accept"`. Otherwise, `SPDMsignatureVerify` shall return failure.

1139 RSA and ECDSA algorithms are described in Signature algorithm references.

1140 16.3 EdDSA signature verification algorithms

1141 RFC 8032 describes these algorithms. RFC 8032, also, defines the `M`, `PH`, and `C` variables.

1142 The public key, also defined in RFC 8032, shall be `PubKey`.

1143 In the specification for these algorithms, the letter `M` denotes the message to be signed.

1144 16.3.1 Ed25519 verify

1145 `M` shall be the concatenation of `combined_spdm_prefix` and `unverified_message_hash`.

1146 `SPDMsignatureVerify` shall return success when step 1 does not result in an invalid signature and when the constraints of the group equation in step 3 are met as described in RFC 8032 section 5.1.7. Otherwise, `SPDMsignatureVerify` shall return failure.

1147 16.3.2 Ed448 verify

1148 `M` shall be the concatenation of `combined_spdm_prefix` and `unverified_message_hash`.

1149 Ed448 defines a context string, `C`. `C` shall be the `spdm_context`.

1150 `SPDMsignatureVerify` shall return success when step 1 does not result in an invalid signature and when the constraints of the group equation in step 3 are met as described in RFC 8032 section 5.2.7. Otherwise, `SPDMsignatureVerify` shall return failure.

1151 16.4 SM2 signature verification algorithm

1152 This algorithm is described in GB/T 32918.2-2016, which also defines the variable `M` and ID_A.

1153 The public key, also defined in GB/T 32918.2-2016, shall be `PubKey`.

1154 In the specification for SM2, the variable `M'` is used to denote the message that is signed. `M'` shall be the concatenation of `combined_spdm_prefix` and `unverified_message_hash`.

1155 The SM2 specification does not define a specific hash function. Thus, the hash function to use shall be the hash function selected by the Responder in `BaseHashSel` or `ExtHashSel`.

1156 Lastly, SM2 expects a distinguishing identifier, which identifies the signer, and is indicated by the variable ID_A. See SM2 signing algorithm to create the value for ID_A.

1157 `SPDMsignatureVerify` shall return success when the Digital signature verification algorithm, as described in GB/T 32918.2-2016, outputs an "accept". Otherwise, `SPDMsignatureVerify` shall return failure.

1158
17 ML-DSA signature verification algorithm

1159 The ML-DSA standard defines a signature verification algorithm in Algorithm 3 (ML-DSA.Verify) and a pre-hash variant in Algorithm 5 (HashML-DSA.Verify). This specification uses Algorithm 3 of ML-DSA and not the pre-hash variant.

1160 The `M` input to ML-DSA.Verify shall be the concatenation of `combined_spdm_prefix` and `unverified_message_hash`. The ML-DSA standard does not define a specific hash function. Thus, the hash function to use shall be the hash function selected by the Responder in `BaseHashSel` or `ExtHashSel`. The `pk` input to ML-DSA.Verify shall be `PubKey`. The `ctx` input to ML-DSA.Verify shall be `spdm_context`.

1161 `SPDMsignatureVerify` shall return success when ML-DSA.Verify returns TRUE. Otherwise, `SPDMsignatureVerify` shall return failure.

1162 # 18 SLH-DSA signature verification algorithm

1163 The SLH-DSA standard defines a signature verification algorithm in Algorithm 24 (slh_verify) and a pre-hash variant in Algorithm 25 (hash_slh_verify). This specification uses Algorithm 24 of SLH-DSA and not the pre-hash variant.

1164 The `M` input to slh_verify shall be the concatenation of `combined_spdm_prefix` and `unverified_message_hash`. The SLH-DSA standard does not define a specific hash function. Thus, the hash function to use shall be the hash function selected by the Responder in `BaseHashSel` or `ExtHashSel`. The `PK` input to slh_verify shall be `PubKey`. The `ctx` input to slh_verify shall be `spdm_context`.

1165 `SPDMsignatureVerify` shall return success when slh_verify returns TRUE. Otherwise, `SPDMsignatureVerify` shall return failure.

1166 19 General ordering rules

1167 These general ordering rules apply to SPDM messages that form a transcript that eventually gets signed.

1168 Out-of-order requests shall nullify the transcript.

1169 A Requester can retry messages. The retries shall be identical to the first message, excluding transport variances. If the Responder sees two or more non-identical `NEGOTIATE_ALGORITHMS`, the Responder shall either return an `ERROR` message of `ErrorCode=UnexpectedRequest` or silently discard non-identical messages. Because a retried message is identical to the first, a retried message shall not be used in transcript hash calculations.

1170 If a Requester wants to retrieve a `CAPABILITIES` response with the Supported Algorithms included, the Requester should first issue `GET_CAPABILITIES` with Bit 1 in `Param1` set to 1. If the Responder does not support the Supported Algorithms block in its `CAPABILITIES` response, it responds with an `ERROR` response. At this point, the Requester can issue a second `GET_CAPABILITIES` with Bit 1 in `Param1` cleared to 0. In this case, the second request is not considered a retry, and both requests and their corresponding responses are used in transcript hash calculations. After a successful `CAPABILITIES` response, if the Responder sees two or more non-identical `GET_CAPABILITIES` requests, the Responder shall either return an `ERROR` message of `ErrorCode=UnexpectedRequest` or silently discard non-identical messages.

1171 For `CHALLENGE` and Session-Secrets-Exchange, the Responder should ensure it can distinguish between the respective retry and the respective original message. Failure to distinguish correctly might lead to an authentication failure, session handshake failures, and other failures. The response to a retried request should be identical to the original response.

1172 When a transcript hash includes the `DIGESTS` response, the first `DIGESTS` that immediately follows the VCA shall be the `DIGESTS` response that is used for the remainder of the SPDM connection. If the Requester does not send a `GET_DIGESTS` immediately following the VCA, then the `DIGESTS` shall no longer be part of the transcript for the remainder of the SPDM connection even if the Requester sends the request later. Similarly, for mutual authentication in a multi key session, if the first encapsulated response is a `DIGESTS` response in the session handshake phase, then that encapsulated `DIGESTS` shall be included in the transcript hash for the corresponding session. If the first encapsulated response is not a `DIGESTS` response from the Requester in mutual authentication, then no encapsulated `DIGESTS` response shall be part of the transcript hash for the corresponding session. Furthermore, the aforementioned rules do not apply to `M1` or `M2` ordering rules.

1173 20 DMTF event types

1174 The DMTF-defined event types are sent using the Event mechanism. DMTF-defined event types shall have an `ID` of 0 and `VendorIDLen` of 0 populated in the `EventGroupId` field in Event data table. If other DMTF DSP wants to generate an event, those DMTF DSP shall use an `ID` of DMTF-DSP accordingly.

1175 The DMTF event types table shows the supported DMTF event types for the DMTF event group. The values in the **Event Type ID** column shall be the same values for `EventTypeId` field in Event data table for the DMTF event group for the corresponding event in the **Event Name** column. The version (`EventGroupVer`) of the DMTF Event Group shall be `1` .

1176 **Table 143 — DMTF event types table**

Event Type ID	Event Name	Requirement	Description
0	Reserved	Reserved	Reserved.
1	EventLost	Mandatory	Events were lost.
2	MeasurementChanged	Optional	One or more measurements changed.
3	MeasurementPreUpdate	Optional	A pending update will change one or more measurements. However, the update has not yet taken effect.
4	CertificateChanged	Optional	Information in one or more certificate slots has changed. This could be the certificate or the associated key.
All others	Reserved	Reserved	Reserved.

1177 20.1 Event type details

1178 Each DMTF event type has its own event-specific information, referred to as `EventDetail` , to describe the event. These clauses describe the format for each DMTF event type. The event types are listed in the DMTF event types table.

1179 20.1.1 Event Lost

1180 This event (`EventTypeId=EventLost`) shall notify the Event Recipient that one or more events were lost. The reasons for event loss are varied and numerous, but one example is loss due to insufficient resources. This event should be retried until the Event Recipient acknowledges it. Retrying this event means that this event was not acknowledged previously.

1181 The Event lost format table describes the format for `EventDetail`.

1182 **Table 144 — Event lost format**

Offset	Field	Size (bytes)	Description
0	LastAckedEventInstID	4	Shall be the last event instance ID acknowledged by the Event Recipient.
4	LastLostEventInstID	4	Shall be the last lost event instance ID.

1183 The range of lost events shall be the range from (`LastAckedEventInstID` + 1) to `LastLostEventInstID` inclusive.

1184 If the Event Notifier cannot or can no longer track the information in Event lost format, then `LastAckedEventInstID` and `LastLostEventInstID` shall both be 0xFFFF_FFFF.

1185 When resending an "event lost" event, the Event Notifier can update fields in Event lost format if new events are lost since the last time the "event lost" event was sent.

1186 20.1.2 Measurement changed event

1187 The measurement changed event (`EventTypeId=MeasurementChanged`) shall notify the Event Recipient when one or more measurement blocks have changed. The `EventDetail` format for this event type shall be as Measurement changed event details format defines.

1188 Table 145 — Measurement changed event details format describes the format for `EventDetail` for the `MeasurementChanged` event.

1189 **Table 145 — Measurement changed event details format**

Offset	Field	Size (bytes)	Description
0	ChangedMeasurements	32	This field is a bit mask where each bit indicates changes to its corresponding measurement index. Specifically, the bit at bit offset X shall be set to indicate a change to the Measurement block at measurement index X. At least one bit in this field shall be set. Bits 0 and 255 shall be reserved.

1190 The Event Recipient can issue `GET_MEASUREMENTS` to obtain further details on the change.

1191 20.1.3 Measurement pre-update event

1192 The measurement pre-update event (`EventTypeId=MeasurementPreUpdate`) notifies the Event Recipient when one or more Measurement blocks will change due to a pending update. The `EventDetail` format for this event type shall be as Measurement pre-update event details format defines.

1193 Table 146 — Measurement pre-update event details format describes the format for `EventDetail` for the `MeasurementPreUpdate` event.

1194 **Table 146 — Measurement pre-update event details format**

Offset	Field	Size (bytes)	Description
0	PreUpdateMeasurementChanges	32	This field is a bit mask where each bit indicates pending changes to the corresponding measurement index in an update scenario such as a firmware update or pending configuration change. Specifically, the bit at bit offset X shall be set to indicate a potential change to the Measurement block at measurement index X as a result of an update. At least one bit in this field shall be set. Bits 0 and 255 shall be reserved.

1195 Upon receiving the `MeasurementPreUpdate` event, the Event Recipient may send GET_MEASUREMENTS with the `NewMeasurementRequested` option (see Table 54 — GET_MEASUREMENTS request attributes) to acquire and evaluate the Event Notifier's pending new measurements. If the Event Recipient deems the Event Notifier's new measurements unacceptable, the Event Recipient may terminate the session.

1196 The pre-update notification mechanism does not allow the Event Recipient to stop the Event Notifier from applying the update. However, an Event Notifier that has sent `MeasurementPreUpdate` to an Event Recipient should not apply the update until at least one of the following events happens for each Event Recipient:

- Arrival of EVENT_ACK from the Event Recipient
- Arrival of END_SESSION from the Event Recipient
- Event Recipient timeout (duration defined by implementation)

1197 20.1.4 Certificate changed event

1198 The certificate changed event (`EventTypeId=CertificateChanged`) shall notify the Event Recipient when data associated with one or more fields in the `DIGESTS` response have changed. The `EventDetail` format for this event type shall be the Certificate changed event details format.

1199 Table 147 — Certificate changed event details format table describes the format for `EventDetail` for the `CertificateChanged` event.

1200 **Table 147 — Certificate changed event details format**

Offset	Field	Size (bytes)	Description
0	CertificateChanged	1	This field is a bit mask where each bit indicates certificate related changes to the corresponding certificate slot. Specifically, the bit at bit offset X shall be set to indicate a change to data associated with one or more fields in `DIGESTS` for certificate slot X. At least one bit in this field shall be set.

1201 The Event Recipient can issue `GET_DIGESTS` or `GET_CERTIFICATE` to obtain further details on the change.

1202

21 ANNEX A (informative) TLS 1.3

1203 This specification heavily models TLS 1.3. TLS 1.3, and consequently this specification, assumes the transport layers provide the following capabilities or attributes:

- Reliability in transmission and reception of data.
- Transmission of data is either in order or the order of data can be reconstructed at reception.

1204 While not all transports are created equal, if a transport cannot meet these capabilities, adoption of SPDM is still possible. In these transports, this specification recommends The Datagram Transport Layer Security (DTLS) Protocol Version 1.3.

1205 22 ANNEX B (informative) Device certificate example

1206 Device certificate example shows an example device certificate:

1207 **Device certificate example**

```
-----
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 8 (0x8)
        Signature Algorithm: ecdsa-with-SHA256
        Issuer: C = CA, ST = NC, L = city, O = ACME, OU = ACME Devices, CN = CA
        Validity
            Not Before: Jan  1 00:00:00 1970 GMT
            Not After : Dec 31 23:59:59 9999 GMT
        Subject: C = US, ST = NC, O = ACME Widget Manufacturing, OU = ACME Widget Manufacturing Unit, CN
= w0123456789
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                RSA Public-Key: (2048 bit)
                Modulus:
                    00:ba:67:47:72:78:da:28:81:d9:81:9b:db:88:03:
                    e1:10:a4:91:b8:48:ed:6b:70:3c:ec:a2:68:a9:3b:
                    5f:78:fc:ae:4a:d1:1c:63:76:54:a8:40:31:26:7f:
                    ff:3e:e0:bf:95:5c:4a:b4:6f:11:56:ca:c8:11:53:
                    23:e1:1d:a2:7a:a5:f0:22:d8:b2:fb:43:da:dd:bd:
                    52:6b:e6:a5:3f:0f:3b:60:b8:74:db:56:08:d9:ee:
                    a0:30:4a:03:21:1e:ee:60:ad:e4:00:7a:6e:6b:32:
                    1c:28:7e:9c:e8:c3:54:db:63:fd:1f:d1:46:20:9e:
                    ef:80:88:00:5f:25:db:cf:43:46:c6:1f:50:19:7f:
                    98:23:84:38:88:47:5d:51:8e:11:62:6f:0f:28:77:
                    a7:20:0e:f3:74:27:82:70:a7:96:5b:1b:bb:10:e7:
                    95:62:f5:37:4b:ba:20:4e:3c:c9:18:b2:cd:4b:58:
                    70:ab:a2:bc:f6:2f:ed:2f:48:92:be:5a:cc:5c:5e:
                    a8:ea:9d:60:e8:f8:85:7d:c0:0d:2f:6a:08:74:d1:
                    2f:e8:5e:3d:b7:35:a6:1d:d2:a6:04:99:d3:90:43:
                    66:35:e1:74:10:a8:97:3b:49:05:51:61:07:c6:08:
                    01:1c:dc:a8:5f:9e:30:97:a8:18:6c:f9:b1:2c:56:
                    e8:67
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            X509v3 Basic Constraints:
                CA:FALSE
            X509v3 Key Usage:
                Digital Signature, Non Repudiation, Key Encipherment
            X509v3 Subject Alternative Name:
                othername: 1.3.6.1.4.1.412.274.1::ACME:WIDGET:0123456789
    Signature Algorithm: ecdsa-with-SHA256
```

```
    Signature Value:
        30:45:02:20:1e:5a:a6:ed:5c:b6:2b:f5:9e:22:28:9c:ef:c7:
        aa:db:1c:87:83:48:c1:50:cb:25:04:ab:c9:6e:7c:f5:6b:01:
        02:21:00:da:48:d4:49:a5:65:5c:2c:83:fc:05:00:66:48:98:
        f8:f0:cb:63:b7:2e:87:db:c8:63:58:6c:21:91:7a:68:95
-----BEGIN CERTIFICATE-----
MIIC4jCCAoigAwIBAgIBCDAKBggqhkjOPQQDAjBcMQswCQYDVQQGEwJDQTELMAkG
A1UECAwCTkMxDTALBgNVBAcMBGNpdHkxDTALBgNVBAoMBEFDTUUxFTATBgNVBAsM
DEFDTUUgRGV2aWNlczELMAkGA1UEAwwCQ0EwIBcNNzAwMTAxMDAwMDAwWhgPOTk5
OTEyMzEyMzU5NTlaMH0xCzAJBgNVBAYTAlVTMQswCQYDVQQIDAJOQzEiMCAGA1UE
CgwZQUNNRSBXaWRnZXQgTWFudWZhY3R1cmluZzEnMCUGA1UECwweQUNNRSBXaWRn
ZXQgTWFudWZhY3R1cmluZyBVbml0MRQwEgYDVQQDDAt3MDEyMzQ1Njc4OTCCASIw
DQYJKoZIhvcNAQEBBQADggEPADCCAQoCggEBALpnR3J42iiB2YGb24gD4RCkkbhI
7WtwPOyiaKk7X3j8rkrRHGN2VKhAMSZ//z7gv5VcSrRvEVbKyBFTI+Edonql8CLY
svtD2t29UmvmpT8PO2C4dNtWCNnuoDBKAyEe7mCt5AB6bmsyHCh+nOjDVNtj/R/R
RiCe74CIAF8l289DRsYfUBl/mCOEOIhHXVGOEWJvDyh3pyAO83QngnCnllsbuxDn
lWL1N0u6IE48yRiyzUtYcKuivPYv7S9Ikr5azFxeqOqdYOj4hX3ADS9qCHTRL+he
Pbc1ph3SpgSZ05BDZjXhdBColztJBVFhB8YIARzcqF+eMJeoGGz5sSxW6GcCAwEA
AaNNMEswCQYDVR0TBAIwADALBgNVHQ8EBAMCBeAwMQYDVR0RBCowKKAmBgorBgEE
AYMcghIBoBgMFkFDTUU6V0lER0VUOjAxMjM0NTY3ODkwCgYIKoZIzj0EAwIDSAAw
RQIgHlqm7Vy2K/WeIiic78eq2xyHg0jBUMslBKvJbnz1awECIQDaSNRJpWVcLIP8
BQBmSJj48Mtjty6H28hjWGwhkXpolQ==
-----END CERTIFICATE-----
```

1208 # 23 ANNEX C (informative) OID reference

1209 Table 148 — Object identifiers (OIDs) lists all object identifiers (OIDs) that this specification defines:

1210 **Table 148 — Object identifiers (OIDs)**

OID	Identifier	Definition	Use
{ 1 3 6 1 4 1 412 }	id-DMTF	DMTF OID	Enterprise ID for DMTF
{ id-DMTF 274 }	id-DMTF-spdm	SPDM OID	Base OID for all SPDM OIDs
{ id-DMTF-spdm 1 }	id-DMTF-device-info	SPDM certificate requirements and recommendations	Certificate device information.
{ id-DMTF-spdm 2 }	id-DMTF-hardware-identity	Identity provisioning	Hardware certificate identifier.
{ id-DMTF-spdm 3 }	id-DMTF-eku-responder-auth	Extended Key Usage authentication OIDs	Certificate Extended Key Usage - SPDM Responder Authentication.
{ id-DMTF-spdm 4 }	id-DMTF-eku-requester-auth	Extended Key Usage authentication OIDs	Certificate Extended Key Usage - SPDM Requester Authentication.
{ id-DMTF-spdm 5 }	id-DMTF-mutable-certificate	Identity provisioning	Mutable certificate identifier.
{ id-DMTF-spdm 6 }	id-DMTF-SPDM-extension	SPDM Non-Critical Certificate OID	To contain other OIDs in a certificate extension.

1211 # 24 ANNEX D (informative) variable name reference

1212 Throughout this document, various sizes and offsets are referred to by a variable. Table 149 — Variables lists variables used in this document, the definition of the variable, and the location in this document that shows how the variable is set.

1213 **Table 149 — Variables**

Symbol	Definition	Set location
A	Number of Requester-supported extended asymmetric key signature algorithms.	Table 15 — NEGOTIATE_ALGORITHMS request message format
A'	Number of extended asymmetric key signature algorithms selected by the Requester.	Table 23 — Successful ALGORITHMS response message format
D	The size of D (and C for ECDHE) that is derived from the selected DHE group.	See the `KEY_EXCHANGE` request message format in Table 74 — KEY_EXCHANGE request message format.
E	Number of Requester-supported extended hashing algorithms.	Table 15 — NEGOTIATE_ALGORITHMS request message format
E'	The number of Requester-supported extended hashing algorithms selected by the Responder.	Table 23 — Successful ALGORITHMS response message format
Lx where x is a number	A generic variable used to indicate the sizes of a field. The x is a number starting with zero. An example of Lx is `L0` , `L1` and so forth. The scope of this variable is always local to the table that uses it. For example, `L0` often appears in more than one table but there is no relationship between an `L0` in one table and an `L0` in another table.	Numerous tables
H	The output size, in bytes, of the hash algorithm agreed upon in `NEGOTIATE_ALGORITHMS` .	Table 23 — Successful ALGORITHMS response message format
HEM	Hash-extend measurement.	Hash-extend measurements clause.
MS	The length of the cryptographic hash or raw bit stream, as indicated in `DMTFSpecMeasurementValueType[7]` .	Table 58 — DMTF measurement specification format
MSHLength	The length of the MeasurementSummaryHash field in the `CHALLENGE_AUTH` , `KEY_EXCHANGE_RSP` , and `PSK_EXCHANGE_RSP` messages.	Table 49 — Successful CHALLENGE_AUTH response message format

Symbol	Definition	Set location
NL	The length of the Nonce field in the `GET_MEASUREMENTS` request and the `MEASUREMENTS` response.	GET_MEASUREMENTS request attributes
n	Number of version entries in the `VERSION` response message.	Table 9 — Successful VERSION response message format
Q	Length of the ResponderContext.	Table 80 — PSK_EXCHANGE_RSP response message format
P	Length of the `PSKHint`.	Table 79 — PSK_EXCHANGE request message format
R	Length of the `RequesterContext`.	Table 79 — PSK_EXCHANGE request message format
`SigLen`	The size of the asymmetric-signing algorithm output, in bytes, that the Responder selected in the last `ALGORITHMS` response message.	Table 23 — Successful ALGORITHMS response message format
SL	The length of the `SlotIDParam` field in the `GET_MEASUREMENTS` request.	Table 49 — GET_MEASUREMENTS request message format

1214 # 25 ANNEX E (informative) change log

1215 ## 25.1 Version 1.0.0 (2019-10-16)

- Initial Release

1216 ## 25.2 Version 1.1.0 (2020-07-15)

- Minor typographical fixes
- USB Authentication Specification 1.0 link updated
- Tables are no longer numbered. They are now named.
- Fix internal document links in SPDM response codes table.
- Added sentence to paragraph 97 to clarify on the potential to skip messages after a reset.
- Removed text at paragraph 181.
- `Subject Alternative Name otherName` field in Optional fields references DMTF OID section.
- `DMTFOtherName` definition changed to properly meet ASN.1 syntax.
- Text in figures is now searchable.
- Corrected example of a leaf certificate in Annex A.
- Minor edits to figures for clarity.
- Clarified that transcript hash could include hash of the raw public key if a certificate is not used.
- New:
  - Added Session support.
    - Added SPDM request and response messages to support initiating, maintaining and terminating a secure session.
    - Added Key schedule for session secrets derivation.
    - Added Application Data to provide overview of how data is encrypted and authenticated in a session.
  - Introduce new terms and definitions.
  - Added Measurement Manifest to `DMTFSpecMeasurementValueType`.
  - Added mutual authentication.
  - Added Encapsulated request flow to support master-slave types of transports.

1217 ## 25.3 Version 1.2.0 (2021-11-01)

- Clarified SPDM version selection after receiving `VERSION` Response with error handling for certain scenarios.
- Fix improper reference in `DMTFSpecMeasurementValue` field in "Measurement field format when MeasurementSpecification field is Bit 0 = DMTF" table.

- Certificate digests in `DIGESTS` calculation clarified.
- Format of certificate in `CertChain` parameter of `CERTIFICATE` message clarified.
- Validity period of X.509 v3 certificate clarified in Required Fields
- Remove `InvalidSession` error code.
- Clarified transport responsibilities in `PSK_EXCHANGE` and `PSK_EXCHANGE_RSP` .
- Clarified the usage of `MutAuthRequested` field in `KEY_EXCHANGE_RSP` .
- Added recommendation of PSK usage when an SPDM endpoint can be a Requester and Responder.
- Added recommendation for usage of `RequesterContext` in PSK scenarios.
- Clarified capabilities for Requester and Responder in `GET_CAPABILITIES` and `CAPABILITIES` messages.
- Clarified timing requirements for encapsulated requests.
- Clarified out of order and retries
- Clarified error handling actions when unexpected requests occur during various mutual authentication flows.
- Refer to slot number fields as `SlotID` and normalize `SlotID` fields to 4 bits where possible.
- Changed `PSK_FINISH` and `FINISH` changes in Table 6 — SPDM request and response messages validity.
- Clarified `HANDSHAKE_IN_THE_CLEAR_CAP` usage in `PSK_EXCHANGE` .
- Change `SPDMVersion` field in every request and response message, except `GET_VERSION` / `VERSION` messages, to point to a central location in this specification where it explains the appropriate value to populate for this field.
- Clarified use case for `Token` field in `ResponseNotReady` .
- Clarified the format of the certificate chain used in the Transcript hash calculation in Transcript hash calculation rules.
- Renamed `Measurement field format when MeasurementSpecification field is Bit 0 = DMTF` table to Table 45 — DMTF measurement specification format.
- Clarified the `ENCAP_CAP` field in the capabilities of the Requester and Responder.
- Renamed Mutual Authentication in `KEY_EXCHANGE` to Session-based mutual authentication.
- `ERROR` responses are no longer required in most error scenarios.
- Clarify the definition of backward-compatible changes in Version encoding.
- Enhanced requirements for when a firmware update occurred on a Responder in GET_VERSION request and VERSION response messages.
- Clarified error code `ResponseNotReady` for M1/M2 and L1/L2 computation.
- Clarified byte order for ASN.1 encoded data, hashes and digests.
- Requester should not use PSK_EXCHANGE if CHALLENGE_AUTH and/or MEASUREMENTS with signature was received from Responder.
- Required `GET_VERSION` , `VERSION` , `GET_CAPABILITIES` , `CAPABILITIES` , `NEGOTIATE_ALGORITHMS` , and `ALGORITHMS` in transcript even if negotiated state is supported.
- Enhanced signature generation and verification with a prefix to mitigate signature misuse attacks.
- Clarified behavior of `END_SESSION` with respect to Negotiated State when there are multiple active sessions.
- Added new defined term `Reset` to mean device reset. Updated use of the word reset for M1/M2, L1/L2.
- Clarified that a Measurement Manifest should support both hash and raw bit stream formats.

- Clarified Measurement Summary Hash construction rules.
- Clarified minimum timing for HEARTBEAT request and HEARTBEAT_ACK response messages to be sufficiently greater than `T1`. Removed command-specific guidance on retry timing.
- Table codification changed to be consistent with DMTF template.
- New:
  - Added support for `AliasCert`s.
    - Compliant Requesters must support a Responder that uses either `DeviceCert`s or `AliasCert`s.
  - Added Certain error handling in encapsulated flows
  - Added Slot 0 certificate-provisioning methodology.
  - Added Allowance for encapsulated requests.
  - Allowed `GET_CERTIFICATE` followed by `CHALLENGE` flow after a reset in `M1` and `M2` message transcript.
  - Added new features for `GET_MEASUREMENTS` and `MEASUREMENTS`:
    - More measurement value types.
    - Allow Requester to request hash or raw bit stream for measurement from the Responder.
  - Added Advice.
  - Added structured representation of device mode Device mode field of a measurement block.
  - Added Text or string encoding.
  - Signature Clarification:
    - Added Signature generation and Signature verification for clarity and interoperability.
    - Change `Sign` and `Verify` abstract function to `SPDMsign` and `SPDMsignatureVerify` respectively.
  - Added General ordering rules and references to it, to describe additional requirements for the various transcript and message transcripts.
  - Added additional clause for checking `FINISH`.`Param2` if handshake is in the clear.
  - Added OIDs to represent:
    - Hardware certificate identifier (Identity provisioning)
    - Certificate Extended Key Usage - SPDM Responder Authentication (Extended Key Usage authentication OIDs)
    - Certificate Extended Key Usage - SPDM Requester Authentication (Extended Key Usage authentication OIDs)
    - Mutable certificate identifier (Identity provisioning)
  - Added SM2 to Base Asymmetric Algorithms and Key Exchange Protocols.
  - Added SM3 to Base Hash Algorithms and Measurement Hash Algorithms.
  - Added SM4 to AEAD Algorithms.
  - Changed symbol "S" denoting signature size to "SigLen" throughout document.
  - Removed potentially confusing mention of "mutual authentication" in `PSK_EXCHANGE` section.
  - Add method to transfer large SPDM messages. See Large SPDM message transfer mechanism.
  - Changed Measurement Summary Hash concatenation function inputs.
  - Clarified requirements for compliant certificate chains.
  - Tables and figures are now numbered. Though these numbers might change in future versions of

specification, the titles will remain the same.
    - Allowed Requester to specify session termination policy when Responder completes firmware or configuration update.

1218 ## 25.4 Version 1.3.0 (2023-04-05)

- Change attribution for this standard from the Platform Management Communications Infrastructure (PMCI) Working Group to the Security Protocols and Data Models Working Group.
- Fix minor typographical errors.
- Clarified `CSRdata` requirements.
- Correct indication that Identity Provisioning OIDs are in the certificate Extended Key Usage, and add SPDM Non-Critical Certificate Extension OID to Table 43 — Optional fields.
- Added Signature Algorithm References clauses to clarify basic information about asymmetric algorithms.
- Clarified `Offset` and `Length` fields in `GET_CERTIFICATE` message.
- Clarified measurement specification related fields in `NEGOTIATE_ALGORITHMS`, `ALGORITHMS` and Table 52 — Measurement block format.
- Added recommended `ErrorCode` for the case when the Responder detects overlapping `SET_CERTIFICATE` commands.
- Clarified `DataTransferSize` and `MaxSPDMmsgSize` in `GET_CAPABILITIES` and `CAPABILITIES` messages.
- Updated General ordering rules to include discussion of the `CAPABILITIES` response with the Support Algorithms block.
- Allow the sender to utilize the Large SPDM message transfer mechanism when the transmit buffer size of the sender is less than the `DataTransferSize` of the receiving SPDM endpoint.
- Clarified that `ENCRYPT_CAP` and `MAC_CAP` apply to all phases of a secure session.
- Clarified the relationship between `MAC_CAP` and `ResponderVerifyData` or `RequesterVerifyData` in Session-Secret-Exchange and Session-Secret-Finish messages.
- Provide more description for `HANDSHAKE_IN_THE_CLEAR_CAP` in `GET_CAPABILITIES` and `CAPABILITIES` messages.
- Added `VERSION` to the chunking forbidden list.
- Added definition of opaque data.
- Make the layout of tables 62 and 63 consistent with other tables.
- Clarified DER encoding for 'RequesterInfo'
- Added more guidance to `RawBitStreamRequested` in `GET_MEASUREMENTS` request.
- Changed ANNEX B from "normative" to "informative".
- Corrected Requester to Responder in Table 71 Successful KEY_EXCHANGE_RSP response message format.
- Correct values in Field and Size columns of Table 61
- Changed the message validity of VENDOR_DEFINED_REQUEST and VENDOR_DEFINED_RESPONSE to “Vendor-defined”.
- Clarified measurement method for various timing parameters in Timing specification table.
- Corrected the signing algorithm in the FINISH request's Signature field.

- Correct Figure 1 — SPDM certificate chain models to show AliasCert model.
- Clarify how retried messages affect transcript hash in General ordering rules.
- Update Table 7 — Timing specification for SPDM messages to clarify that Responders can exceed `ST1` and `CT` using `ErrorCode=ResponseNotReady` .
- Clarified rules around when the old key can be discarded during `KEY_UPDATE` .
- Updated link and information for IETF DTLS 1.3.
- Clarified that `AlgCount` field in Algorithm request and response structures shall be a value of 2.
- Edit Figure 22 so that a Secure Session does not encompass Session-Secrets-Exchange.
- Clarified measurement signing capabilities in `SignatureRequested` field of `GET_MEASUREMENTS` .
- Clarified retries from the perspective Responder and Requester in Timing requirements.
- Changed "or" to "and" in Large SPDM message transfer mechanism section.
- Clarified that `MeasurementHashAlgo` should be zero if `MeasurementSpecificationSel` is zero.
- Remove "in general" from normative text.
- Clarified that the use of `BaseAsymAlgo` in the `NEGOTIATE_ALGORITHMS` request is dependent on the capabilities of the Responder.
- Removed directive to save the public key of the leaf certificate retrieved through the `GET_CERTIFICATE` request.
- Added trusted environment to glossary.
- Clarified how the value of `MinDataTransferSize` is calculated.
- Added `LargeResponse` error to description of chunking certificates.
- Clarified that if endpoint does not support chunking then it must set `MaxSPDMmsgSize` equal to `DataTransferSize` .
- Clarified effects on out-of-order message on the transcript and other clarifications in General ordering rules.
- Clarified the definition of Session-Secret-Exchange and removed the duplicate definition of it.
- Replaced wording of "internal buffer" in `GET_CERTIFICATE` with `DataTransferSize` and "transmit buffer".
- Specify the hashing algorithm for `MeasurementSummaryHash` in multiple tables.
- Added normative statement that `VERSION` entries should be unique.
- Clarified conditions for `LargeResponse` error.
- Clarified `CERTIFICATE` response when the `Length` field of `GET_CERTIFICATE` is zero.
- Clarified the assumption that version entries are not duplicated when calculating `MinDataTransferSize` .
- Introduced `Context` field in `CHALLENGE` and `GET_MEASUREMENTS` requests.
- Clarified restrictions on Bit 0 through 2 of the `MutAuthRequested` field of `KEY_EXCHANGE_RSP` .
- Separated nonce and non-repeating counter in `PSK_EXCHANGE` and `PSK_EXCHANGE_RSP` .
- Added definitions for *sequentially decreasing*, *sequentially increasing*, and *monotonically increasing*.
- Clarified updating keys in `KEY_UPDATE` .
- Added size of the transmit buffer as a condition for `CHUNK_SEND` .
- Clarified measurement support in the `MeasurementHashAlgo` field of the `ALGORITHMS` response.
- Clarified conditions under which `CERT_CAP` must be `0b` .
- Allowed `GET_DIGESTS` and `GET_CERTIFICATE` in session.
- Clarified that extended algorithms are external to this specification.

- Changed "should" to "shall" in the `LargeMessageSize` field of `CHUNK_SEND`.
- Clarified (A1, B4, C1) message flow is permitted.
- Required root certificate to always be included in `SET_CERTIFICATE`.
- Changed "cancel" to "invalidate state and data associated with" in `GET_VERSION` and `VERSION` response messages.
- Removed non-normative text from the `Length` field of `GET_CERTIFICATE`.
- Changed link to VCA from acronym to definition in the "transcript computation rules for M1/M2" table.
- Clarified Session-Secrets-Exchange in Optimized encapsulated request flow
- Clarified the `Request ID` for the first message in an optimized encapsulated request flow.
- Clarified the presence of the `SlotIDParam` field in `GET_MEASUREMENTS`.
- Removed informative statement that chunks are equal in size.
- Clarified that SPDM messages sent outside of a session do not contribute to in-session transcripts.
- Fixed typo in table 88.
- Deprecated the `CHAL_CAP` capability of the Requester.
- Clarified value of `HANDSHAKE_IN_THE_CLEAR_CAP` when using Pre-Shared Keys.
- Removed "after Reset" from M1/M2 ordering.
- Clarify that Integers are unsigned.
- Clarified requirements for chunking the `CERTIFICATE` response.
- Clarified relevant capabilities in BaseAsymAlgo, BaseHashAlgo.
- Clarified that Export Master Secret does not get updated with `KEY_UPDATE`.
- Removed the "full" modifier in front of `MeasurementRecord` in the `MEASUREMENTS` response table.
- Fixed typos and removed redundant grammar in Table 50.
- Fixed OID value for id-DMTF-device-info to match earlier releases.
- Clarified definition of DecryptError.
- Clarified that endpoints must ensure proper ordering and existence of messages when calculating transcripts hashes.
- Fixed typo in table 90.
- Move `DMTFSpecMeasurementValueType[6:0]` to its own table to improve readability.
- Changed instances of `Concatenation()` to the defined `Concatenate()` operator.
- Clarified slots 1-7 certificate provisioning.
- Removed normative text that prohibited reuse of session IDs.
- Clarified that non-encapsulated requests are prohibited during the session handshake phase.
- Removed potentially confusing statements on Slot provisioning for `GET_CSR`.
- Removed normative error statement from the `BasicMutAuthReq` field of `CHALLENGE_AUTH`.
- Clarified exclusion of signature in CHALLENGE_AUTH and usage of concatenation in Table 47
- Clarified that the Negotiated State Preservation Indicator applies to the cached Negotiated State.
- Clarified CSR signing.
- Removed encapsulation requirements from MUT_AUTH_CAP definition.

- Removed deprecation status from ENCAP_CAP.
- Clarified that a provisioned public key can be used to generate the Transcript for KEY_EXCHANGE_RSP HMAC.
- Clarified use of `DataTransferSize` and `MaxSPDMmsgSize` in GET_CAPABILITIES request and CAPABILITIES response messages.
- Fixed typo in table 52.
- Replaced links to ITU-T X.509 with RFC5280 and removed ITU-T X.509 from the Normative references section.
- Moved general text for transcript calculations from "Transcript and transcript hash calculation rules" to the "SPDM messaging protocol" section.
- Clarified that `KEY_EX_CAP` only applies to Requester's request message and Responder's response message.
- Clarified that if either Requester or Responder do not support Heartbeat then the value of `HeartbeatPeriod` would be `0`.
- Renamed "VendorLen" to "VendorIDLen".
- Used different Salt_0 value for PSK session in key schedule.
- Corrected `PK` to `PubKey` in CHALLENGE_AUTH signature verification.
- Removed quotation mark of VCA in L1/L2 definition.
- Clarified which portions of a certificate chain in the Alias certificate model is immutable.
- Updated link and version to ISO/IEC Directives, Part 2.
- Fixed size of MeasurementSummaryHash field to include 0 as a possible size value when the field is absent.
- Renamed the `HMAC-Hash` to `HKDF-Extract`.
- Moved message and field notation to Notations.
- Clarified `VCA` for the case where capabilities and algorithms are provisioned alongside PSK.
- Clarified that `ProvisionedSlotMask` in the `CHALLENGE_AUTH` response is dependent on the negotiated algorithms.
- Clarified runtime measurement change detection.
- Removed "between devices" in the introduction of SPDM.
- Used different `Salt_0` value for PSK session in key schedule.
- Removed the restriction to set `Length` to be `0xFFFF` in `GET_CERTIFICATE` if both endpoints support the large SPDM message transfer mechanism.
- Clarified `RequesterContext` in PSK_EXCHANGE.
- The Responder now always returns error ResponseTooLarge and no longer silently discards the request that caused this error.
- Clarified certificate chain validation in Figure 8.
- Clarified that a `GET_VERSION` request can also cancel a pending request at the responder in section about Requirement for Requesters.
- Restructure the Identity provisioning clause. Split the existing content into multiple clauses to help organization and incorporate the Generic certificate model. Make the use of Device Certificate and Alias Certificate consistent rather than using the terms `DeviceCert` and `AliasCert` to refer to specific certificates.
- Add missing `ffdhe3072` in DHE secret computation section.
- Clarified that the Requester should not use `PSK_EXCHANGE` after receiving any Responder-signed response

messages.

- Clarified that SPDM certificates are still compliant to the requirements of RFC 5280.
- Clarified field requirements for SPDM certificates and clarified that RFC 5280 defines the certificate format.
- Clarify allowed session phases for GET_CSR, SET_CERTIFICATE, GET_DIGESTS, and GET_CERTIFICATE in Table 6 — SPDM request and response messages validity.
- Clarified RESPOND_IF_READY request validity.
- Clarified that erroneous GET_VERSION shall not affect connection state, not session state.
- Clarify the device behavior when a reset is required for a pending previous `SET_CERTIFICATE` request.
- Clarified sessions can be established one at a time when `HANDSHAKE_IN_THE_CLEAR_CAP` is set.
- New:
  - Added Signature byte order and Octet string byte order clauses.
  - Add the Manifest format for a measurement block to define a measurement manifest header format that leverages the `SVH` format.
  - Added `SET_CERT_CAP` , `CSR_CAP` and `CERT_INSTALL_RESET_CAP` capabilities bits.
  - Add a section to discuss differences in cryptographic and non-cryptographic Timing parameters.
  - Added option in `SET_CERTIFICATE` to delete existing certificate chain from slot.
  - Add a `SlotSizeRequested` request attribute to the GET_CERTIFICATE request and CERTIFICATE response messages.
  - Added the IANA CBOR registry and VESA standards body to Registry or standards body ID.
  - Added a tracking tag in GET_CSR request and CSR response messages for use after a reset.
  - Added missing `MaxSPDMmsgSize` to `GET_CAPABILITIES` request and `CAPABILITIES` response messages.
  - Add an `Overwrite` bit to the `GET_CSR` request.
  - Added requirements on population of Slot 0 in Certificates and certificate chains.
  - Added GET_ENDPOINT_INFO request and ENDPOINT_INFO response messages.
  - Added the `InvalidPolicy` error code.
  - Added Supported algorithms block to Successful CAPABILITIES response message format.
  - Added column to table 132 that specifies whether values are secret or not.
  - Added new request `GET_MEASUREMENT_EXTENSION_LOG` and response `MEASUREMENT_EXTENSION_LOG` , measurement extension log formats, and examples.
  - Added new "hash-extended" measurement type.
  - Added Multiple asymmetric key support.
  - Added Generic certificate model.
  - Added Notification overview and Event Mechanism
  - Added DMTF event types
  - Added Custom environments clauses.
  - Added `NewMeasurementRequested` in GET_MEASUREMENTS.
  - Add missing `ffdhe3072` in DHE secret computation section.
  - Change FIPS PUB 186-4 reference to FIPS PUB 186-5.

- Defined the data models for the first four bytes of `VendorDefinedReqPayload` and `VendorDefinedRespPayload` when standards body is DMTF.
- Added normative information in Table 13 — Flag fields definitions for the Requester and Table 14 — Flag fields definitions for the Responder.

1219 ## 25.5 Version 1.4.0 (pending)

- Introduced post-quantum cryptography (PQC) support, including digital signature and key encapsulation for session establishment.
- Clarified that `ERROR` is only allowed in response to `GET_VERSION` in cases explicitly defined in this specification.
- Changed instances of "Requester-direction" to "Request-direction" and instances of "Responder-direction" to "Response-direction".
- Expanded `ResponseTooLarge` to include the transmit buffer size and `DataTransferSize`.
- Added clause that sizes and lengths are in units of bytes.
- Clarified that `LargeResponse` shall not re-initialize L1/L2 to null.
- Clarified relationship between `PUB_KEY_ID_CAP` and `KeyPairID`.
- Changed statement about graceful error handling during chunking from normative to non-normative.
- Added evaluation of the Responder's transmit buffer size to `LargeResponse`.
- Clarify that the key size listed in Table 136 — SPDM Asymmetric Signature Reference Information is the private key size.
- Removed non-normative text for measurement changed event.
- Removed "If present" from mandatory fields in Field requirements.
- Clarified that start of the Heartbeat timer can include `PSK_EXCHANGE_RSP`.
- Clarified that WT_{Max} includes the wait time from the most recently received `ResponseNotReady`.
- Changed endianness of the Context field of `BinConcat` from little to hash byte order.
- Stated that Event Recipient timeout is defined by the implementation.
- Renamed instances of OEM to vendor-defined.
- Clarified that the `CSRTrackingTag` is managed by the Responder and is set to `0` on a new `GET_CSR` request.
- Clarified that key pairs in `Param2` of `DIGESTS` are consistent with negotiated algorithms.
- Clarified in Table 59 — Registry or standards body ID that the registry specifies the value used for the `VendorID` field.
- Clarified that Event Notifier needs to wait for all Event Recipients before updating measurement.
- Clarified that `MULTI_KEY_CAP` is not allowed when `PUB_KEY_ID_CAP` is enabled.
- Clarified that DMTF event types must be supported by an Event Notifier.
- Clarify the definition of errata versions and that they may contain behavioral changes to fix security issues or defects.
- Corrected `DMTFSpecMeasurementValueType[6:0]` to 0x06 for mutable firmware in Table 54 — DMTFSpecMeasurementValueType values.
- Clarified error condition for `EventAllPolicy`.

- Clarified requirements for `AssocCertSlotMask` field in `GET_KEY_PAIR_INFO` response.
- Clarified that a valid certificate slot can only be used for identity authentication.
- Specified Responder's response to invalid measurement index.
- Clarified that `CSRCertModel` and `SetCertModel` cannot be `0` when `MULTI_KEY_CONN_RSP` is true.
- Corrected error in the MEL specification field format table by changing Requester to Responder.
- Clarified Table 63 — Standards body or vendor-defined header (SVH) when the payload is standards body or registry organization defined.
- Move statement that DMTF does not define extended algorithms to Table 31 — Extended Algorithm field format.
- Clarified that a `CSRTrackingTag` of `0` indicates a new `GET_CSR` request, and associated behavior.
- Clarified that size of `ResponseToLargeRequest` may be smaller than `DataTransferSize`.
- Clarified that Responders can alter requested CSR fields.
- Clarified that the value of `SetCertModel` is conditional on the existence of a corresponding `CSRCertModel`.
- Clarified that authorization for key information only applies to changing it and not retrieving it.
- Figures in SPDM bits-to-bytes-mapping updated.
- Clarify `SET_CERTIFICATE` `Erase` behavior when the slot is empty and that the associated key is not erased in the operation.
- Renamed "Negotiated State Preservation Indicator" field of END_SESSION request to "Negotiated State Clearing Indicator".
- Clarified that hardware identity is recommended for device and alias certificate models. Add a definition of hardware identity.
- Added explanation as to how the `RDT` value is measured at the Responder.
- Add SVH to the table of abbreviated terms.
- Increased size of `ChunkSeqNo` from 2 bytes to 4 bytes.
- Clarify that `CERTIFICATE` response shall not return a partial certificate chain in case of chunking enabled and the Requester asking for a complete certificate chain.
- Remove text that says `ENCRYPT_CAP` and `MAC_CAP` apply to all phases of a secure session.
- Made normative text for vendor-defined transcripts more general.
- Clarified the value of `KeyPairID` when `MULTI_KEY_CAP` is not set.
- Correct the references to Table 5 — SPDM response codes in Table 102 — KEY_PAIR_INFO response message format and Table 107 — SET_KEY_PAIR_INFO_ACK response message format.
- Clarified that the minimum number of supported sessions shall be one per connection.
- Clarified error handling scenarios in `SET_CERTIFICATE` request.
- Clarified that both Requester and Responder need to support bits in `OtherParamsSelection` for them to be selected by Responder.
- Correct `ReqLength` to `RespLength` in Table 66 — VENDOR_DEFINED_RESPONSE response message format.
- Clarified the chunked transfer including transcript update and interruption of the chunk transfer sequence.
- Clarified the definition of RDT as the additional time needed by the responder and not as a delay.
- Removed `Overwrite` condition from the `CSRCertModel` field in `GET_CSR`.
- Adjusted the section numbering for Certificates and certificate chains

- Added caption for Table 56
- Clarified that only one endpoint needs to set `HANDSHAKE_IN_THE_CLEAR_CAP` to `0` for the handshake to be secure.
- Removed normative statement about `RTT` due to redundancy with later normative statements.
- Clarified the requirements for `TotalKeyPairs` .
- Updated the link to the measurement block in Param2 in the `GET_MEASUREMENTS` request message
- Clarified Responder's support for retry.
- Fixed wrong side of Certificate owner (Responder shall be Requester and vise versa) in Param2 of FINISH Request.
- Clarified that `SlotID` field value in `SET_CERTIFICATE_RSP` Response shall be the same as `SlotID` value in correspondent `SET_CERTIFICATE` Request.
- Clarified changing parameter rules with `SET_KEY_PAIR_INFO` on key pair ID with generated key pair.
- Stated that all figures are informative unless otherwise specified explicitly.
- Clarify the offset of `Context` in `GET_MEASUREMENTS` request message format.
- Clarify that the `KeyExUse` bit mask is used for `FINISH` message.
- Added new ID (DMTF-DSP) in Registry or standards body ID.
- Clarified ID to use in DMTF event types.
- Clarify `CertModel` value if `MULTI_KEY_CAP` is not set
- Clarify which `DIGESTS` response is included in a transcript hash.
- Clarify Param2 (slot mask) of the CHALLENGE_AUTH response message.
- Clarify that `MeasurementSpecificationSel` shall be set when the Responder supports either MEL or MEASUREMENTS.
- Clarify the value of version text in Key Schedule.
- Reformatted Table 130 — Value of Version Text and Table 134 — Combined SPDM prefix to improve readability.
- Clarified that `HEARTBEAT` shall be sent if no other messages within the session were sent/received within `HeartbeatPeriod`
- Clarified that `HeartbeatPeriod` for each secure session is tracked independently
- Removed some recommendations for out of order error handling in General ordering rules.
- Allow the `END_SESSION` request to be sent from the Responder to the Requester.
- Removed `Additional Constraints` subchapter from `GET_ENCAPSULATED_REQUEST request and ENCAPSULATED_REQUEST response messages` since it's redundant and contradictory.
- Clarify `ResetRequired` for `GET_KEY_PAIR_INFO` and `SET_KEY_PAIR_INFO` .
- Add `SET_KEY_PAIR_RESET_CAP` capability.
- Fixed typo for `PublicKeyInfoLen` .
- Clarify that `MEASUREMENTS` with signature shall re-initialize L1/L2 to `null` .
- Clarify that an algorithm structure table should be present only if the Responder supports that `AlgType` .
- Clarify that MEL and HEM may not match and how to handle this scenario.
- Add PQC related fields/structures to `NEGOTIATE_ALGORITHMS` , `ALGORITHMS` , `GET_CERTIFICATE` , `CERTIFICATE` , `SET_CERTIFICATE` , `KEY_PAIR_INFO` , and `SET_KEY_PAIR_INFO` . Updated table numbers throughout to accommodate

for the new PQC specific tables.

1220 # 26 Bibliography

1221 DMTF DSP4014, *DMTF Process for Working Bodies 2.6*.