1 Document Identifier: DSP0274

2 Date: 2021-08-19

3 Version: 1.2.0WIP.89

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 the DMTF or its members. Because this document is a Work in Progress, this document may still change, perhaps profoundly and without notice. This document is available for public review and comment until superseded.

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

7 **Supersedes: 1.1.1**

8 **Document Class: Normative**

9 **Document Status: Work in Progress**

10 **Document Language: en-US**

11 DMTF is a not-for-profit association of industry members dedicated to promoting enterprise and systems management and interoperability. Members and non-members may reproduce DMTF specifications and documents, provided that correct attribution is given. As DMTF specifications may be revised from time to time, the particular version and release date should always be noted.

12 Implementation of certain elements of this standard or proposed standard may be subject to third party patent rights, including provisional patent rights (herein "patent rights"). DMTF makes no representations to users of the standard as to the existence of such rights, and is not responsible to recognize, disclose, or identify any or all such third party patent right, owners or claimants, nor for any incomplete or inaccurate identification or disclosure of such rights, owners or claimants. DMTF shall have no liability to any party, in any manner or circumstance, under any legal theory whatsoever, for failure to recognize, disclose, or identify any such third party patent rights, or for such party's reliance on the standard or incorporation thereof in its product, protocols or testing procedures. DMTF shall have no liability to any party implementing such standard, whether such implementation is foreseeable or not, nor to any patent owner or claimant, and shall have no liability or responsibility for costs or losses incurred if a standard is withdrawn or modified after publication, and shall be indemnified and held harmless by any party implementing the standard from any and all claims of infringement by a patent owner for such implementations.

13 For information about patents held by third-parties which have notified the DMTF that, in their opinion, such patent may relate to or impact implementations of DMTF standards, visit http://www.dmtf.org/about/policies/disclosures.php.

14 This document's normative language is English. Translation into other languages is permitted.

15 CONTENTS

16 1 Foreword

17 The Platform Management Components Intercommunication (PMCI) 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 1.1 Acknowledgments

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

20 **Contributors:**

- Richelle Ahlvers — Broadcom Inc.
- Lee Ballard — Dell Technologies
- 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
- Yuval Itkin — Mellanox Technologies
- Theo Koulouris — Hewlett Packard Enterprise
- Luis Luciani — Hewlett Packard Enterprise
- Masoud Manoo — Lenovo
- Donald Matthews — Advanced Micro Devices, Inc.
- Mahesh Natu — Intel Corporation
- Edward Newman — Hewlett Packard Enterprise
- Jim Panian — Qualcomm Inc.
- Scott Phuong — Cisco Systems Inc.
- Jeffrey Plank — Microchip
- Viswanath Ponnuru — Dell Technologies
- Xiaoyu Ruan — Intel Corporation
- Nitin Sarangdhar — Intel Corporation
- Hemal Shah — Broadcom Inc.
- Srikanth Varadarajan — Intel Corporation

- Jiewen Yao — Intel Corporation

21 2 Introduction

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

23 2.1 Advice

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

25 2.2 Conventions

26 The following conventions apply to all SPDM specifications.

27 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.

28 2.2.2 Reserved and unassigned values

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

30 Unless otherwise specified, reserved numeric and bit fields shall be written as zero (`0`) and ignored when read.

31 2.2.3 Byte ordering

32 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).

33 **2.2.3.1 Hash byte ordering**

34 For fields or values containing a digest or hash, SPDM preserves the byte order of the digest as defined by the specification of a given hash algorithm. SPDM views these digests, simply, as a string of octets where the first byte is the left most 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.

35 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 do 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 value at H_0[15:8], the forth byte is 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 H_0, H_1, H_7 are defined in the FIPS 180-4 specification.

36 **2.2.3.2 Encoded ASN.1 byte ordering**

37 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 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. Figure 1, Figure 2 and identifier octets are defined in X.690 specification. 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.

38 **2.2.4 SPDM data types**

39 The SPDM data types table lists the abbreviations and descriptions for common data types that SPDM message fields and data structure definitions use. These definitions follow DSP0240.

40 **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 eight bit fields. Each bit field can be separately defined.
`bitfield16`	Two-byte word with 16-bit fields. Each bit field can be separately defined.

41 ### 2.2.5 Version encoding

42 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.

43 EXAMPLE:

44 Version 3.7 → `0x37`

45 Version 1.0 → `0x10`

46 Version 1.2 → `0x12`

47 An *endpoint* that supports Version 1.2 can interoperate with an older endpoint that supports Version 1.0 only, but the available functionality is limited to what SPDM specification Version 1.0 defines.

48 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`.

49 The detailed version encoding that the `VERSION` response message returns contains an additional byte that indicates specification bug fixes or development versions. See the Successful VERSION response message format table.

50 ### 2.2.6 Notations

51 SPDM specifications use the following notations:

Notation	Description
`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.

Notation	Description
`[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, indicated by the leading `0x` .
`N+`	Variable-length byte range that starts at byte offset N.
`{ Payload }`	Used mostly in figures, this notation indicates 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.
`{ Payload }::[[S`$_X$`]]`	Used mostly in figures, this notation indicates 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`$_2$.

52 ### 2.2.7 Text or string encoding

53 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 where 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 on forth until the last character is placed in the last byte.

54 Each byte in the array shall be the numeric value that represents that character as defined in the ISO 646/ASCII table.

55 The "spdm" encoding example table shows an encoding example of the string "spdm".

56 **"spdm" encoding example**

Offset	Character	Value
0	s	0x73
1	p	0x70

Offset	Character	Value
2	d	0x64
3	m	0x6D

57 2.2.8 Deprecated material

58 Deprecated material is not recommended for use in new development efforts. Existing and new implementations may 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 in order to achieve backwards compatibility. Although implementations may use deprecated elements, they are directed to use the favored elements instead.

59 The following typographical convention indicates deprecated material:

---

60 DEPRECATED

61 Deprecated material appears here.

62 DEPRECATED

---

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

64 # 3 Scope

65 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 measurement.

66 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.

67 4 Normative references

68 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 a 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 - 2018 (8th 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 RFC2986, *PKCS #10: Certification Request Syntax Specification*, November 2000
- IETF RFC4716, *The Secure Shell (SSH) Public Key File Format*, November 2006
- IETF RFC5234, *Augmented BNF for Syntax Specifications: ABNF*, January 2008
- IETF RFC5280, *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*, May 2008
- IETF RFC7250, *Using Raw Public Keys in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)*, June 2014
- IETF RFC7919, *Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for Transport Layer Security (TLS)*, August 2016
- IETF RFC8032, *Edwards-Curve Digital Signature Algorithm (EdDSA)*, January 2017
- IETF RFC8446, *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 RFC8439, *ChaCha20 and Poly1305 for IETF Protocols*, June 2018
- IETF RFC8998, *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
- **X.509 — ISO-9594-8**
  - ITU-T X.509, 08/2015
- **ASCII — ISO/IEC 646:1991**, 09/1991
- **ECDSA**
  - Section 6, The Elliptic Curve Digital Signature Algorithm (ECDSA) in FIPS PUB 186-4 Digital Signature Standard (DSS)
  - Appendix D: Recommended Elliptic Curves for Federal Government Use in FIPS PUB 186-4 Digital Signature Standard (DSS)
- ANSI X9.62, 2005
- **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

69 # 5 Terms and definitions

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

71 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 parenthesis 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.

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

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

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

75 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 exist 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	Process of determining whether an entity is who or what it claims to be.
authentication initiator	Endpoint that initiates the authentication process by challenging another endpoint.
byte	Eight-bit quantity. Also known as an *octet*.
certificate	Digital form of identification that provides information about an entity and certifies ownership of a particular asymmetric key-pair.

Term	Definition
certificate authority (CA)	Trusted entity that issues certificates.
certificate chain	Series of two or more certificates. Each certificate is signed by the preceding certificate in the chain.
component	Physical device, contained in a single package.
device	Physical entity such as a network controller or a fan.
device certificate	Certificate that contains information that identifies the component. May be a leaf certificate or an intermediate certificate.
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 `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 for details.
endpoint	Logical entity that communicates with other endpoints over one or more transport protocol.
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.
intermediate certificate	Certificate that is neither a root certificate nor a leaf certificate.
large SPDM message	An SPDM message that is greater than the `DataTransferSize` of the receiving SPDM endpoint.
large SPDM request	A Large SPDM message that is an SPDM request.
large SPDM response	A Large SPDM message that is an SPDM response.
invasive debug mode	A device mode that enables debug access that might expose or allow modification of security critical firmware, hardware, or settings. Invasive debug mode might include access to the device TCB.
leaf certificate	Last certificate in a certificate chain.
measurement	Representation of 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.
most significant byte (MSB)	Highest order *byte* in a number consisting of multiple bytes.
Negotiated State	Set of parameters that represent 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. Nonce may be generated by combining a pseudo random number of at least 64 bits, optionally concatenated with a monotonic counter of size suitable for the application.
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. In some instances, a field can be both a payload field and a transport field.
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.
Platform Management Component Intercommunications (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 record is 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.
Reset	This term is used to denote a Reset or restart of a device that runs the Requester or Responder code, that typically leads to 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.

Term	Definition
root certificate	First certificate in a certificate chain, which is self-signed.
session keys	Session Keys are any secrets, derived cryptographic keys or any cryptographic information bound to the session.
Session-Secrets-Exchange	This term denotes 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 their 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`.
secure session	A secure session is a session that provides either or both of encryption or message authentication for communicating data over a transport.
SPDM message	Unit of communication in SPDM communications. See Generic SPDM message format for details.
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.
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 to the security policy. Reference: https://en.wikipedia.org/wiki/Trusted_computing_base

76 # 6 Symbols and abbreviated terms

77 The abbreviations defined in DSP0004, DSP0223, and DSP1001 apply to this document.

78 The following additional abbreviations are used in this document.

Abbreviation	Definition
CA	*certificate authority*
MAC	Message Authentication Code
DMTF	Formerly the *Distributed Management Task Force*
MSB	*most significant byte*
PMCI	*Platform Management Component Intercommunications*
SPDM	Security Protocol and Data Model
TCB	*trusted computing base*
AEAD	Authenticated Encryption with Associated Data
VCA	Version-Capabilities-Algorithms
KDF	Key Derivation Function

79 # 7 SPDM message exchanges

80 The message exchanges defined in this specification are between two endpoints and are performed and exchanged through sending and receiving of SPDM messages defined in SPDM messages. 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.

81 The specification-defined message exchanges enable Requesters to:

- Discover and negotiate the security capabilities of a Responder.
- Authenticate the 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.

82 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.

83 ## 7.1 Security capability discovery and negotiation

84 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 are defined in this specification. 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.

85 ## 7.2 Identity authentication

86 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.

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

- Identity provisioning
- Runtime authentication

88 ### 7.2.1 Identity provisioning

89 Identity provisioning is the process that device vendors follow during or after hardware manufacturing. A trusted root *certificate authority (CA)* generates a *root certificate* (`RootCert`) that is provisioned to the *authentication initiator*. The authentication initiator uses this certificate to verify the validity of certificate chains. A device carries a *certificate chain*, which has the `RootCert` as the root of the certificate chain and a *leaf certificate* as the last certificate of the certificate chain.

90 The certificate chain may be built according to one of two models, both of which are shown in the *SPDM certificate chain models* figure. In one model, shown on the left in the following figure, the leaf certificate is a device certificate (`DeviceCert`), which contains the public key that corresponds to the device private key. In the other model, shown on the right in the following figure, the leaf certificate is an alias certificate (`AliasCert`), in which case there may be one or more intermediate `AliasCert` certificates between the `DeviceCert` and the leaf `AliasCert` . In the `AliasCert` model, the device private key signs the next level `AliasCert` , and then the private key associated with the public key in each `AliasCert` signs the `AliasCert` below it.

91 A device that implements the `AliasCert` model may 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 `AliasCert` should be treated as mutable.

92 Through the certificate chain, the root CA indirectly endorses the per-device public/private key pair in the `DeviceCert` , where the private key is provisioned to or generated by the endpoint. When the `AliasCert` model is in use, the `AliasCert`s are endorsed by the per-device private key pair, meaning that the `AliasCert`s are also indirectly endorsed by the root CA.

93 The certificate chain should contain at least one certificate that includes hardware identity information, and the hardware identity information should be present in the device certificate, whether the `DeviceCert` or `AliasCert` model is in use. Though existing deployments may not include the hardware identity information 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 version of firmware running on the device. The Extended Key Usage extension of a hardware identity certificate may include `id-DMTF-hardware-identity` OID.

```
id-DMTF-hardware-identity  OBJECT IDENTIFIER ::= { 1 3 6 1 4 1 412 274 2 }
```

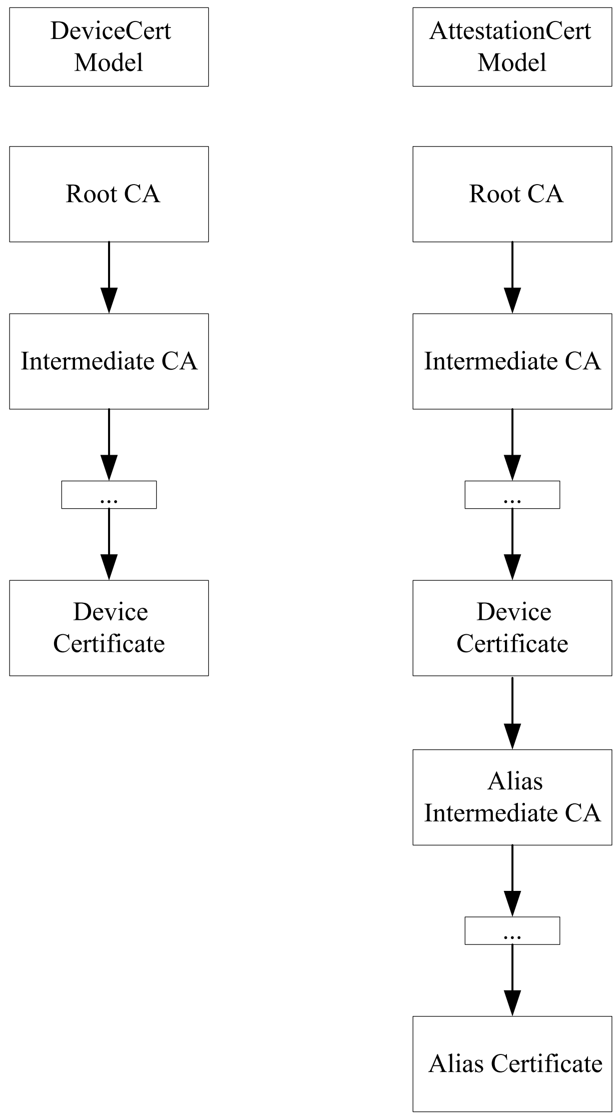
94 When the `AliasCert` model is used, the device creates and endorses one or more certificates. The certificates from the root certificate to the device certificate are immutable, and can only be changed through the `SET_CERTIFICATE` command or an equivalent capability. The certificates below the device certificate may be created on the device, and are mutable certificates, in that they may change when the device state changes, such as a device reset. The mutable certificates may include the `id-DMTF-mutable-certificate` OID in the Extended Key Usage extension of the certificate to identify them as mutable.

```
id-DMTF-mutable-certificate  OBJECT IDENTIFIER ::= { 1 3 6 1 4 1 412 274 5 }
```

95 In addition, when the `AliasCert` model is used, one or more `AliasCert`s may contain firmware identity information. The format of the firmware identity information may be defined by other standards bodies, and is outside the scope of this specification.

96 A Responder may use the `DeviceCert` model or the `AliasCert` model. A Requester should be capable of performing Runtime authentication on a certificate chain that conforms to either model.

97 **SPDM certificate chain models**



98 #### 7.2.1.1 Raw public keys

99 Alternatively to certificate chains, the vendor may 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.

100 The format of the provisioned public key is out of scope of this specification. Vendors can use proprietary formats or public key formats that other standards define, such as RFC7250 and RFC4716.

101 7.2.2 Runtime authentication

102 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.

103 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. The transport layer should handle device identification, which is outside the scope of this specification.

104 7.3 Firmware and configuration measurement

105 A measurement is a representation of firmware/software or configuration data on an endpoint. A measurement is typically 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.

106 7.4 Secure sessions

107 Many devices exchange data with other devices that may require protection. In this specification, the device-specific data that is communicated is generically referred to as application data. The protocol of the application data usually exists at a higher layer and it is outside the scope of this specification. The protocol of the application data usually allows for encrypted and/or authenticated data transfer.

108 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 which can be considered equivalent to Runtime authentication. For more details, see the Session clause.

109 Finally, many SPDM requests and their corresponding responses can also be afforded the same protection. See the SPDM request and response messages validity table and SPDM request and response code issuance allowance clause for more details.

110 The SPDM messaging protocol flow gives a very high-level view of when the secure session actually starts.

111 ## 7.5 Mutual authentication overview

112 The ability for 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 apply to the Requester as well.

113 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 will provide more details.

114 ## 7.6 Notifications overview

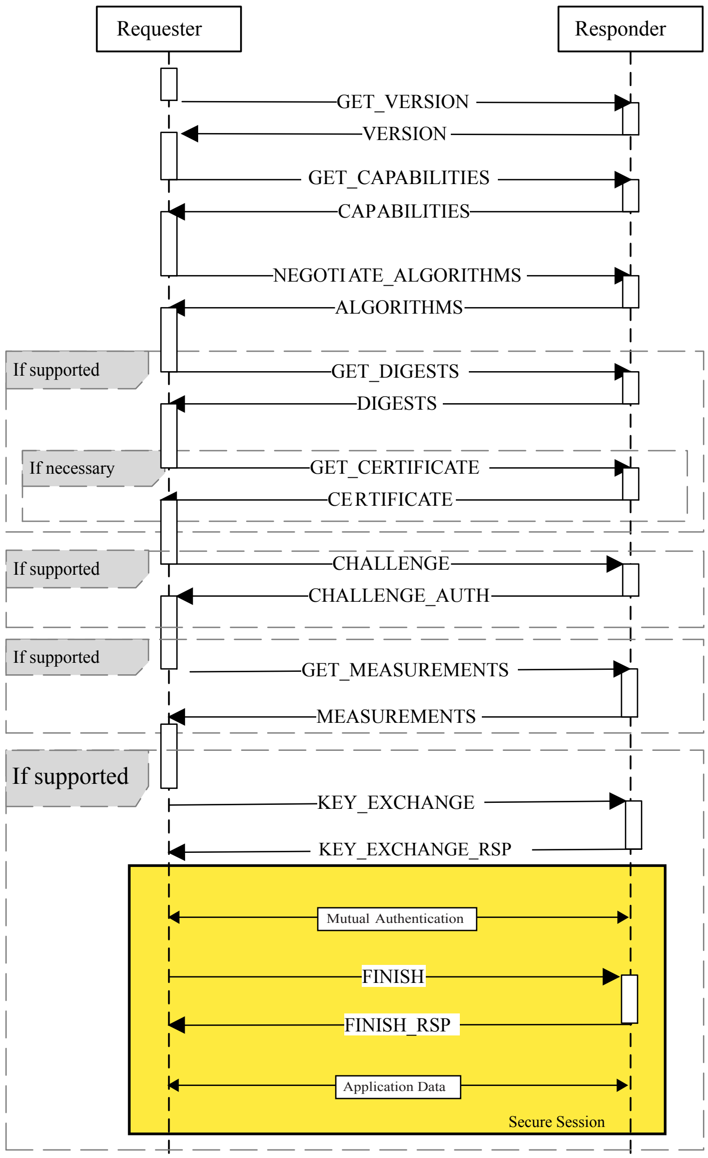
115 To aid an SPDM endpoint in enforcing its security policy requirements in an efficient, reliable and timely manner, SPDM event mechanism provides a method to asynchronously deliver or receive a notification to the interested SPDM endpoint. This mechanism allows an interested SPDM endpoint to choose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

116 8 SPDM messaging protocol

117 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 defined in this specification unless otherwise stated in this specification.

118 The SPDM messaging protocol flow depicts the 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.

119 **SPDM messaging protocol flow**



120 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 each of the request and response messages.

121 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` may be saved by the Requester so that after Reset the Requester may skip these requests.

122 8.1 SPDM bits-to-bytes mapping

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

124 **One-byte field bit map**

125 Example: A One-Byte Field

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

126 **Two-byte field bit map**

127 Example: A Two-Byte Field

Byte 3								Byte 2							
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

128 8.2 Generic SPDM message format

129 The Generic SPDM message field definitions table defines the fields that constitute a generic SPDM message, including the message header and payload.

130 **Generic SPDM message field definitions**

Byte	Bits	Length (bits)	Field	Description
`0`	`[7:4]`	4	`SPDM Major Version`	The major version of the SPDM Specification. An endpoint shall not communicate by using an incompatible SPDM version value. See Version encoding.

Byte	Bits	Length (bits)	Field	Description
`0`	`[3:0]`	4	`SPDM Minor Version`	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.
`1`	`[7:0]`	8	`Request Response Code`	The request message code or response code, which Table 4 and Table 5 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`	The first one-byte parameter. The contents of the parameter is specific to the `Request Response Code`.
`3`	`[7:0]`	8	`Param2`	The second one-byte parameter. The contents of the parameter is specific to the `Request Response Code`.
`4`	See the description.	Variable	SPDM message payload	Zero or more bytes that are specific to the `Request Response Code`.

131 ### 8.2.1 SPDM version

132 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 the 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.

133 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.

134 The version of this specification can be found on the title page or the header or footer of each page in this document.

135 The `SPDMVersion` for the version of this specification shall be 0x12.

136 The `FullSPDMversionString` shall be the string form of the concatenation of major version, ".", minor version, "." and update version. For example, if the version of this specification is 1.2.3, then `FullSPDMversionString` is "1.2.3".

137 ## 8.3 SPDM request codes

138 The SPDM request codes table defines the SPDM request codes. The **Implementation requirement** column indicates requirements on the Requester.

139 All SPDM-compatible implementations shall use the following SPDM request codes.

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

141 **SPDM request codes**

Request	Code value	Implementation requirement	Message format
`GET_DIGESTS`	`0x81`	Optional	GET_DIGESTS request message format
`GET_CERTIFICATE`	`0x82`	Optional	GET_CERTIFICATE request message format
`CHALLENGE`	`0x83`	Optional	CHALLENGE request message format
`GET_VERSION`	`0x84`	Required	GET_VERSION request message format
`CHUNK_SEND`	`0x85`	Optional	CHUNK_SEND message format
`CHUNK_GET`	`0x86`	Optional	CHUNK_GET request message format
`GET_MEASUREMENTS`	`0xE0`	Optional	GET_MEASUREMENTS request message format
`GET_CAPABILITIES`	`0xE1`	Required	GET_CAPABILITIES request message format
`GET_SUPPORTED_EVENT_GROUPS`	`0xE2`	Optional	GET_SUPPORTED_EVENT_GROUPS request message format
`NEGOTIATE_ALGORITHMS`	`0xE3`	Required	NEGOTIATE_ALGORITHMS request message format
`KEY_EXCHANGE`	`0xE4`	Optional	KEY_EXCHANGE request message format
`FINISH`	`0xE5`	Optional	FINISH request message format
`PSK_EXCHANGE`	`0xE6`	Optional	PSK_EXCHANGE request message format
`PSK_FINISH`	`0xE7`	Optional	PSK_FINISH request message format
`HEARTBEAT`	`0xE8`	Optional	HEARTBEAT request message format
`KEY_UPDATE`	`0xE9`	Optional	KEY_UPDATE request message format
`GET_ENCAPSULATED_REQUEST`	`0xEA`	Optional	GET_ENCAPSULATED_REQUEST request message format
`DELIVER_ENCAPSULATED_RESPONSE`	`0xEB`	Optional	DELIVER_ENCAPSULATED_RESPONSE request message format
`END_SESSION`	`0xEC`	Optional	END_SESSION request message format

Request	Code value	Implementation requirement	Message format
`GET_CSR`	`0xED`	Optional	GET_CSR request message format
`SET_CERTIFICATE`	`0xEE`	Optional	SET_CERTIFICATE request message format
`SUBSCRIBE_EVENT_GROUP`	`0xEF`	Optional	SUBSCRIBE_EVENT_GROUP request message format
`SEND_EVENT`	`0xF0`	Optional	SEND_EVENT request message format
`VENDOR_DEFINED_REQUEST`	`0xFE`	Optional	VENDOR_DEFINED_REQUEST request message format
`RESPOND_IF_READY`	`0xFF`	Required	RESPOND_IF_READY request message format
Reserved	`0x80` , `0x85` - `0xDF` , `0xF1` - `0xFD`	SPDM implementations compatible with this version shall not use the reserved request codes.	

142 ## 8.4 SPDM response codes

143 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 the following SPDM response codes.

144 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.

145 The SPDM response codes table defines the response codes for SPDM. The **Implementation requirement** column indicates requirements on the Responder.

146 **SPDM response codes**

Response	Value	Implementation requirement	Message format
`DIGESTS`	`0x01`	Optional	Successful DIGESTS response message format
`CERTIFICATE`	`0x02`	Optional	Successful CERTIFICATE response message format
`CHALLENGE_AUTH`	`0x03`	Optional	Successful CHALLENGE_AUTH response message format

Response	Value	Implementation requirement	Message format
`VERSION`	`0x04`	Required	Successful VERSION response message format
`CHUNK_SEND_ACK`	`0x05`	Optional	CHUNK_SEND_ACK response message format
`CHUNK_RESPONSE`	`0x06`	Optional	CHUNK_RESPONSE response message format
`MEASUREMENTS`	`0x60`	Optional	Successful MEASUREMENTS response message format
`CAPABILITIES`	`0x61`	Required	Successful CAPABILITIES response message format
`SUPPORTED_EVENT_GROUPS`	`0x62`	Optional	SUPPORTED_EVENT_GROUPS response message format
`ALGORITHMS`	`0x63`	Required	Successful ALGORITHMS response message format
`KEY_EXCHANGE_RSP`	`0x64`	Optional	Successful KEY_EXCHANGE_RSP response message format
`FINISH_RSP`	`0x65`	Optional	Successful FINISH_RSP response message format
`PSK_EXCHANGE_RSP`	`0x66`	Optional	PSK_EXCHANGE_RSP response message format
`PSK_FINISH_RSP`	`0x67`	Optional	Successful PSK_FINISH_RSP response message format
`HEARTBEAT_ACK`	`0x68`	Optional	HEARTBEAT_ACK response message format
`KEY_UPDATE_ACK`	`0x69`	Optional	KEY_UPDATE_ACK response message format
`ENCAPSULATED_REQUEST`	`0x6A`	Optional	ENCAPSULATED_REQUEST response message format
`ENCAPSULATED_RESPONSE_ACK`	`0x6B`	Optional	ENCAPSULATED_RESPONSE_ACK response message format
`END_SESSION_ACK`	`0x6C`	Optional	END_SESSION_ACK response message format
`CSR`	`0x6D`	Optional	CSR response message format

Response	Value	Implementation requirement	Message format
`SET_CERTIFICATE_RSP`	`0x6E`	Optional	SET_CERTIFICATE_RSP response message format
`SUBSCRIBE_EVENT_GROUP_ACK`	`0x6F`	Optional	SUBSCRIBE_EVENT_GROUP_ACK response message format
`EVENT_ACK`	`0x70`	Optional	EVENT_ACK response message format
`VENDOR_DEFINED_RESPONSE`	`0x7E`	Optional	VENDOR_DEFINED_RESPONSE response message format
`ERROR`	`0x7F`		ERROR response message format
Reserved	`0x00` , `0x05` - `0x5F` , `0x71` - `0x7D`	SPDM implementations compatible with this version shall not use the reserved response codes.	

147 ## 8.5 SPDM request and response code issuance allowance

148 The SPDM request and response messages validity table describes the conditions under which a request and response can be issued.

149 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.

150 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.

151 A request and its corresponding response can have the `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 may have additional restrictions. See the respective request and response clause for further details.

152 A request and its corresponding response that has `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 of the SPDM specification. The session ID is implicit from the session used to transmit the commands.

153 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.

154 For details, see the Session clause.

155 **SPDM request and response messages validity**

Request	Response	Session	Outside of a session	Session phases
`GET_MEASUREMENT`	`MEASUREMENT`	Allowed	Allowed	Application Phase
`FINISH`	`FINISH_RSP`	Allowed	Prohibited	Session Handshake
`PSK_FINISH`	`PSK_FINISH_RSP`	Allowed	Prohibited	Session Handshake
`HEARTBEAT`	`HEARTBEAT_ACK`	Allowed	Prohibited	Application Phase
`KEY_UPDATE`	`KEY_UPDATE_ACK`	Allowed	Prohibited	Application Phase
`END_SESSION`	`END_SESSION_ACK`	Allowed	Prohibited	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`	Allowed	Allowed	Application Phase
`GET_SUPPORTED_EVENT_GROUPS`	`SUPPORTED_EVENT_GROUPS`	Allowed	Prohibited	Application Phase
`SUBSCRIBE_EVENT_GROUP`	`SUBSCRIBE_EVENT_GROUP_ACK`	Allowed	Prohibited	Application Phase
`SEND_EVENT`	`EVENT_ACK`	Allowed	Prohibited	Application Phase
`CHUNK_SEND`	`CHUNK_SEND_ACK`	Allowed	Allowed	All Phases
`CHUNK_GET`	`CHUNK_RESPONSE`	Allowed	Allowed	All Phases
All others	All others	Prohibited	Allowed	Not Applicable

156 For `ERROR` response in the session handshake or application phase of a session, the Requester is only allowed in certain situations to send the `ERROR` response.

157 8.6 Concurrent SPDM message processing

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

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

160 8.7 Requirements for Requesters

161 A Requester shall not have multiple outstanding requests to the same Responder, with the following exception: as addressed in GET_VERSION request and VERSION response messages, a Requester may issue a `GET_VERSION` to a Responder to restart the protocol at any time, even if the Requester has existing outstanding requests to the same Responder.

162 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.

163 A Requester may send simultaneous request messages to different Responders.

164 8.8 Requirements for Responders

165 A Responder is not required to process more than one request message at a time.

166 A Responder that is not ready to accept a new request message shall either respond with an `ERROR` response message with `ErrorCode=Busy` or silently discard the request message.

167 If a Responder is working on a request message from a Requester, the Responder may respond with `ErrorCode=Busy`.

168 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.

169 # 9 Timing requirements

170 The Timing specification for SPDM messages table shows the timing specifications for Requesters and Responders.

171 If the Requester does not receive a response within **T1** or **T2** time accordingly, the Requester may retry a request message. A retry of a request message shall be a complete retransmission of the original SPDM request message.

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

173 ## 9.1 Timing measurements

174 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. 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.

175 ## 9.2 Timing specification table

176 The **Ownership** column in the Timing specification for SPDM messages table 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.

177 **Timing specification for SPDM messages**

Timing parameter	Ownership	Value	Units	Description
`RTT`	Requester	See the description.	μs	Worst case round-trip transport timing. The maximum 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.
`ST1`	Responder	100,000	μs	Shall be the maximum amount of time the Responder has to provide a response to requests that do not require cryptographic processing, such as the `GET_CAPABILITIES` , `GET_VERSION` , or `NEGOTIATE_ALGORITHMS` request messages.

Timing parameter	Ownership	Value	Units	Description
`T1`	Requester	`RTT+ST1`	μs	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 `ST1` .
`CT`	Requester and Responder	$2^{\texttt{CTExponent}}$	μs	`CTExponent` is reported in `GET_CAPABILITIES` and `CAPABILITIES` messages. This timing parameter shall be the maximum amount of time the endpoint has to provide any response requiring cryptographic processing, such as the `GET_MEASUREMENTS` or `CHALLENGE` request messages.
`T2`	Requester	`RTT+CT`	μs	Shall be the minimum amount of time the Requester shall wait before issuing a retry for requests that require cryptographic processing. For details, see `CT` .
`RDT`	Responder	$2^{\texttt{RDTExponent}}$	μs	Recommended delay, 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. See the ResponseNotReady extended error data table for the `RDTExponent` value. For details, see `ErrorCode=ResponseNotReady` in the ResponseNotReady extended error data table.
`WT`	Requester	`RDT`	μs	Amount of time that the Requester should wait before issuing the `RESPOND_IF_READY` request message. The Requester shall measure this time parameter from the reception of the `ERROR` response to the transmission of `RESPOND_IF_READY` request. The Requester may take into account the transmission time of the `ERROR` from the Responder to Requester when calculating `WT` . For details, see `RDT` .

Timing parameter	Ownership	Value	Units	Description
`WT Max`	Requester	`(RDT*RDTM)−RTT`	μs	Maximum wait time the Requester has to issue `RESPOND_IF_READY` request unless the Requester issued a successful `RESPOND_IF_READY` request message earlier. After this time the Responder is allowed to drop the response. The Requester shall take into account the transmission time of the `ERROR` from the Responder to Requester when calculating `WT Max`. The `RDTM` value appears in the ResponseNotReady extended error data. The Responder should ensure that `WT Max` does not result in less than `WT` in determination of `RDTM`. For details, see `ErrorCode=ResponseNotReady` in the ResponseNotReady extended error data table.
`HeartbeatPeriod`	Requester and Responder	Variable	s	See HEARTBEAT request and HEARTBEAT_ACK response for detail.
`LMTO`	See Description	`ST1`	μs	Large SPDM message timeout. This parameter shall be the maximum amount of time the Requester of `CHUNK_SEND` or `CHUNK_GET` has to issue the respective request for the next chunk in the sequence after receiving the previous chunk of data. Failure to comply with this timing requirement may result in the loss or unexpected termination of a Large SPDM message transfer. See Large SPDM message transfer for details.

178 10 SPDM messages

179 SPDM messages can be divided into the following categories, supporting different aspects of security exchanges between a Requester and Responder:

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

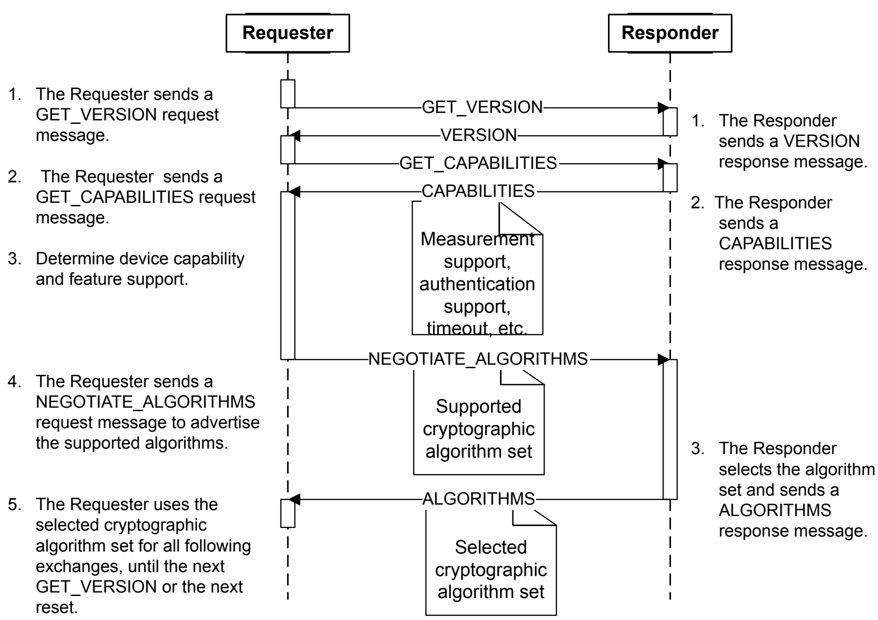
180 10.1 Capability discovery and negotiation

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

182 The Capability discovery and negotiation flow shows the high-level request-response flow and sequence for the capability discovery and negotiation:

183 **Capability discovery and negotiation flow**

184



185 10.1.1 Negotiated state preamble

186 The `VCA` (Version-Capabilities-Algorithms) refers to the concatenation of messages `GET_VERSION` , `VERSION` , `GET_CAPABILITIES` , `CAPABILITIES` , `NEGOTIATE_ALGORITHMS` , and `ALGORITHMS` last exchanged between the Requester and the Responder.

187 If the Responder supports caching the negotiated state (`CACHE_CAP=1`), the Requester may 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.

188 If the two endpoints support session key establishment with the PSK (Pre-Shared Key) option, then Negotiated State is not applicable and `VCA` is not stored.

189 10.2 GET_VERSION request and VERSION response messages

190 This request message shall retrieve the SPDM version of an endpoint. The GET_VERSION request message format table shows the `GET_VERSION` request message format and the Successful VERSION response message format table shows the `VERSION` response message format.

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

192 The Requester shall begin the discovery process by sending a `GET_VERSION` request message with major version `0x1` . All Responders shall always support the `GET_VERSION` request message with major version `0x1` and provide a `VERSION` response containing all supported versions, as the GET_VERSION request message format table describes.

193 The Requester shall consult the `VERSION` response to select a common supported version, which is typically 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 `ErrorCode=ResponseNotReady` .

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

195 After receiving a `GET_VERSION` request, the Responder shall cancel all previous requests from the same Requester. All active sessions between the Requester and the Responder are terminated, i.e., information (such as session keys, 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.

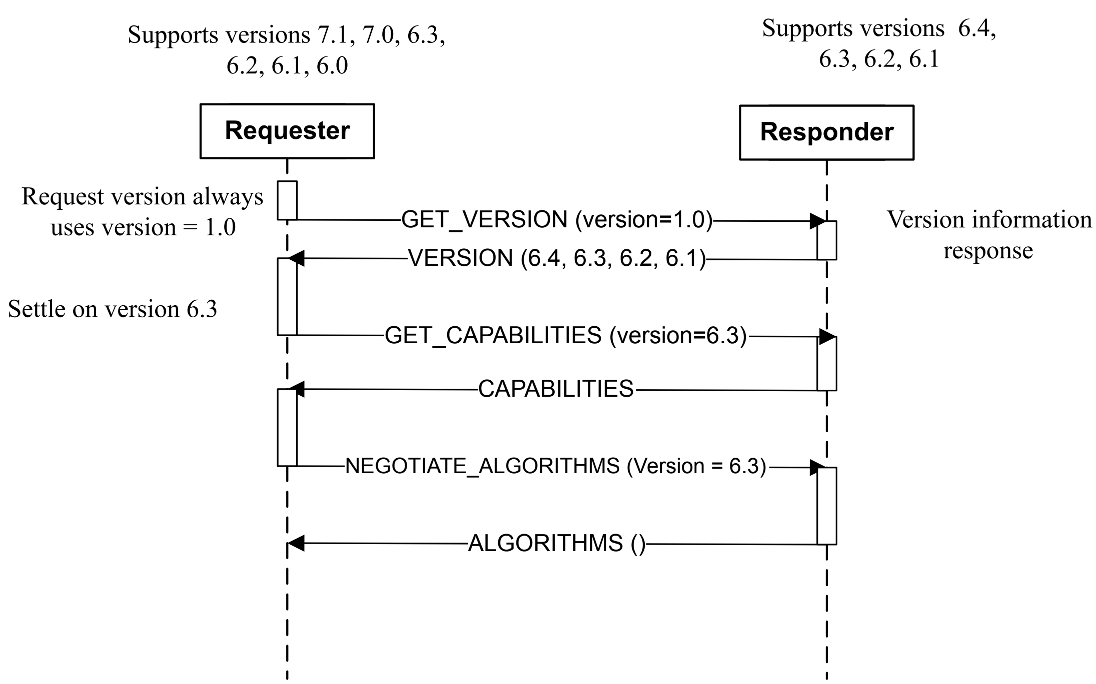
196 After sending the `VERSION` response for a `GET_VERSION` request, if the Responder completes a runtime code or

configuration change for its hardware or firmware measurement and the change has taken effect, then the Responder shall perform these steps:

1. If the Responder is an Event Notifier (i.e. `EVENT_CAP` is set) and supports `MeasurementEvent` in DMTF event group and the Requester subscribed to the DMTF event group, the Responder shall send each changed measurement as a `MeasurementEvent`. See Event mechanism for details.
2. Otherwise, the Responder shall silently discard any request or respond with `ErrorCode=RequestResynch` to any request until a `GET_VERSION` request is received.

197 **Discovering the common major version**

198



199 **GET_VERSION request message format**

Offset	Field	Size (bytes)	Value
`0`	`SPDMVersion`	1	Shall be 0x10 (V1.0).
`1`	`RequestResponseCode`	1	`0x84=GET_VERSION`
`2`	`Param1`	1	Reserved.

Offset	Field	Size (bytes)	Value
3	Param2	1	Reserved.

200 **Successful VERSION response message format**

Offset	Field	Size (bytes)	Value
0	SPDMVersion	1	Shall be 0x10 (V1.0).
1	RequestResponseCode	1	0x04=VERSION
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 the VersionNumberEntry definition table.

201 **VersionNumberEntry definition**

Bit	Field	Value
[15:12]	MajorVersion	Version of the specification with changes that are incompatible with one or more functions in earlier major versions of the specification.
[11:8]	MinorVersion	Version of the specification with changes that are compatible with functions in earlier minor versions of this major version specification.
[7:4]	UpdateVersionNumber	Version of the specification with editorial updates but no functionality additions or changes. Informational; possible errata fixes. Ignore when checking versions for interoperability.
[3:0]	Alpha	Pre-release work-in-progress version of the specification. Backward compatible with earlier minor versions of this major version specification. However, 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 may not be fully interoperable. Released versions shall have an Alpha value of zero (0).

202 10.3 GET_CAPABILITIES request and CAPABILITIES response messages

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

204 The GET_CAPABILITIES request message format table shows the `GET_CAPABILITIES` request message format.

205 The Successful CAPABILITIES response message format table shows the `CAPABILITIES` response message format.

206 The Requester flag fields definitions table shows the flag fields definitions for the Requester.

207 Likewise, the Responder flag fields definitions table shows the flag fields definitions for the Responder.

208 A Responder shall not respond to `GET_CAPABILITIES` request message with `ErrorCode=ResponseNotReady`.

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

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

210 Both the Requester and Responder shall support a minimum buffer size in order to successfully transfer SPDM messages. The minimum size, referred to as `MinDataTransferSize`, shall be the size, in bytes, of the SPDM message with the largest size in this list:

- `GET_VERSION`
- `GET_CAPABILITIES`
- `CAPABILITIES`
- `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

211 The calculation of `MinDataTransferSize` shall assume all fields are present. For this version of the specification, the `MinDataTransferSize` shall be 44.

212 **GET_CAPABILITIES request message format**

Offset	Field	Size (bytes)	Value
`0`	`SPDMVersion`	1	Shall be the `SPDMVersion` as described in SPDM version.
`1`	`RequestResponseCode`	1	`0xE1=GET_CAPABILITIES`
`2`	`Param1`	1	Reserved.

Offset	Field	Size (bytes)	Value
3	Param2	1	Reserved.
4	Reserved	1	Reserved.
5	CTExponent	1	Shall be exponent of base 2, which is used to calculate CT. See the Timing specification for SPDM messages table. 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 the Requester flag fields definitions table.
12	DataTransferSize	4	This field shall indicate the maximum buffer size, in bytes, of the Requester for receiving a single SPDM message. The value of this field shall be equal to or greater than MinDataTransferSize.
16	MaxSPDMmsgSize	4	This field shall indicate the maximum size, in bytes, of the internal buffer of a Requester for processing a single Large SPDM message. This field shall be greater than or equal to DataTransferSize.

213 **Successful CAPABILITIES response message format**

Offset	Field	Size (bytes)	Value
0	SPDMVersion	1	Shall be the SPDMVersion as described in SPDM version.
1	RequestResponseCode	1	0x61=CAPABILITIES
2	Param1	1	Reserved.
3	Param2	1	Reserved.
4	Reserved	1	Reserved.
5	CTExponent	1	Shall be the exponent of base 2, which used to calculate CT. See the Timing specification for SPDM messages table. 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.

Offset	Field	Size (bytes)	Value
8	Flags	4	See the Responder flag fields definitions table.
12	DataTransferSize	4	This field shall indicate the maximum buffer size, in bytes, of the Responder for receiving a single SPDM message. The value of this field shall be equal to or greater than MinDataTransferSize .
16	MaxSPDMmsgSize	4	This field shall indicate the maximum size, in bytes, of the internal buffer of a Responder for processing a single Large SPDM message. This field shall be greater than or equal to DataTransferSize .

214 As described in other parts of this specification, a Requester or Responder can reverse roles or be both roles for certain SPDM messages and flows. Thus, in general, an SPDM endpoint cannot send an 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 the receiving 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 ErrorCode == ResponseTooLarge or silently discard the request. Likewise, if the size of a request message exceeds the size of the MaxSPDMmsgSize of the responding SPDM endpoint, the responding SPDM endpoint shall respond with ErrorCode=RequestTooLarge or silently discard the request. Additionally, an SPDM endpoint should provide graceful error handling (e.g., buffer overflow/underflow protection) in the event they receive an SPDM messages that exceed their MaxSPDMmsgSize .

215 **Requester flag fields definitions**

216 Unless otherwise stated, if a Requester indicates support of 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 CERT_CAP bit to 1 , the Requester can receive a GET_CERTIFICATE request and send back a successful CERTIFICATE response message.

Byte	Bit	Field	Value
0	0	Reserved	Reserved.
0	1	CERT_CAP	If set, Requester supports DIGESTS and CERTIFICATE response messages.
0	2	CHAL_CAP	If set, Requester supports CHALLENGE_AUTH response message.
0	5:3	Reserved	Reserved.
0	6	ENCRYPT_CAP	If set, Requester supports 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	Bit	Field	Value
`0`	`7`	`MAC_CAP`	If set, Requester supports 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.
`1`	`0`	`MUT_AUTH_CAP`	If set, Requester supports mutual authentication.
`1`	`1`	`KEY_EX_CAP`	If set, Requester supports `KEY_EXCHANGE` messages. If set, `ENCRYPT_CAP` or `MAC_CAP` shall be set.
`1`	`3:2`	`PSK_CAP`	Pre-shared key capabilities of the Requester. `00b` . Requester does not support pre-shared key capabilities. `01b` . Requester supports pre-shared key `10b` and `11b` . Reserved. If supported, `ENCRYPT_CAP` or `MAC_CAP` shall be set.
`1`	`4`	`ENCAP_CAP`	**DEPRECATED:** If Basic mutual authentication is supported, this field shall be set.
`1`	`5`	`HBEAT_CAP`	If set, Requester supports `HEARTBEAT` messages.
`1`	`6`	`KEY_UPD_CAP`	If set, Requester supports `KEY_UPDATE` messages.
`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, 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 does not support encryption and message authentication, then this bit shall be zero.
`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` of the Requester shall be `0` .
`2`	`1`	`EVENT_CAP`	If set, the Requester is an Event Notifier. See Event mechanism for details.
`2`	`2`	`Reserved`	Reserved.
`2`	`7:3`	`Reserved`	Reserved.
`3`	`7:0`	`Reserved`	Reserved.

217 **Responder flag fields definitions**

218 Unless otherwise stated, if a Responder indicates support of 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 `CERT_CAP` bit to `1`, the Responder can receive a `GET_CERTIFICATE` request and send back a successful `CERTIFICATE` response message.

Byte	Bit	Field	Value
`0`	`0`	`CACHE_CAP`	If set, the Responder supports 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 supports `DIGESTS` and `CERTIFICATE` response messages.
`0`	`2`	`CHAL_CAP`	If set, Responder supports `CHALLENGE_AUTH` response message.
`0`	`4:3`	`MEAS_CAP`	`MEASUREMENT` response capabilities of the Responder. `00b`. The Responder does not support `MEASUREMENTS` response capabilities. `01b`. The Responder supports `MEASUREMENTS` response but cannot perform signature generation. `10b`. The Responder supports `MEASUREMENTS` response and can generate signatures. `11b`. Reserved.
`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 supports 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 supports 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 supports message authentication in a secure session. If set, `PSK_CAP` or `KEY_EX_CAP` shall be set accordingly to indicate support.
`1`	`0`	`MUT_AUTH_CAP`	If set, Responder supports mutual authentication.
`1`	`1`	`KEY_EX_CAP`	If set, Responder supports `KEY_EXCHANGE` messages. If set, `ENCRYPT_CAP` or `MAC_CAP` shall be set.

Byte	Bit	Field	Value
1	3:2	PSK_CAP	Pre-Shared Key capabilities of the Responder. 00b . Responder does not support Pre-Shared Key capabilities. 01b . Responder supports Pre-Shared Key but does not provide ResponderContext for session key derivation. 10b . Responder supports Pre-Shared Key and provides ResponderContext for session key derivation. 11b . Reserved. If supported, ENCRYPT_CAP or MAC_CAP shall be set.
1	4	ENCAP_CAP	If set, Responder supports GET_ENCAPSULATED_REQUEST , ENCAPSULATED_REQUEST , DELIVER_ENCAPSULATED_RESPONSE , and ENCAPSULATED_RESPONSE_ACK messages. Additionally, the transport may require the Responder to support these messages. **DEPRECATED:** If Basic mutual authentication is supported, this field shall be set.
1	5	HBEAT_CAP	If set, Responder supports HEARTBEAT messages.
1	6	KEY_UPD_CAP	If set, Responder supports 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, discussed in other parts of this specification. If the Responder does not support encryption and message authentication, then this bit shall be zero.
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, CERT_CAP of the Responder shall be 0 .
2	1	EVENT_CAP	If set, the Responder is an Event Notifier. See Event mechanism for details.
2	2	ALIAS_CERT_CAP	If set, the Responder uses the AliasCert model. See Identity provisioning for details.
2	7:3	Reserved	Reserved.
3	7:0	Reserved	Reserved.

219 In the case where an SPDM implementation incorrectly returns an illegal combination of capability flags, as these 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 and should set the ErrorCode = InvalidRequest . If a Requester detects an illegal capability flag combination reported by the Responder it should retry once by issuing GET_VERSION

and `GET_CAPABILITIES`. If the illegal combination is returned again it should cease communicating with this particular Responder over SPDM and log an error in an implementation-specific manner to assist with identifying the problem.

220 10.4 NEGOTIATE_ALGORITHMS request and ALGORITHMS response messages

221 This request message shall negotiate cryptographic algorithms. A Requester shall not issue a `NEGOTIATE_ALGORITHMS` request message until it receives a successful `CAPABILITIES` response message.

222 A Requester shall not issue any other SPDM requests, with the exception of `GET_VERSION`, until it receives a successful `ALGORITHMS` response message.

223 A Responder shall not respond to `NEGOTIATE_ALGORITHMS` request message with `ErrorCode=ResponseNotReady`.

224 The NEGOTIATE_ALGORITHMS request message format table shows the `NEGOTIATE_ALGORITHMS` request message format.

225 The Successful ALGORITHMS response message format table shows the `ALGORITHMS` response message format.

226 **NEGOTIATE_ALGORITHMS request message format**

Offset	Field	Size (bytes)	Value
`0`	`SPDMVersion`	1	Shall be the `SPDMVersion` as described in SPDM version.
`1`	`RequestResponseCode`	1	`0xE3=NEGOTIATE_ALGORITHMS`
`2`	`Param1`	1	Number of algorithms structure tables in this request using ReqAlgStruct
`3`	`Param2`	1	Reserved
`4`	`Length`	2	Length of the entire request message, in bytes. Length shall be less than or equal to 128 bytes.
`6`	`MeasurementSpecification`	1	Bit mask. The `MeasurementSpecification` field of the Measurement block format table defines the values in this field. The Requester may set more than one bit to indicate multiple measurement specification support.
`7`	`Reserved`	1	Reserved

Offset	Field	Size (bytes)	Value
`8`	`BaseAsymAlgo`	4	Bit mask listing Requester-supported SPDM-enumerated asymmetric key signature algorithms for the purpose of signature verification. If the capabilities do not support this algorithm, this value is not used and shall be set to zero. Let `SigLen` be the size of the signature in bytes. If the size of a signature component is less than specified size, then `0x00` octets are padded to the left of the most significant byte. 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 r followed by 32-byte 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.

Offset	Field	Size (bytes)	Value
`12`	`BaseHashAlgo`	4	Bit mask listing Requester-supported SPDM-enumerated cryptographic hashing algorithms. If the capabilities do not support this algorithm, this value is not used and 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.
`16`	`Reserved`	12	Reserved
`28`	`ExtAsymCount`	1	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 capabilities do not support this algorithm, this value is not used and shall be set to zero.
`29`	`ExtHashCount`	1	Number of Requester-supported extended hashing algorithms (=E). A + E + ExtAlgCount2 + ExtAlgCount3 + ExtAlgCount4 + ExtAlgCount5 shall be less than or equal to 20. If the capabilities do not support this algorithm, this value is not used and shall be set to zero.
`30`	`Reserved`	2	Reserved
`32`	`ExtAsym`	4*A	List of Requester-supported extended asymmetric key signature algorithms for the purpose of signature verification. The Extended algorithm field format table describes the format of this field.
`32 + 4*A`	`ExtHash`	4*E	List of the extended hashing algorithms supported by Requester. The Extended algorithm field format table describes the format of this field.
`32 + 4*A + 4*E`	`ReqAlgStruct`	AlgStructSize	See the `AlgStructure` request field.

227 `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.

228 **Algorithm request structure**

Offset	Field	Size (bytes)	Value
`0`	`AlgType`	1	Type of algorithm. 0 and 1 = Reserved 2 = DHE 3 = AEADCipherSuite 4 = ReqBaseAsymAlg 5 = KeySchedule All other values reserved.
`1`	`AlgCount`	1	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	Bit mask listing Requester-supported SPDM-enumerated algorithms.
`2 + FixedAlgCount`	`AlgExternal`	4*ExtAlgCount	List of Requester-supported extended algorithms. The Extended algorithm field format table describes the format of this field.

229 The following tables describe the associated fixed fields for the individual types.

230 **DHE structure**

Offset	Field	Size (bytes)	Value
`0`	`AlgType`	1	`0x2=DHE`
`1`	`AlgCount`	1	Bit [7:4] = 2. Bit [3:0] = Number of Requester-supported extended DHE groups (= ExtAlgCount2).

Offset	Field	Size (bytes)	Value
2	`AlgSupported`	2	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) (D = 64, C = 32) All other values reserved.
4	`AlgExternal`	4*ExtAlgCount2	List of Requester-supported extended DHE groups. The Extended algorithm field format table describes the format of this field.

231 **AEAD structure**

Offset	Field	Size (bytes)	Value
`0`	`AlgType`	1	`0x3=AEAD`
`1`	`AlgCount`	1	Bit [7:4] = 2. Bit [3:0] = Number of Requester supported extended AEAD algorithms (= ExtAlgCount3).
2	`AlgSupported`	2	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.

Offset	Field	Size (bytes)	Value
`4`	`AlgExternal`	4*ExtAlgCount3	List of Requester-supported extended AEAD algorithms. The Extended algorithm field format table describes the format of this field.

232 **ReqBaseAsymAlg structure**

Offset	Field	Size (bytes)	Value
`0`	`AlgType`	1	`0x4=ReqBaseAsymAlg`
`1`	`AlgCount`	1	Bit [7:4] = 2. Bit [3:0] = Number of Requester supported extended asymmetric key signature algorithms for the purpose of signature generation (= ExtAlgCount4).
`2`	`AlgSupported`	2	Bit mask listing Requester-supported SPDM-enumerated asymmetric key signature algorithms for the purpose of signature generation. 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.
`4`	`AlgExternal`	4*ExtAlgCount4	List of Requester-supported extended asymmetric key signature algorithms for the purpose of signature generation. The Extended algorithm field format table describes the format of this field.

233 **KeySchedule structure**

Offset	Field	Size (bytes)	Value
`0`	`AlgType`	1	`0x5=KeySchedule`
`1`	`AlgCount`	1	Bit [7:4] = 2. Bit [3:0] = Number of Requester supported extended key schedule algorithms (= ExtAlgCount5).
`2`	`AlgSupported`	2	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	List of Requester-supported extended key schedule algorithms. The Extended algorithm field format table describes the format of this field.

234 **Successful ALGORITHMS response message format**

Offset	Field	Size (bytes)	Value
`0`	`SPDMVersion`	1	Shall be the `SPDMVersion` as described in SPDM version.
`1`	`RequestResponseCode`	1	`0x63=ALGORITHMS`
`2`	`Param1`	1	Number of algorithms structure tables in this request using RespAlgStruct
`3`	`Param2`	1	Reserved
`4`	`Length`	2	Length of the response message, in bytes.
`6`	`MeasurementSpecificationSel`	1	Bit mask. The Responder shall select one of the measurement specifications supported by the Requester. Thus, no more than one bit shall be set. The `MeasurementSpecification` field of the Measurement block format table defines the values in this field.
`7`	`Reserved`	1	Reserved

Offset	Field	Size (bytes)	Value
`8`	`MeasurementHashAlgo`	4	Bit mask indicating the SPDM-enumerated hashing algorithms used for measurements. Bit 0. Raw Bit Stream Only Bit 1. TPM_ALG_SHA_256 Bit 2. TPM_ALG_SHA_384 Bit 3. TPM_ALG_SHA_512 Bit 4. TPM_ALG_SHA3_256 Bit 5. TPM_ALG_SHA3_384 Bit 6. TPM_ALG_SHA3_512 Bit 7. TPM_ALG_SM3_256 Note that different measurement indices may use different hashing algorithms and/or a raw bit stream. If the Responder supports `GET_MEASUREMENTS`, then the Responder shall set all applicable bits. If the Responder does not support `GET_MEASUREMENTS`, then the Responder shall set this field to `0`.
`12`	`BaseAsymSel`	4	Bit mask indicating the SPDM-enumerated asymmetric key signature algorithm selected for the purpose of signature generation. If the capabilities do not support this algorithm, this value is not used and shall be set to zero. The Responder shall set no more than one bit.
`16`	`BaseHashSel`	4	Bit mask indicating the SPDM-enumerated hashing algorithm selected. If the capabilities do not support this algorithm, this value is not used and shall be set to zero. The Responder shall set no more than one bit.
`20`	`Reserved`	12	Reserved
`32`	`ExtAsymSelCount`	1	Number of extended asymmetric key signature algorithms selected for the purpose of signature generation. Shall be either `0` or `1` (=A'). If the capabilities do not support this algorithm, this value is not used and shall be set to zero.
`33`	`ExtHashSelCount`	1	The number of extended hashing algorithms selected. Shall be either `0` or `1` (=E'). If the capabilities do not support this algorithm, this value is not used and shall be set to zero.
`34`	`Reserved`	2	Reserved.

Offset	Field	Size (bytes)	Value
`36`	`ExtAsymSel`	4*A'	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'	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 Response AlgStructure field format

235 `AlgStructSize` is the sum of the size of all Algorithm structure tables, as the following tables show. The algorithm structure table need be present only if the Responder supports that `AlgType`. `AlgType` shall monotonically increase for subsequent entries.

236 **Response AlgStructure field format**

Offset	Field	Size (bytes)	Value
0	`AlgType`	1	Type of algorithm. 0 and 1 = Reserved 2 = DHE 3 = AEADCipherSuite 4 = ReqBaseAsymAlg 5 = KeySchedule All other values reserved.
1	`AlgCount`	1	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	Bit mask for indicating a Requester-supported, Responder-selected, SPDM-enumerated algorithm. Responder shall set at most one bit to 1.

Offset	Field	Size (bytes)	Value
2 + FixedAlgCount	`AlgExternal`	4*ExtAlgCount'	If present: a Requester-supported, Responder-selected, extended algorithm. Responder shall select at most one external algorithm. The Extended algorithm field format table describes the format of this field.

237 The tables for each of the individual type with the associated fixed fields are described below.

238 DHE structure

Offset	Field	Size (bytes)	Value
0	`AlgType`	1	`0x2=DHE`
1	`AlgCount`	1	Bit [7:4] = 2. Bit [3:0] = Number of Requester-supported, Responder-selected, extended DHE groups (= ExtAlgCount2'). This value shall be either 0 or 1.
2	`AlgSupported`	2	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. 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) (D = 64, C = 32) All other values reserved.
4	`AlgExternal`	4*ExtAlgCount2'	If present: a Requester-supported, Responder-selected, extended DHE algorithm. The Extended algorithm field format table describes the format of this field.

239 AEAD structure

Offset	Field	Size (bytes)	Value
0	`AlgType`	1	`0x3=AEAD`

Offset	Field	Size (bytes)	Value
1	`AlgCount`	1	Bit [7:4] = 2. Bit [3:0] = Number of Requester-supported, Responder-selected, extended AEAD algorithms (= ExtAlgCount3'). This value shall be either 0 or 1.
2	`AlgSupported`	2	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: a Requester-supported, Responder-selected, extended AEAD algorithm. The Extended algorithm field format table describes the format of this field.

240 **ReqBaseAsymAlg structure**

Offset	Field	Size (bytes)	Value
0	`AlgType`	1	`0x4=ReqBaseAsymAlg`
1	`AlgCount`	1	Bit [7:4] = 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.

Offset	Field	Size (bytes)	Value
2	`AlgSupported`	2	Bit mask for indicating a Requester-supported, Responder-selected, SPDM-enumerated asymmetric key signature algorithm for the purpose of signature verification. 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.
4	`AlgExternal`	4*ExtAlgCount4'	If present: a Requester-supported, Responder-selected extended asymmetric key signature algorithm for the purpose of signature verification. The Extended algorithm field format table describes the format of this field.

241 **KeySchedule structure**

Offset	Field	Size (bytes)	Value
0	`AlgType`	1	`0x5=KeySchedule`
1	`AlgCount`	1	Bit [7:4] = 2. Bit [3:0] Number of Requester-supported, Responder-selected, extended key schedule algorithms (= ExtAlgCount5'). This value shall be either 0 or 1.

Offset	Field	Size (bytes)	Value
2	`AlgSupported`	2	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: a Requester-supported, Responder-selected, extended key schedule algorithm. The Extended algorithm field format table describes the format of this field.

242 **Extended Algorithm field format**

Offset	Field	Description
0	Registry ID	Shall represent the registry or standards body. The **ID** column in the Registry or standards body ID table describes the value of this field.
1	Reserved	Reserved
[2:3]	Algorithm ID	Shall indicate the desired algorithm. The registry or standards body owns the value of this field. For details, see the Registry or standards body ID table.

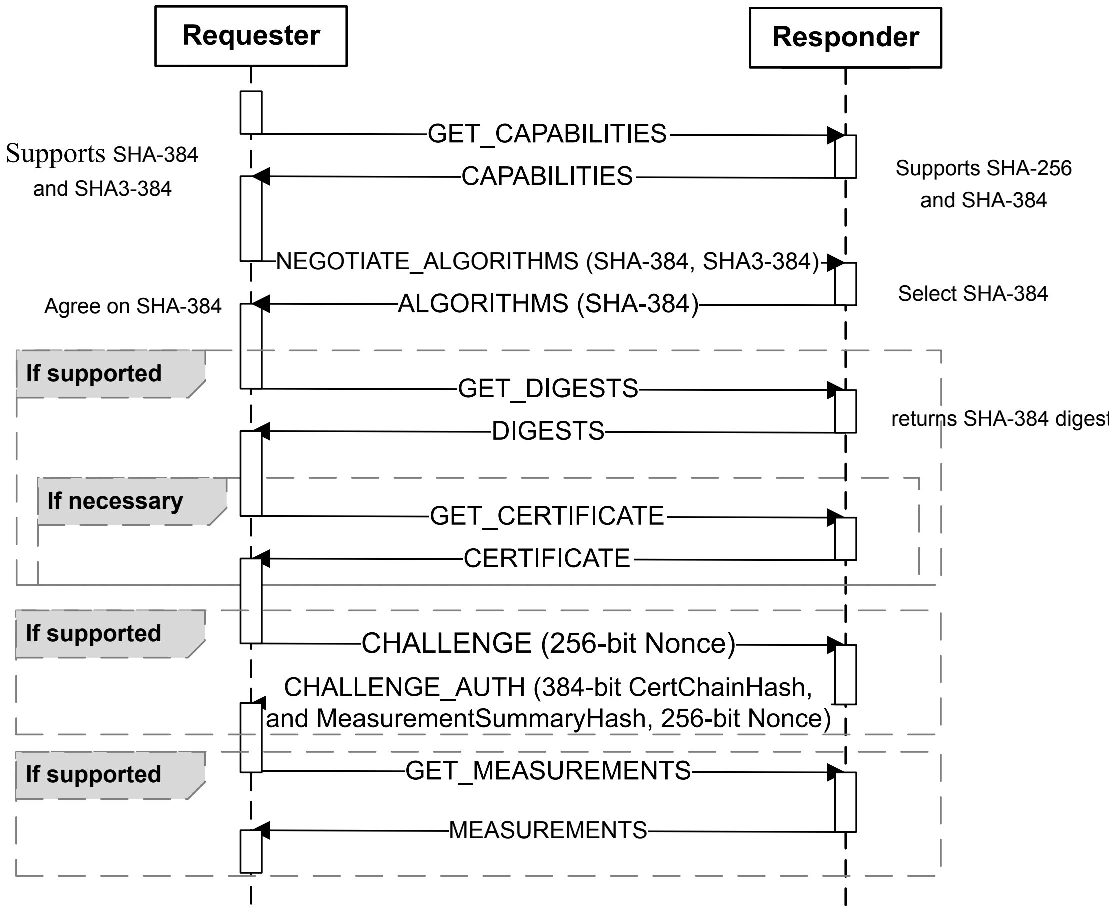
243 For each algorithm type, a Responder shall not select both an SPDM-enumerated algorithm and an extended algorithm.

244 Hashing algorithm selection: Example 1 illustrates how two endpoints negotiate a base hashing algorithm.

245 In Hashing algorithm selection: Example 1, endpoint A issues `NEGOTIATE_ALGORITHMS` request message and endpoint B selects an algorithm of which both endpoints are capable.

246 **Hashing algorithm selection: Example 1**

247



248 The SPDM protocol accounts for the possibility that both endpoints may issue `NEGOTIATE_ALGORITHMS` request messages independently of each other. In this case, the endpoint A Requester and endpoint B Responder communication pair may select a different algorithm compared to the endpoint B Requester and endpoint A Responder communication pair.

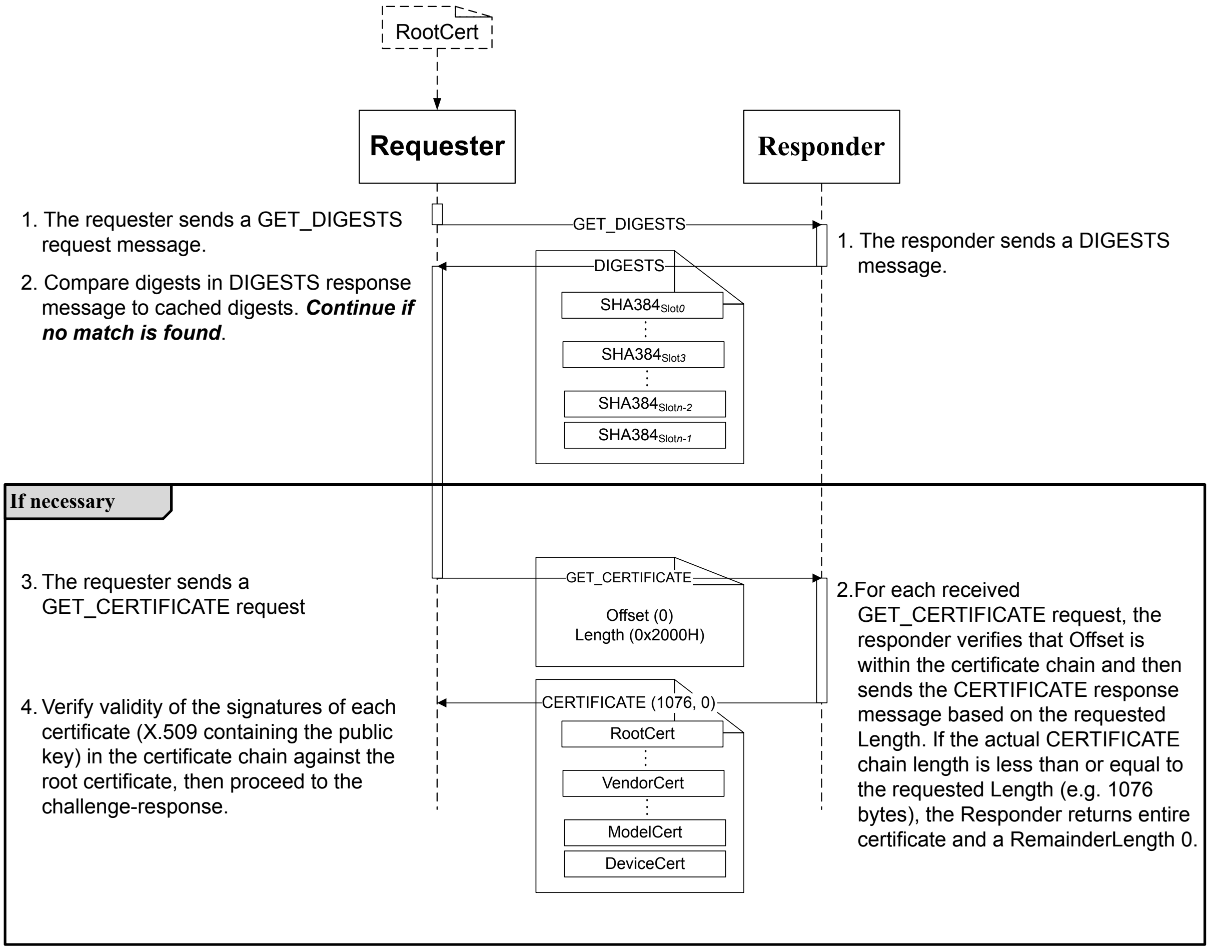
249 10.5 Responder identity authentication

250 This clause describes request messages and response messages associated with the identity of the Responder authentication operations. The `GET_DIGESTS` and `GET_CERTIFICATE` messages shall be supported by a Responder that returns `CERT_CAP` =1 in the `CAPABILITIES` response message. The `CHALLENGE` message defined in this clause shall be supported by a Responder that returns `CHAL_CAP` =1 in the `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.

251 The Responder authentication: Example certificate retrieval flow shows the high-level request-response message flow and sequence for *certificate* retrieval.

252 **Responder authentication: Example certificate retrieval flow**

253



254 The `GET_DIGESTS` request message and `DIGESTS` response message may 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 each of the certificate chains stored on an endpoint is returned with the `DIGESTS` response message, such that the Requester can cache the previously retrieved certificate chain hash values to detect any change to the certificate chains stored on the device before issuing the `GET_CERTIFICATE` request message.

255 For the runtime challenge-response flow, the signature field in the `CHALLENGE_AUTH` response message payload shall be signed by using the private key associated with the leaf certificate over the hash of the message transcript. See the Request ordering and message transcript computation rules for M1/M2 table.

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

257 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 with the latest `GET_VERSION` request received.

258 10.6 Requester identity authentication

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

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

261 10.6.1 Certificates and certificate chains

262 Each SPDM endpoint that supports identity authentication using certificates shall carry at least one certificate chain. A certificate chain contains an ordered list of certificates, presented as the binary (byte) concatenation of the fields that the Certificate chain format shows.

263 Each certificate shall be in ASN.1 DER-encoded X.509 v3 format. The ASN.1 DER encoding of each individual certificate can be analyzed to determine its length. The minimum number of certificates within a chain shall be one, in which case the single certificate is the `DeviceCert` certificate. The SPDM endpoint shall contain a single public-private key pair per supported algorithm for its leaf certificate, regardless of how many certificate chains are stored on the device. The Responder selects a single asymmetric key signature algorithm per Requester.

264 Certificate chains are stored in locations called slots. Each slot shall either be empty or contain one complete certificate chain. A device shall not contain more than eight slots, numbered zero through seven inclusive. Slot 0 is populated by default. If a device uses `AliasCert`s, each certificate chain shall include the `AliasCert`s. Additional slots may be populated through the supply chain such as by a platform integrator or by an end user such as the IT administrator. A slot mask identifies the certificate chains from the eight slots.

265 In this document, `H` refers to the output size, in bytes, of the hash algorithm agreed upon in `NEGOTIATE_ALGORITHMS`.

266 **Certificate chain format**

Offset	Field	Size	Description
0	`Length`	2	Total length of the certificate chain, in bytes, including all fields in this table. This field is little endian.
2	`Reserved`	2	Reserved.
4	`RootHash`	H	Digest of the Root Certificate. Note that Root Certificate is ASN.1 DER-encoded for this digest. This field shall be in Hash byte order.
4 + H	`Certificates`	Length - (4 + H)	One or more ASN.1 DER-encoded X.509 v3 certificates where the first certificate is signed by the Root Certificate or is the Root Certificate itself and each subsequent certificate is signed by the preceding certificate. The last certificate is the *leaf certificate*. This field shall be in Encoded ASN.1 byte order.

267 10.7 GET_DIGESTS request and DIGESTS response messages

268 This request message shall be used to retrieve the certificate chain digests.

269 The GET_DIGESTS request message format table shows the `GET_DIGESTS` request message format.

270 The Successful DIGESTS response message table shows the `DIGESTS` response message format.

271 The digests in the Successful DIGESTS response message table shall be computed over the certificate chain as shown in Certificate chain format.

272 **GET_DIGESTS request message format**

Offset	Field	Size (bytes)	Value
0	`SPDMVersion`	1	Shall be the `SPDMVersion` as described in SPDM version.
1	`RequestResponseCode`	1	`0x81=GET_DIGESTS`
2	`Param1`	1	Reserved
3	`Param2`	1	Reserved

273 **Successful DIGESTS response message format**

Offset	Field	Size (bytes)	Value
0	`SPDMVersion`	1	Shall be the `SPDMVersion` as described in SPDM version.
1	`RequestResponseCode`	1	`0x01=DIGESTS`

Offset	Field	Size (bytes)	Value
2	`Param1`	1	Reserved
3	`Param2`	1	Slot mask. The bit in position K of this byte shall be set to 1b if and only if slot number K contains a certificate chain for the protocol version in the SPDMVersion field. (Bit 0 is the least significant bit of the byte.) The number of digests returned shall be equal to the number of bits set in this byte. The digests shall be returned in order of increasing slot number.
4	`Digest[0]`	H	Digest of the first certificate chain. This field shall be in Hash byte order.
...	...	...	...
4 + (H * (n -1))	`Digest[n-1]`	H	Digest of the last (n^{th}) certificate chain. This field shall be in Hash byte order.

274 10.8 GET_CERTIFICATE request and CERTIFICATE response messages

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

276 The GET_CERTIFICATE request message format table shows the `GET_CERTIFICATE` request message format.

277 The Successful CERTIFICATE response message table shows the `CERTIFICATE` response message format.

278 The Requester should, at a minimum, save the public key of the leaf certificate and associate it with each of the digests returned by `DIGESTS` message response. The Requester sends one or more `GET_CERTIFICATE` requests to retrieve the certificate chain of the Responder.

279 **GET_CERTIFICATE request message format**

Offset	Field	Size (bytes)	Value
0	`SPDMVersion`	1	Shall be the `SPDMVersion` as described in SPDM version.
1	`RequestResponseCode`	1	`0x82=GET_CERTIFICATE`
2	`Param1`	1	Bit [7:4] = Reserved. Bit[3:0] = `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	Reserved

280 **Successful CERTIFICATE response message format**

Offset	Field	Size (bytes)	Value
0	`SPDMVersion`	1	Shall be the `SPDMVersion` as described in SPDM version.
1	`RequestResponseCode`	1	`0x02=CERTIFICATE`
2	`Param1`	1	Bit [7:4] = Reserved. Bit[3:0] = `SlotID` . Slot number of the certificate chain returned.
3	`Param2`	1	Reserved.
4	`CertChain`	Variable	Requested contents of target certificate chain, as described in Certificates and certificate chains.

281 10.8.1 Mutual authentication requirements for GET_CERTIFICATE and CERTIFICATE messages

282 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 GET_CERTIFICATE request and CERTIFICATE response messages clause shall also apply to the Responder. These two statements essentially describe a role reversal.

283 10.8.2 Leaf certificate

284 The SPDM endpoints for authentication shall be provisioned with DER-encoded X.509 v3 format certificates. For endpoint devices to verify the certificate, the following required fields shall be present. In addition, to provide device information, use the `Subject Alternative Name` certificate extension `otherName` field. See the Definition of otherName using the DMTF OID.

285 **Required fields**

Field	Description
`Version`	Version of the encoded certificate shall be present and shall be `3` (encoded as value `2`).
`Serial Number`	CA-assigned serial number shall be present with a positive integer value.
`Signature Algorithm`	Signature algorithm that CA uses shall be present.
`Issuer`	CA distinguished name shall be specified.

Field	Description
`Subject Name`	Subject name shall be present and shall represent the distinguished name associated with the leaf certificate.
`Validity`	See Certificate Validity details below, and RFC5280 for further details.
`Subject Public Key Info`	Device public key and the algorithm shall be present.
`Key Usage`	Shall be present and key usage bit for digital signature shall be set.

286 **Optional fields**

Field	Description
`Basic Constraints`	If present, the CA value shall be `FALSE` in the leaf certificate.
`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.
`Extended Key Usage (EKU)`	If present, the Extended Key Usage extension indicates one or more purposes for which the public key should be used. The following Extended Key Usage purposes are defined for SPDM certificate authentication: `SPDM Responder Authentication (1.3.6.1.4.1.412.274.3)` : The presence of this OID shall indicate that a leaf certificate is used for Responder authentication purposes. `SPDM Requester Authentication (1.3.6.1.4.1.412.274.4)` : The presence of this OID shall indicate that a leaf certificate is used for Requester authentication purposes. The presence of both OIDs shall indicate that the leaf certificate is used for both Requester and Responder authentication purposes. 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.

287 **Certificate Validity details**

288 As per RFC5280, 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 a ASN.1-encoded SEQUENCE of two dates: the date on which the certificate validity period begins (`notBefore`) and the date on which the certificate validity period ends (`notAfter`).

289 For all `DeviceCert` leaf certificates (which are immutable) as well as the 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`. In general, 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.

290 For `AliasCert` leaf certificates as well as leaf certificates whose chains are stored in Slots 1-7, the `notBefore` date should be the date of certificate creation. The `notAfter` date may be set according to end user requirements, including values that will cause certificate expiration and necessitate certificate renewal, and thus device re-certification, during the lifetime of the device.

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

```
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})
```

292 The Leaf certificate example shows an example leaf certificate.

293 10.9 CHALLENGE request and CHALLENGE_AUTH response messages

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

295 The CHALLENGE request message format table shows the `CHALLENGE` request message format.

296 The Successful CHALLENGE_AUTH response message table shows the `CHALLENGE_AUTH` response message format.

297 **CHALLENGE request message format**

Offset	Field	Size (bytes)	Value
0	SPDMVersion	1	Shall be the SPDMVersion as described in SPDM version.
1	RequestResponseCode	1	0x83=CHALLENGE
2	Param1	1	SlotID . Slot number of the Responder certificate chain that shall be used for authentication. It shall be 0xFF if the public key of the Responder was provisioned to the Requester in a trusted environment, otherwise the value in this field shall be between 0 and 7 inclusive.
3	Param2	1	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 CAPABILITIES response), the Requester shall set this value to 0x0 .
4	Nonce	32	The Requester should choose a random value.

298 **Successful CHALLENGE_AUTH response message format**

Offset	Field	Size (bytes)	Value
0	SPDMVersion	1	Shall be the SPDMVersion as described in SPDM version.
1	RequestResponseCode	1	0x03=CHALLENGE_AUTH
2	Param1	1	Response Attribute Field. Please see CHALLENGE_AUTH Response Attribute Table for details.
3	Param2	1	Slot mask. The bit in position K of this byte shall be set to 1b if and only if slot number K contains a certificate chain for the protocol version in the SPDMVersion field. 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	Hash of the certificate chain or public key (if the public key of the Responder was provisioned to the Requester in a trusted environment) 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.

Offset	Field	Size (bytes)	Value
4 + H	`Nonce`	32	Responder-selected random value.
36 + H	`MeasurementSummaryHash`	H	If the Responder does not support measurements (`MEAS_CAP=00b` in `CAPABILITIES` response) or 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(Concatenation(MeasurementBlock[0], MeasurementBlock[1], ...))` where `MeasurementBlock[x]` denotes a measurement of an element in the TCB. Measurements are concatenated in ascending order based on their measurement index. When the requested `Param2` = `0x1` and there are no measurable components in the TCB required to generate this response, this field shall be `0` . If requested `Param2` = `0xFF` , this field shall be computed as `hash(Concatenation(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 section for details. If the Responder supports both raw bit stream and digest representations for a given measurement index, then the Responder shall use the digest form. This field shall be in Hash byte order.
36 + 2H	`OpaqueDataLength`	2	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 + 2H	`OpaqueData`	`OpaqueDataLength`	The Responder may include Responder-specific information and/or information defined by its transport. If present, this field shall conform to the General opaque Data Format.
38 + 2H + `OpaqueDataLength`	`Signature`	`SigLen`	`SigLen` is the size of the asymmetric-signing algorithm output that the Responder selected through 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.

299 **CHALLENGE_AUTH response attribute**

Bit Offset	Field Name	Description
[3:0]	SlotID	This field 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 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 for more details. If mutual authentication is not supported, this bit shall be zero; otherwise, it should be considered an error.

300 10.9.1 CHALLENGE_AUTH signature generation

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

302 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.

303 where:

304 `Concatenate()` is the standard concatenation function that is performed only after a successful completion response on the entire request and response contents.

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

306 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.

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

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

309 2. The Responder shall generate:

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

310 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.

311 10.9.2 CHALLENGE_AUTH signature verification

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

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

314 1. The Requester shall perform:

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

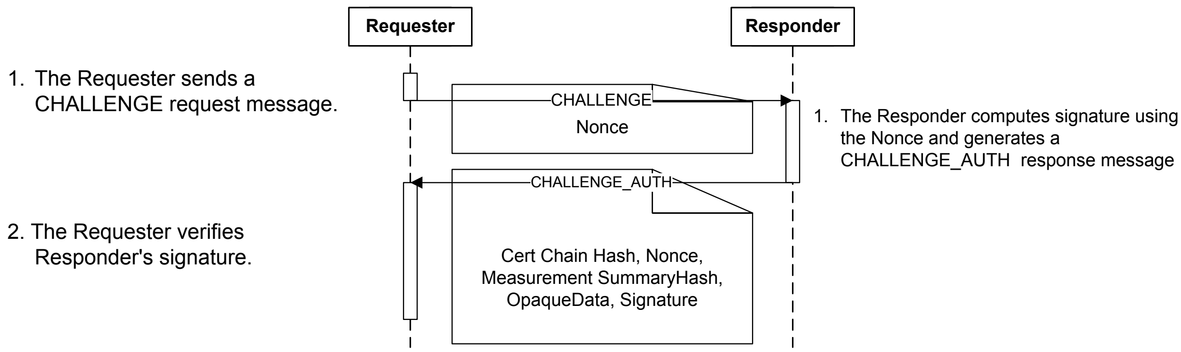
315 where:

- `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 of the Responder with `slot=Param1` of the `CHALLENGE` request message. If the public key of the Responder was provisioned to the Requester, then `PK` is the provisioned public key.

316 The 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.

317 **Responder authentication: Runtime challenge-response flow**

318



319 **10.9.2.1 Request ordering and message transcript computation rules for M1 and M2**

320 This clause applies to Responder-only authentication.

321 The Request ordering and message transcript computation rules for M1/M2 table 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.

322 The possible request orderings after Reset 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` , `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)

323 Immediately after Reset, M1 and M2 shall be null.

324 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.

325 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.

326 For additional rules, see General ordering rules.

327 **Request ordering and message transcript computation rules for M1/M2**

Requests	Implementation requirements	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 if the Requester detects an out of sync condition, 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	Requester skipped issuing these requests after a Reset or a completed `CHALLENGE_AUTH` response, that caused M1/M2 to re-initialize to `null` , if the Responder has 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	Requester issued these requests in this order after `NEGOTIATE_ALGORITHMS` request completion, or after a Reset or a completed `CHALLENGE_AUTH` response, that caused M1/M2 to re-initialize to `null` , if it chose to skip the previous three requests.	`B1=Concatenate(GET_DIGESTS, DIGESTS, GET_CERTFICATE, CERTIFICATE)`
`GET_DIGESTS` , `GET_CERTIFICATE` skipped	Requester skipped both requests after a Reset or a completed `CHALLENGE_AUTH` response, that caused M1/M2 to re-initialize to `null` , since it could use previously cached certificate information.	`B2=null`
`GET_DIGESTS` issued, `GET_CERTIFICATE` skipped	Requester skipped `GET_CERTIFICATE` request after a Reset or a completed `CHALLENGE_AUTH` response, that caused M1/M2 to re-initialize to `null` since it could use the previously cached `CERTIFICATE` response.	`B3=(GET DIGESTS, DIGESTS)`
`GET_DIGESTS` skipped, `GET_CERTIFICATE` issued	Requester skipped `GET_DIGEST` request after after a Reset or a completed `CHALLENGE_AUTH` response, that caused M1/M2 to re-initialize to `null` . The Requester uses the previously cached `CERTIFICATE` response for a byte-by-byte comparison.	`B4=(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=(CHALLENGE, CHALLENGE_AUTH\Signature)` . See the CHALLENGE request message format table.
`CHALLENGE` completion	Completion of `CHALLENGE` sets M1/M2 to `null` .	`M1/M2=null`
Other issued	If the Requester issued `GET_MEASUREMENTS` or `KEY_EXCHANGE` or `FINISH` or `PSK_EXCHANGE` or `PSK_FINISH` or `KEY_UPDATE` or `HEARTBEAT` or `GET_ENCAPSULATED_REQUEST` or `DELIVER_ENCAPSULATED_RESPONSE` or `END_SESSSION` request(s) and skipped `CHALLENGE` completion, M1/M2 are set to `null` .	`M1/M2=null`

328 The Basic mutual authentication flow is DEPRECATED. Implementations should use Session-based mutual authentication or Optimized Session-based mutual authentication.

---

329 DEPRECATED

330 10.9.3 Basic mutual authentication

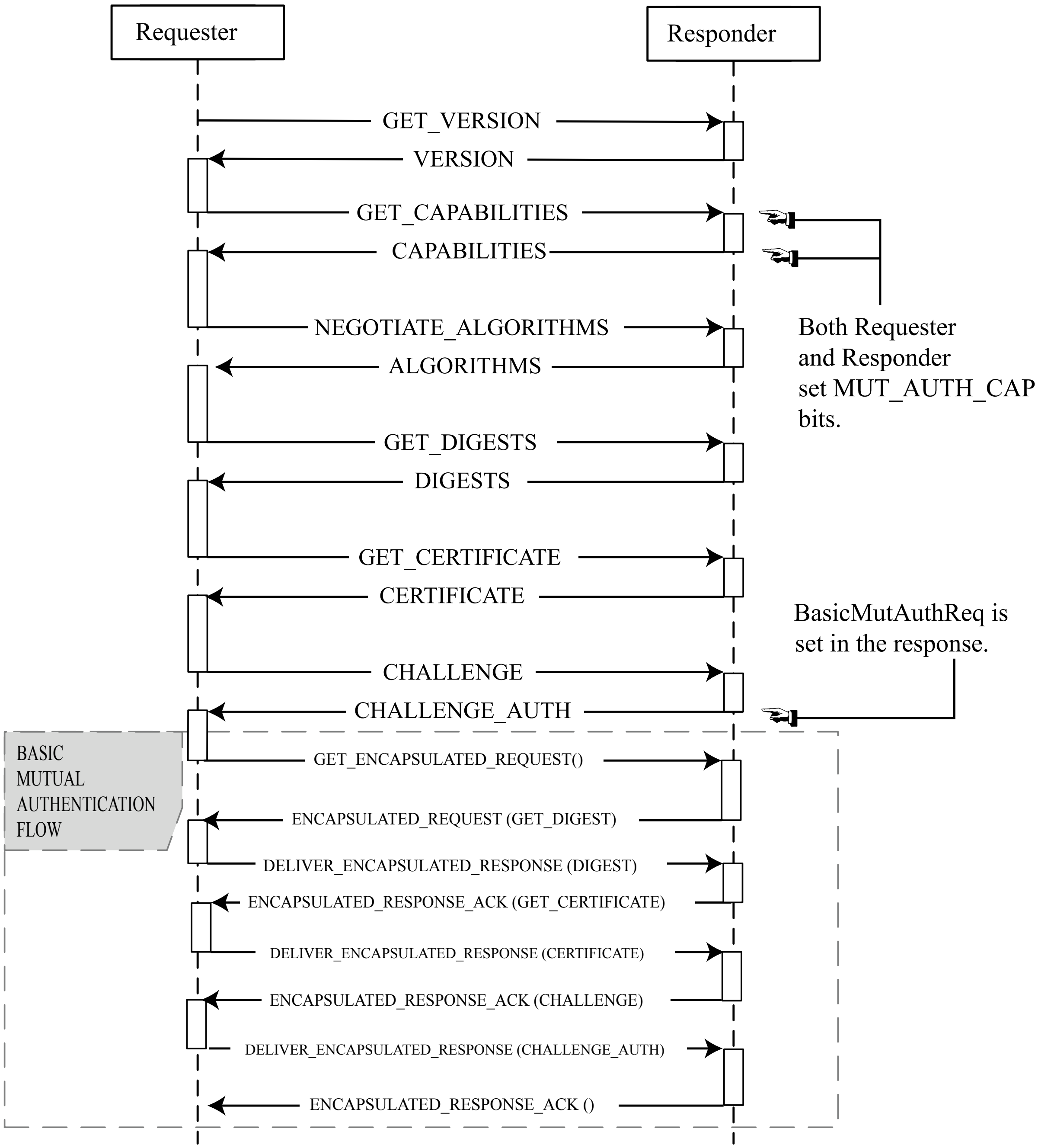
331 Unless otherwise stated, if the Requester supports mutual authentication, the requirements placed on the Responder in the CHALLENGE request and CHALLENGE_AUTH response messages clause shall also apply to the Requester. Unless otherwise stated, if the Responder supports mutual authentication, the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

332 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 GET_ENCAPSULATED_REQUEST request and ENCAPSULATED_REQUEST response messages to retrieve request messages. The basic mutual authentication flow shall end when the encapsulated request flow ends.

333 This flow shall only allow `GET_DIGESTS`, `GET_CERTIFICATE`, `CHALLENGE` and their corresponding responses to be encapsulated. If other requests are encapsulated, the Requester may send an `ERROR` response with `ErrorCode=UnexpectedRequest` and shall terminate the flow.

334 The Mutual authentication basic flow illustrates, as an example, the basic mutual authentication flow.

335 **Mutual authentication basic flow**

336 

337 **10.9.3.1 Mutual authentication message transcript**

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

339 The Basic mutual authentication message transcript table 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.

340 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*)

341 When the basic mutual authentication flow starts (i.e., when `GET_ENCAPSULATED_REQUEST` is issued) M1 and M2 shall be set to null.

342 **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)

343 For `GET_CERTIFICATE` and `CERTIFICATE`, these messages may 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.

344 DEPRECATED

345
10.10 Firmware and other measurements

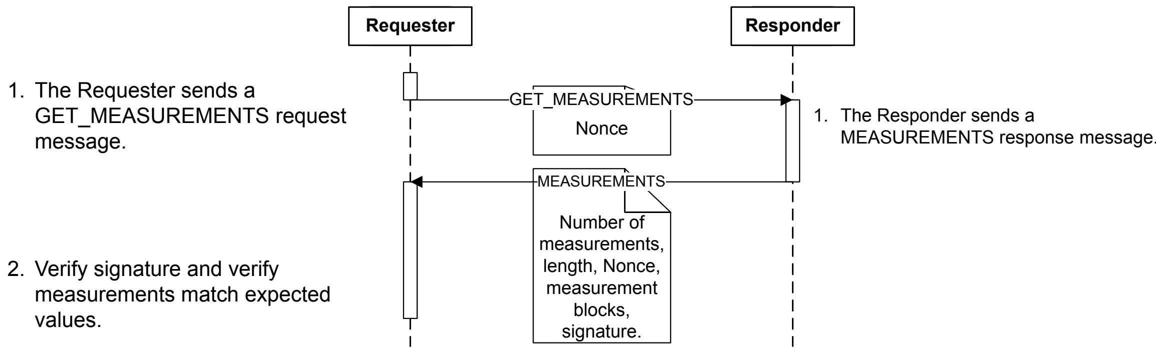
346 This clause describes request messages and response messages associated with endpoint measurement. All request messages in this clause shall be supported by an endpoint that returns `MEAS_CAP=01b` or `MEAS_CAP=10b` in `CAPABILITIES` response.

347 The Measurement retrieval flow shows the high-level request-response flow and sequence for endpoint measurement. If `MEAS_FRESH_CAP` bit in the `CAPABILITIES` response message returns 0, and 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.

348 **Measurement retrieval flow**

349



350 10.11 GET_MEASUREMENTS request and MEASUREMENTS response messages

351 Measurements in SPDM are represented in the form of measurement *blocks*. Measurement block defines the measurement block structure. A device may 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 more than one measurement blocks is called a *measurement record*.

352 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_CERTIFICATES` / `CERTIFICATES` has been authenticated at least once.

353 The GET_MEASUREMENTS request message format table shows the `GET_MEASUREMENTS` request message format.

354 The GET_MEASUREMENTS request attributes table shows the `GET_MEASUREMENTS` request message attributes.

355 The Successful MEASUREMENTS response message format table shows the `MEASUREMENTS` response message format. The measurement blocks in `MeasurementRecord` shall be sorted in ascending order by index.

356 **GET_MEASUREMENTS request message format**

Offset	Field	Size (bytes)	Value
0	`SPDMVersion`	1	Shall be the `SPDMVersion` as described in SPDM version.
1	`RequestResponseCode`	1	`0xE0=GET_MEASUREMENTS`
2	`Param1`	1	Request attributes. See the GET_MEASUREMENTS request attributes table.
3	`Param2`	1	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`, inclusively, shall request the measurement block at the index corresponding to that value.
4	`Nonce`	`NL=32` or `NL=0`	The Requester should choose a random value. This field is only present if Bit [0] of `Param1` is `1`. See the GET_MEASUREMENTS request attributes table.
4 + `NL`	`SlotIDParam`	1	Bit[7:4] = Reserved. Bit[3:0] = `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`. This field is only present if Bit [0] of `Param1` is `1`. See the GET_MEASUREMENTS request attributes table.

357 **GET_MEASUREMENTS request attributes**

Bits	Field	Description
`0`	`SignatureRequested`	If the Responder can generate a signature (`MEAS_CAP` is `10b` in the `CAPABILITIES` response), value of `1` indicates that a signature on the measurement log (L1/L2 defined in MEASUREMENTS signature generation) is required. The `Nonce` field shall be present in the request where this bit is set. The Responder shall generate and send a signature in the response. 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) the Requester shall always use `0`.

Bits	Field	Description
`1`	`RawBitStreamRequested`	This bit is applicable only if the measurement specification supports only two representations, raw bit stream and digest (for example, when `MeasurementSpecification` of Measurement block format is set to DMTF). If the measurement specification supports other representations, this bit is ignored. If the Responder is able to 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 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. Value `0` shall request the Responder to return a hash version of the measurement. If the Responder cannot return hash of the measurement (for example, if the measurement represents a data structure where digest is not applicable), it shall return the corresponding raw bit stream.
`[7:2]`	Reserved	Reserved

358 **Measurement index assignments**

359 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 may use the same index to report different types of measurements based on their implementation. If available, a Requester may use a measurement manifest (a measurement of type `DMTFSpecMeasurementValueType[6:0]` = `0x04` if measurements follow the DMTF measurement specification format) to discover information about the specific measurement types available by a particular Responder and the indices they correspond to.

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

361 **Measurement index assigned range table**

Measurement index	Measurement type	Description
`0xF0` - `0xFC`	Reserved	Reserved
`0xFD`	Measurement manifest	Metadata on available measurements, as defined by type `DMTFSpecMeasurementValueType[6:0]` = `0x04`
`0xFE`	Device mode	Structured device mode information, as defined by type `DMTFSpecMeasurementValueType[6:0]` = `0x05`

362 **Successful MEASUREMENTS response message format**

Offset	Field	Size (bytes)	Value
0	`SPDMVersion`	1	Shall be the `SPDMVersion` as described in SPDM version.
1	`RequestResponseCode`	1	`0x60=MEASUREMENTS`
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 indicates if one or more entries in the measurement log being signed have changed. `00b` : the Responder does not support detection of runtime measurement changes, or this message does not contain a signature. `01b` : the Responder detected that one or more entries in the measurement log being signed have changed. The Requester may consider issuing `GET_MEASUREMENTS` again to acquire current measurements. `10b` : the Responder detected no change in the entries in the measurement log being signed. `11b` : reserved. Bit[3:0] = `SlotID` . If this message contains a signature, this field contains 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	Number of measurement blocks in the full `MeasurementRecord` . If Param2 in the requested measurement operation is `0` , this field shall be `0` .
5	`MeasurementRecordLength`	3	Size of the full `MeasurementRecord` in bytes. If `Param2` in the requested measurement operation is `0` , this field shall be `0` .
8	`MeasurementRecordData`	L= `MeasurementRecordLength`	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.

Offset	Field	Size (bytes)	Value
40 + `L`	`OpaqueDataLength`	2	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 may include Responder-specific information and/or information defined by its transport. If present, this field shall conform to the General opaque Data Format.
42 + `L` + `OpaqueDataLength`	`Signature`	`SigLen`	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 only present in the MEASUREMENTS response corresponding to a GET_MEASUREMENTS request with Param1[0] set to 1.

363 ### 10.11.1 Measurement block

364 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 correspond to a particular measurement index and measurement type. The blocks are ordered by `Index`.

365 The Measurement block format table shows the format for a measurement block:

366 **Measurement block format**

Offset	Field	Size (bytes)	Value
0	`Index`	1	Index. When `Param2` of `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.

Offset	Field	Size (bytes)	Value
1	`MeasurementSpecification`	1	Bit mask. The value shall indicate the measurement specification that the requested `Measurement` follows and shall match the selected measurement specification in the `ALGORITHMS` message. See the Successful ALGORITHMS response message format table. Only one bit shall be set in the measurement block. Bit 0 = DMTF, as specified in the DMTF measurement specification format table. All other bits are reserved.
2	`MeasurementSize`	2	Size of `Measurement`, in bytes.
4	`Measurement`	MeasurementSize	The `MeasurementSpecification` defines the format of this field.

367 **10.11.1.1 DMTF specification for the Measurement field of a measurement block**

368 The present clause is the specification for the format of the `Measurement` field in a measurement block when the `MeasurementSpecification` field selects DMTF (Bit 0). This format is specified in DMTF measurement specification format table.

369 The measurement manifest of `DMTFSpecMeasurementValueType` refers to a manifest that describes contents of other indexes. For example, the set of firmware modules executing on the Responder may change at runtime. The measurement manifest tells the Requester which firmware modules' measurements are reported in this response and their indexes. The format of measurement manifest is out of scope of this specification.

370 **DMTF measurement specification format**

Offset	Field	Size (bytes)	Value
0	`DMTFSpecMeasurementValueType`	1	Composed of: Bit [7] indicates the representation in `DMTFSpecMeasurementValue`. Bits [6:0] indicate what is being measured by `DMTFSpecMeasurementValue`. These values are set independently and are interpreted as follows: `[7]=0b`. Digest. `[7]=1b`. Raw bit stream. Note: the Responder shall make sure the raw bit stream does not contain secrets. `[6:0]=00h`. Immutable ROM. `[6:0]=01h`. Mutable firmware. `[6:0]=02h`. Hardware configuration, such as straps. `[6:0]=03h`. Firmware configuration, such as configurable firmware policy. `[6:0]=04h`. Measurement manifest. When `DMTFSpecMeasurementValueType[6:0]=04h`, the Responder should support setting `DMTFSpecMeasurementValueType[7]` to either `0b` or `1b`. `[6:0]=05h`. Structured representation of debug and device mode. See Device mode field of a measurement block. When `DMTFSpecMeasurementValueType[6:0]=05h`, `DMTFSpecMeasurementValueType[7]` shall be set to `1b`. `[6:0]=06h`. Mutable firmware's version number. This specification does not mandate a format for firmware version number. When `DMTFSpecMeasurementValueType[6:0]=06h`, `DMTFSpecMeasurementValueType[7]` should be set to `1b`. `[6:0]=07h`. Mutable firmware's security version number, which should be formatted as an 8-byte unsigned integer. When `DMTFSpecMeasurementValueType[6:0]=07h`, `DMTFSpecMeasurementValueType[7]` should be set to `1b`. All other values reserved.

Offset	Field	Size (bytes)	Value
1	`DMTFSpecMeasurementValueSize`	2	The value of this field indicates the format and size of `DMTFSpecMeasurementValue`. The possible values for this field shall be these values: `0x0000` : Raw Bit Stream (`MS=MeasurementSize - 3`). `MeasurementSize` is a field in Measurement block. `0x0001` : TPM_ALG_SHA_256 (`MS=32`) `0x0002` : TPM_ALG_SHA_384 (`MS=48`) `0x0003` : TPM_ALG_SHA_512 (`MS=64`) `0x0004` : TPM_ALG_SHA3_256 (`MS=32`) `0x0005` : TPM_ALG_SHA3_384 (`MS=48`) `0x0006` : TPM_ALG_SHA3_512 (`MS=64`) `0x0007` : TPM_ALG_SM3_256 (`MS=32`) `0x0008 - 0xFFFF` : Reserved
3	`DMTFSpecMeasurementValue`	`MS`	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 byte order for raw bit streams is vendor defined.

371 ### 10.11.1.2 Device mode field of a measurement block

Offset	Field	Size (bytes)	Value
0	`OperationalModeCapabilties`	4	Fields with bits set to 1 indicate support for reporting the associated state in `OperationalModeState`. • Bit [0] Indicates support for reporting device in manufacturing mode. • Bit [1] Indicates support for reporting device in validation mode. • Bit [2] Indicates support for reporting device in normal operational mode. • Bit [3] Indicates support for reporting device in RMA mode. • Bit [4] Indicates 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] Indicates the device is in manufacturing mode. • Bit [1] Indicates the device is in validation mode. • Bit [2] Indicates the device is in normal operational mode. • Bit [3] Indicates the device is in RMA mode. • Bit [4] Indicates the device is in decommissioned mode. All other values reserved.

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

372 10.11.2 MEASUREMENTS signature generation

373 While a Requester may 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:

374 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)
```

375 where:

376 ◦ `Concatenate()`

377 Standard concatenation function.

378 ◦ `GET_MEASUREMENTS_REQUEST1`

379 Entire first `GET_MEASUREMENTS` request message under consideration, where the Requester has not requested a signature on that specific `GET_MEASUREMENTS` request.

380 ◦ `MEASUREMENTS_RESPONSE1`

381 Entire `MEASUREMENTS` response message without the signature bytes that the Responder sent in response to `GET_MEASUREMENTS_REQUEST1`.

382 ◦ `GET_MEASUREMENTS_REQUESTn-1`

383 Entire last consecutive `GET_MEASUREMENTS` request message under consideration, where the Requester has not requested a signature on that specific `GET_MEASUREMENTS` request.

384 ◦ `MEASUREMENTS_RESPONSEn-1`

385 Entire `MEASUREMENTS` response message without the signature bytes that the Responder sent in response to `GET_MEASUREMENTS_REQUESTn-1`.

386 ◦ `GET_MEASUREMENTS_REQUESTn`

387 Entire first `GET_MEASUREMENTS` request message under consideration, where the Requester has requested a signature on that specific `GET_MEASUREMENTS` request.

388 *n* is a number greater than or equal to `1`.

389 When *n* equals `1`, the Requester has not made any `GET_MEASUREMENTS` requests without signature prior to issuing a `GET_MEASUREMENTS` request with signature.

390 ◦ `MEASUREMENTS_RESPONSEn`

391 Entire `MEASUREMENTS` response message without the signature bytes that the Responder sent in response to `GET_MEASUREMENTS_REQUESTn`.

392 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.

393 An error response with `ErrorCode=ResponseNotReady` shall not re-initialize L1/L2 - Requester and

Responder shall continue to construct L1/L2 with `GET_MEASUREMENTS` and `MEASUREMENTS`. An error response with any error code other than `ResponseNotReady` shall re-initialize L1/L2 to null.

394 2. The Responder shall generate:

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

395 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.

396 ### 10.11.3 MEASUREMENTS signature verification

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

398 1. The Requester shall perform:

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

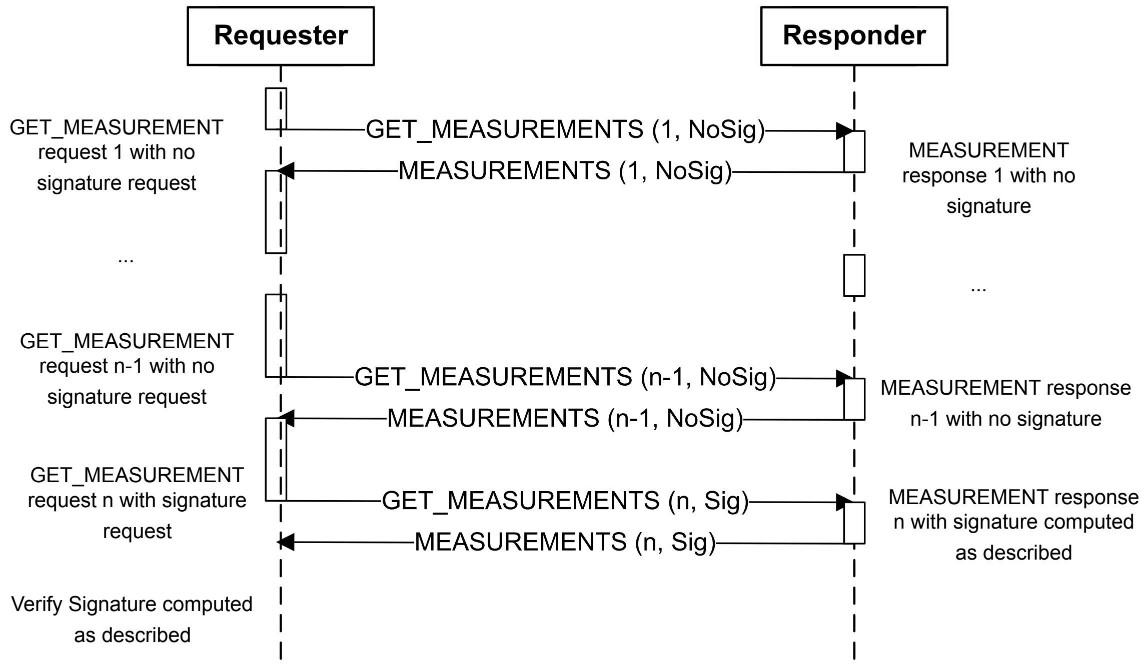
399 where:

- `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 `CERTIFICATES` response. If the public key of the Responder was provisioned to the Requester, then `PubKey` shall be the provisioned public key.

400 The 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, followed by a single `GET_MEASUREMENTS` request *n* with a signature.

401 **Measurement signature computation example**

402



403 10.12 ERROR response message

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

405 The ERROR response message format table shows the `ERROR` response format.

406 The Error code and error data table shows the detailed error code, error data, and extended error data.

407 The ResponseNotReady extended error data table shows the `ResponseNotReady` extended error data.

408 The Registry or standards body ID table shows the registry or standards body ID.

409 The ExtendedErrorData format for vendor or other standards-defined ERROR response message table shows the `ExtendedErrorData` format definition for vendor or other standards-defined `ERROR` response message.

410 **ERROR response message format**

Offset	Field	Size (bytes)	Value
0	SPDMVersion	1	Shall be the SPDMVersion as described in SPDM version.
1	RequestResponseCode	1	0x7F=ERROR
2	Param1	1	Error Code. See Error code and error data.
3	Param2	1	Error Data. See Error code and error data.
4	ExtendedErrorData	0-32	Optional extended data. See Error code and error data.

411 **Error code and error data**

Error code	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 may 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 of the record cannot decrypt the record or verify data during the session handshake.	Reserved	No extended error data is provided.
UnsupportedRequest	0x07	The RequestResponseCode in the request message is unsupported.	RequestResponseCode 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.

Error code	Value	Description	Error data	ExtendedErrorData
`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 `GENERATE_KEY` or `SET_CERTIFICATE` message.	0x00	No extended error data is provided.
`ResponseTooLarge`	`0x0D`	The response is greater than the `MaxSPDMmsgSize` of the requesting SPDM endpoint.	Reserved	See ExtendedErrorData 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` of the requesting SPDM endpoint.	Reserved	See ExtendedErrorData for LargeResponse.
Reserved	`0x10` - `0x40`	Reserved	Reserved	Reserved
`MajorVersionMismatch`	`0x41`	Requested SPDM Major Version is not supported.	`0x00`	No extended error data is provided.
`ResponseNotReady`	`0x42`	See the RESPOND_IF_READY request message format.	`0x00`	See the ResponseNotReady extended error data table.
`RequestResynch`	`0x43`	Responder is requesting Requester to reissue `GET_VERSION` to resynchronize. An example is following a firmware update.	`0x00`	No extended error data is provided.
Reserved	`0x44` - `0xFE`	Reserved	Reserved.	Reserved
Vendor/Other Standards Defined	`0xFF`	Vendor or Other Standards defined	Shall indicate the registry or standard body using one of the values in the **ID** column in the Registry or standards body ID table.	See the ExtendedErrorData format for vendor or other standards-defined ERROR response message table for format definition.

412 **ResponseNotReady extended error data**

Offset	Field	Size (bytes)	Value
0	`RDTExponent`	1	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. Responder should use `RDT` to avoid continuous pinging and issue the `RESPOND_IF_READY` request message after `RDT` time. For timing requirement details, see the Timing specification for SPDM messages table.
1	`RequestCode`	1	The request code that triggered this response.
2	`Token`	1	The opaque handle that the Requester shall pass in with the `RESPOND_IF_READY` request message. 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	Multiplier used to compute WT_{Max} in μs to indicate the response may be dropped after this delay. The multiplier shall always be greater than 1. The Responder may also stop processing the initial request if the same Requester issues a different request. For timing requirement details, see the Timing specification for SPDM messages table.

413 **Registry or standards body ID**

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

ID	Vendor ID length (bytes)	Registry or standards body name	Description
`0x0`	0	DMTF	DMTF does not have a Vendor ID registry. At present, DMTF does not have any algorithms defined for use in extended algorithms fields.
`0x1`	2	TCG	Vendor is identified by using TCG Vendor ID Registry. For extended algorithms, see TCG Algorithm Registry.
`0x2`	2	USB	Vendor is identified by using the vendor ID assigned by USB.
`0x3`	2	PCI-SIG	Vendor is identified using PCI-SIG Vendor ID.

ID	Vendor ID length (bytes)	Registry or standards body name	Description
`0x4`	4	IANA	The Private Enterprise Number (PEN) assigned by the Internet Assigned Numbers Authority (IANA) identifies the vendor.
`0x5`	4	HDBaseT	Vendor is identified by using HDBaseT HDCD entity.
`0x6`	2	MIPI	The Manufacturer ID assigned by MIPI identifies the vendor.
`0x7`	2	CXL	Vendor is identified by using CXL vendor ID.
`0x8`	2	JEDEC	Vendor is identified by using JEDEC vendor ID.

415 **ExtendedErrorData format for vendor or other standards-defined ERROR response message**

Byte offset	Length	Field name	Description
0	1	Len	Length of the `VendorID` field. If the `ERROR` is vendor defined, the value of this field shall equal the `Vendor ID Len`, as the Registry or standards body ID table describes, of the corresponding registry or standard body name. If the `ERROR` is defined by a registry or a standard, 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, such as `Param2`, and is one of the values in the **ID** column in the Registry or standards body ID table.
1	Len	VendorID	The value of this field shall indicate the Vendor ID, as assigned by the registry or standards body. The Registry or standards body ID table describes the length of this field. Shall be in little endian format. The registry or standards body name in the `ERROR` is indicated in the `Error Data` field, such as `Param2`, and is one of the values in the **ID** column in the Registry or standards body ID table.
1 + Len	Variable	OpaqueErrorData	Defined by the vendor or other standards.

416 **ExtendedErrorData format for ResponseTooLarge**

Byte offset	Length	Field name	Description
0	4	ActualSize	The size of the actual Response.

417 **ExtendedErrorData format for LargeResponse**

Byte offset	Length	Field name	Description
0	1	Handle	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 entirely monotonically increase or entirely monotonically 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. See CHUNK_GET request and CHUNK_RESPONSE response message for details.

418 10.12.1 Standard body or vendor-defined header

419 The Standard body or vendor-defined header (SVH) format is used in numerous places in this specification to help identify the entity that defined the format for a given payload. The clauses in the other parts of this specification will indicate which payload this header applies to.

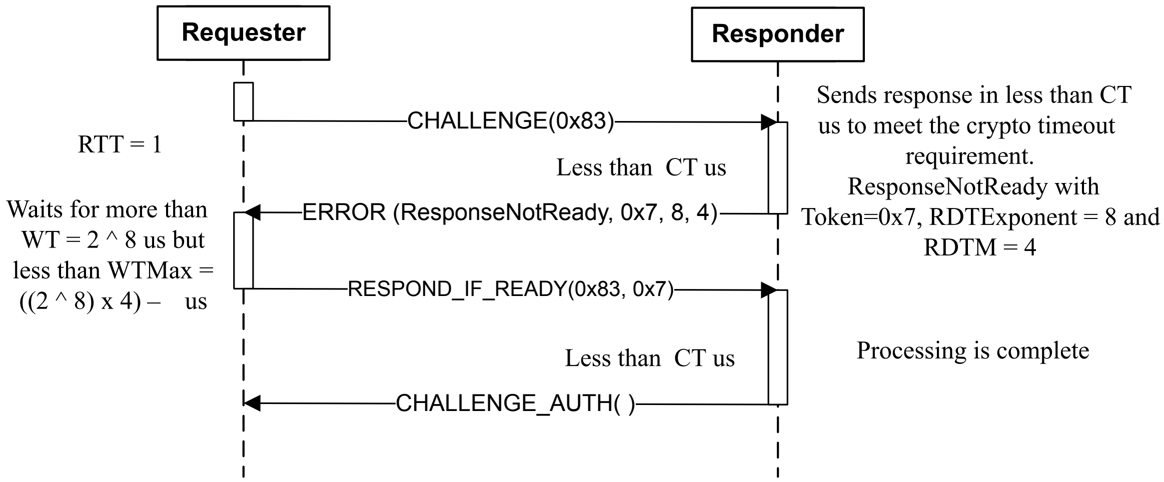
420 **Standard body or vendor-defined header (SVH)**

Offset	Field	Length (bytes)	Description
0	ID	1	Shall be one of the values in the `ID` column of Registry or standards body ID.
1	VendorLen	1	Length in bytes of the `VendorID` field. If the given payload belongs to a standards body, this field shall be 0. Otherwise, the given payload belongs to the vendor and therefore, this field shall be the length indicated in the `Vendor ID` column of Registry and standards body ID table for the respective `ID`.
2	VendorID	VendorLen	If `VendorLen` is greater than zero, this field shall be the ID of the vendor corresponding to the `ID` field. Otherwise, this field shall be absent.

421 10.13 RESPOND_IF_READY request message format

422 This request message shall ask for the response to the original request upon receipt of `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 the `ERROR` response message, set `ErrorCode` = `ResponseNotReady` and return the same token as the previous `ResponseNotReady` response message.

423



424 The RESPOND_IF_READY request message format table shows the `RESPOND_IF_READY` request message format.

425 **RESPOND_IF_READY request message format**

Offset	Field	Size (bytes)	Value
0	`SPDMVersion`	1	Shall be the `SPDMVersion` as described in SPDM version.
1	`RequestResponseCode`	1	`0xFF=RESPOND_IF_READY`
2	`Param1`	1	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	The token that was returned as part of the `ResponseNotReady` extended error data.

426 10.14 VENDOR_DEFINED_REQUEST request message

427 A Requester intending to define a unique request to meet its need can use this request message. The VENDOR_DEFINED_REQUEST request message format table defines the format.

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

429 If the vendor intends that these messages are to be used before a session has been established, and 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.

430 The VENDOR_DEFINED_REQUEST request message format table shows the `VENDOR_DEFINED_REQUEST` request message format.

431 **VENDOR_DEFINED_REQUEST request message format**

Offset	Field	Size (bytes)	Value
0	`SPDMVersion`	1	Shall be the `SPDMVersion` as described in SPDM version.
1	`RequestResponseCode`	1	`0xFE=VENDOR_DEFINED_REQUEST`
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 in the Registry or standards body ID table.
6	`Len`	1	Length of the `Vendor ID` field. If the `VendorDefinedRequest` is standard defined, Len shall be `0`. If the `VendorDefinedRequest` is vendor-defined, Len shall equal `Vendor ID Len`, as the Registry or standards body ID table describes.
7	`VendorID`	Len	Vendor ID, as assigned by the registry or standards body. Shall be in little endian format.
7 + Len	`ReqLength`	2	Length of the `VendorDefinedReqPayload`.
7 + Len + 2	`VendorDefinedReqPayload`	ReqLength	The standard or vendor shall use this field to send the request payload.

432 10.15 VENDOR_DEFINED_RESPONSE response message

433 A Responder can use this response message in response to `VENDOR_DEFINED_REQUEST`. The VENDOR_DEFINED_RESPONSE response message format table defines the format.

434 The VENDOR_DEFINED_RESPONSE response message format table shows the response message format.

435 **VENDOR_DEFINED_RESPONSE response message format**

Offset	Field	Size (bytes)	Value
0	`SPDMVersion`	1	Shall be the `SPDMVersion` as described in SPDM version.
1	`RequestResponseCode`	1	`0x7E=VENDOR_DEFINED_RESPONSE`
2	`Param1`	1	Reserved
3	`Param2`	1	Reserved
4	`StandardID`	2	Shall indicate the registry or standard body using one of the values in the **ID** column in the Registry or standards body ID table.
6	`Len`	1	Length of the `Vendor ID` field. If the `VendorDefinedRequest` is standards-defined, length shall be `0`. If the `VendorDefinedRequest` is vendor-defined, length shall equal `Vendor ID Len`, as the Registry or standards body ID table 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	Length of the `VendorDefinedRespPayload`
7 + Len + 2	`VendorDefinedRespPayload`	ReqLength	Standard or vendor shall use this value to send the response payload.

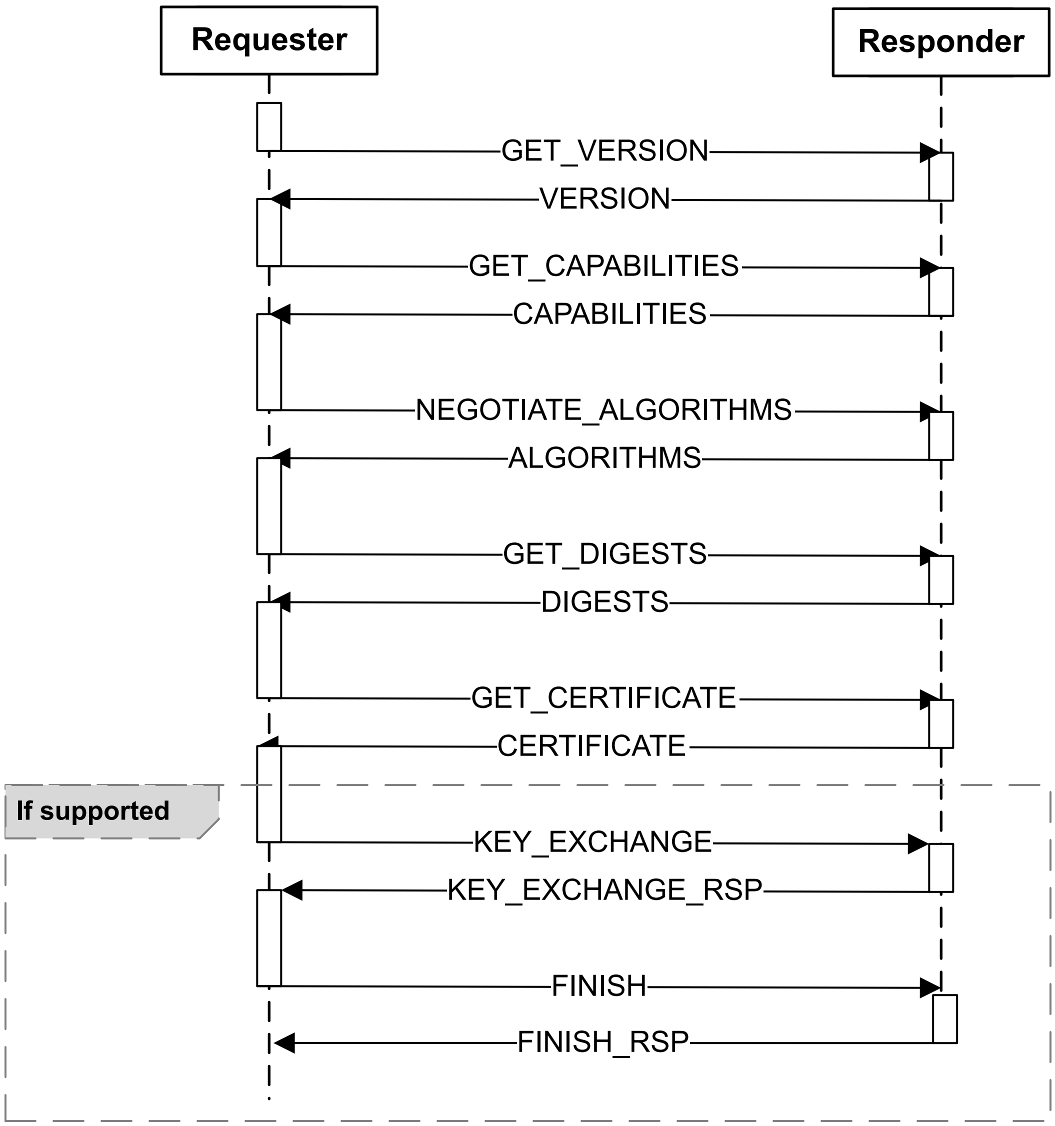
436 ## 10.16 KEY_EXCHANGE request and KEY_EXCHANGE_RSP response messages

437 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. The KEY_EXCHANGE request message format table shows the `KEY_EXCHANGE` request message format and the Successful KEY_EXCHANGE_RSP response message format table 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, and depends on the tight coupling between the two request/response message pairs.

438 The Requester and Responder pair may 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 KEY_EXCHANGE_RSP message for subsequent message communication until FINISH_RSP message completion.

439 **Responder authentication key exchange example**

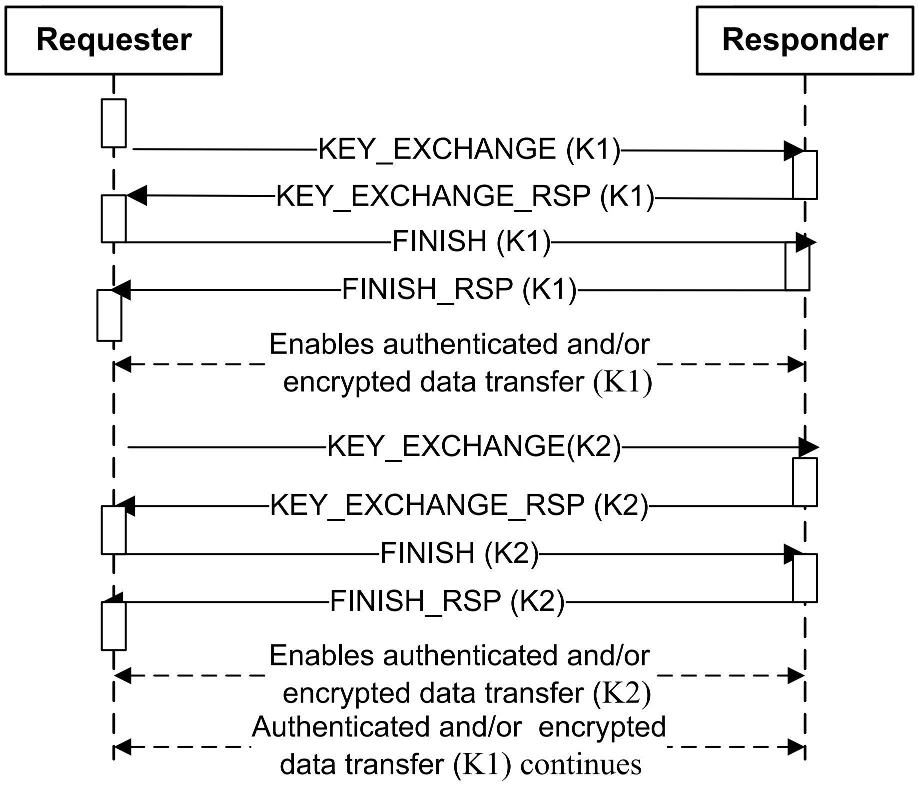
440



441 The Responder authentication multiple key exchange example provides an example of multiple sessions using two independent sets of root session keys that coexist at the same time. The specification does not require a specific temporal relationship between the second `KEY_EXCHANGE` request message and the first `FINISH_RSP` response message. To simplify implementation, however a Responder may generate an `ErrorCode=Busy` response to the second `KEY_EXCHANGE` request message until the first `FINISH_RSP` response message is complete.

442 **Responder authentication multiple key exchange example**

443



444 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.

445 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 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 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.

446 For SM2_P256 key exchange, an additional identifier, ID_A and ID_B, defined by GB/T 32918.3-2016 specification, is needed to derive the shared secret. If this algorithm is selected, the ID for the Requester (i.e. ID_A) shall be the concatenation of "Requester-KEP-dmtf-spdm-v", FullSPDMversionString and if any, the transport-specific identity. Likewise, the ID for the Responder shall be the concatenation of "Responder-KEP-dmtf-spdm-v", FullSPDMversionString and if any, the transport-specific identity. The transport should specify the transport-specific identity.

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

448 **KEY_EXCHANGE request message format**

Offset	Field	Size in bytes	Value
0	SPDMVersion	1	Shall be the SPDMVersion as described in SPDM version.
1	RequestResponseCode	1	0xE4 = KEY_EXCHANGE
2	Param1	1	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 CAPABILITIES response), the Requester shall set this value to 0x0 .
3	Param2	1	SlotID . Slot number of the Responder certificate chain that shall be used for authentication. The value in this field shall be between 0 and 7 inclusive. It shall be 0xFF if the public key of the Responder was provisioned to the Requester previously.
4	ReqSessionID	2	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	Reserved	2	Reserved

Offset	Field	Size in bytes	Value
8	`RandomData`	32	Requester-provided random data.
40	`ExchangeData`	D	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.
40 + D	`OpaqueDataLength`	2	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, OpaqueData sent by the Requester. Used to indicate any parameters that Requester wishes to pass to the Responder as part of key exchange. This field shall conform to the General Opaque Data Format.

449 **Successful KEY_EXCHANGE_RSP response message format**

Offset	Field	Size in bytes	Value
0	`SPDMVersion`	1	Shall be the `SPDMVersion` as described in SPDM version.
1	`RequestResponseCode`	1	`0x64 = KEY_EXCHANGE_RSP`
2	`Param1`	1	HeartbeatPeriod The value of this field shall be zero if Heartbeat is not supported. Otherwise, the value shall be in units of seconds.
3	`Param2`	1	Reserved.
4	`RspSessionID`	2	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).

Offset	Field	Size in bytes	Value
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. Only one of Bit 0, Bit 1 and Bit 2 shall be set. For details on the encapsulated request flow or the optimized encapsulated request flow, see the GET_ENCAPSULATED_REQUEST request and ENCAPSULATED_REQUEST response messages clause.
7	`SlotIDParam`	1	Bit[7:4] = Reserved. Bit[3:0] = `SlotID` . Slot number of the Requester certificate chain that shall be used for mutual authentication, if `MutAuthRequested` Bit 0 is set. The value in this field shall be between 0 and 7 inclusive, or `0xF` if the public key of the Requester was provisioned to the Responder through other means. 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	Responder-provided random data.
40	`ExchangeData`	D	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.

Offset	Field	Size in bytes	Value
40 + D	`MeasurementSummaryHash`	H	If the Responder does not support measurements (`MEAS_CAP=00b` in `CAPABILITIES` response) or 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(Concatenation(MeasurementBlock[0], MeasurementBlock[1], ...))` where `MeasurementBlock[x]` denotes a measurement of an element in the TCB. Measurements are concatenated in ascending order based on their measurement index. When the requested `Param2` = `0x1` and there are no measurable components in the TCB required to generate this response, this field shall be `0` . If requested `Param2` = `0xFF` , this field shall be computed as `hash(Concatenation(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 section for details. If the Responder supports both raw bit stream and digest representations for a given measurement index, then the Responder shall use the digest form. This field shall be in Hash byte order.
40 + D + H	`OpaqueDataLength`	2	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 + H	`OpaqueData`	`OpaqueDataLength`	If present, OpaqueData sent by the Responder. Used to indicate any parameters that the Responder wishes to pass to the Requester as part of key exchange. This field shall conform to the General opaque Data Format.
42 + D + H + `OpaqueDataLength`	`Signature`	`SigLen`	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 construction of the transcript hash is defined in Transcript for `KEY_EXCHANGE_RSP` signature.

Offset	Field	Size in bytes	Value
42 + D + H + `OpaqueDataLength` + `SigLen`	`ResponderVerifyData`	H	Conditional field. If the Session Handshake Phase is encrypted and/or message authenticated, then this field shall be of length H and it 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 Transcript Hash for `KEY_EXCHANGE_RSP` HMAC. The `finished_key` shall be derived from the Response Direction Handshake Secret and is described in the finished_key derivation clause. HMAC is described in RFC 2104. If both the Requester and Responder set `HANDSHAKE_IN_THE_CLEAR_CAP` to 1, then this field shall be absent.

450 **10.16.1 Session-based mutual authentication**

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

452 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 details and illustrations of these flows, see GET_ENCAPSULATED_REQUEST request and ENCAPSULATED_REQUEST response messages.

453 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 using `ErrorCode=UnexpectedRequest` and shall terminate the session.

454 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 which have obtained a Requester's certificate chains in a previous interaction.

455 **10.16.1.1 Specifying Requester certificate for Session-based mutual authentication**

456 The SPDM key exchange protocol is optimized to perform key exchange with the least number of messages exchanged. When Responder-only authentication, or mutual authentication where the Responder has obtained the certificate chains of the Requester in a previous interaction is performed, key exchange is carried out with two request/response message pairs (`KEY_EXCHANGE`, `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 authenticating a Requester whose certificates it has not obtained in a previous interaction, using a slot other than the default (slot 0), has no way of knowing in advance which `SlotID` value to use.

457 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 a `ENCAPSULATED_RESPONSE_ACK` request with `Param2` = `0x02` and an 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.

458 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`.

459 10.17 FINISH request and FINISH_RSP response messages

460 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 identify of each party to the exchanged keys and protect the entire handshake against manipulation by an active attacker. The FINISH request message format table shows the `FINISH` request message format and the Successful FINISH_RSP response message format table shows the `FINISH_RSP` response message format.

461 **FINISH request message format**

Offset	Field	Size in bytes	Value
0	`SPDMVersion`	1	Shall be the `SPDMVersion` as described in SPDM version.
1	`RequestResponseCode`	1	`0xE5 = FINISH`
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.
3	`Param2`	1	`SlotID`. Only valid if `Param1` = `0x01`, otherwise reserved. Slot number of the Requester certificate chain that shall be authenticated in Signature field. The value in this field shall be between 0 and 7 inclusive. It shall be `0xFF` if the public key of the Requester was provisioned to the Responder through other means.
4	`Signature`	`SigLen`	Signature over the transcript. `SigLen` is the size of the asymmetric signing algorithm (`BaseAsymSel` or `ExtAsymSel`) output the Responder selected via the last `ALGORITHMS` response message to the Requester. `SigLen` is zero and field not present if `Param1` = `0x00`. The construction of the transcript, signature generation and verification are defined in Transcript for `FINISH` signature, mutual authentication.

Offset	Field	Size in bytes	Value
4+ `SigLen`	`RequesterVerifyData`	H	This field 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 Transcript Hash for `FINISH` HMAC, mutual authentication. Otherwise, it shall be the Transcript Hash for `FINISH` HMAC, Responder-only authentication. The `finished_key` shall be derived from Request Direction Handshake Secret and is described in the finished_key derivation clauses. HMAC is described in RFC 2104.

462 The following clause applies when the handshake is performed in the clear (i.e. both Requester and Responder have set `HANDSHAKE_IN_THE_CLEAR_CAP` to 1): If `KEY_EXCHANGE_RSP.MutAuthRequested` equals either `0x02` or `0x04`, upon receiving `FINISH` the Responder shall confirm that the value in `FINISH.Param2` matches the value specified by the Responder in the final `ENCAPSULATED_RESPONSE_ACK.EncapsulatedRequest`.

463 **Successful FINISH_RSP response message format**

Offset	Field	Size in bytes	Value
0	`SPDMVersion`	1	Shall be the `SPDMVersion` as described in SPDM version.
1	`RequestResponseCode`	1	`0x65 = FINISH_RSP`
2	`Param1`	1	Reserved.
3	`Param2`	1	Reserved.
4	`ResponderVerifyData`	H	Conditional field. If the Session Handshake Phase is encrypted and/or message authenticated (i.e., 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 it 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 shall be the Transcript Hash for `FINISH_RSP` HMAC, mutual authentication. Otherwise, the transcript hash shall be the Transcript Hash for `FINISH_RSP` HMAC, Responder Only authentication. The `finished_key` shall be derived from Response Direction Handshake Secret and is described in the finished_key derivation clause. HMAC is described in RFC 2104.

464 10.17.1 Transcript hash calculation rules

465 The transcript hash is calculated by hashing the concatenation of the prescribed full messages or message fields in order. For messages that are encrypted, the plaintext messages are used in calculating the transcript hash.

466 The notation `[${message_name}]` . `${field_name}` is used, where:

- `${message_name}` is the name of the request or response message.
- `${field_name}` is the name of the field in the request or response message. The asterisk (`*`) means all fields in that message, except from any conditional fields that are empty (for example `KEY_EXCHANGE.OpaqueData`).

467 **Transcript for KEY_EXCHANGE_RSP signature**

```
1. `VCA`
2.  Hash of the specified certificate chain in DER format (i.e., KEY_EXCHANGE Param2) or hash of the public key in
3. [KEY_EXCHANGE].*
4. [KEY_EXCHANGE_RSP].* except the `Signature` and `ResponderVerifyData` fields.
```

468 The Responder shall generate the `KEY_EXCHANGE_RSP` signature from `SPDMsign(PrivKey, transcript, "key_exchange_rsp signing")` where `transcript` shall be the concatenation of the messages for a `KEY_EXCHANGE_RSP` signature and the `PrivKey` shall be the private 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. `SPDMsign` is described in Signature generation.

469 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 the `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.

470 **Transcript hash for KEY_EXCHANGE_RSP HMAC**

```
1. `VCA`
2. Hash of the specified certificate chain in DER format (i.e., KEY_EXCHANGE Param2) or hash of the public key in
3. [KEY_EXCHANGE].*
4. [KEY_EXCHANGE_RSP].* except the `ResponderVerifyData` field.
```

471 **Transcript for FINISH signature, mutual authentication**

```
1. `VCA`
2. Hash of the specified certificate chain in DER format (i.e., KEY_EXCHANGE Param2) or hash of the public key in i
3. [KEY_EXCHANGE].*
4. [KEY_EXCHANGE_RSP].*
5. Hash of the specified certificate chain in DER format (i.e., FINISH Param2) or hash of the public key in its pro
6. [FINISH].SPDM Header Fields
```

472 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.

473 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 `FINISH` request. `SPDMsignatureVerify` is described in Signature verification. A successful verification is when `SPDMsignatureVerify` returns success.

474 **Transcript hash for FINISH HMAC, Responder-only authentication**

```
1. `VCA`
2. Hash of the specified certificate chain in DER format (i.e., KEY_EXCHANGE's request Param2) or hash of the publi
3. [KEY_EXCHANGE].*
4. [KEY_EXCHANGE_RSP].*
5. [FINISH].SPDM Header Fields
```

475 **Transcript hash for FINISH HMAC, mutual authentication**

```
1. `VCA`
2. Hash of the specified certificate chain in DER format (i.e., KEY_EXCHANGE's request Param2) or hash of the publi
3. [KEY_EXCHANGE].*
4. [KEY_EXCHANGE_RSP].*
5. Hash of the specified certificate chain in DER format (i.e., FINISH's Param2) or hash of the public key in its p
6. [FINISH].SPDM Header Fields
7. [FINISH].Signature
```

476 **Transcript hash for FINISH_RSP HMAC, Responder-only authentication**

```
1. `VCA`
2. Hash of the specified certificate chain in DER format (i.e., KEY_EXCHANGE's request Param2) or hash of the publi
3. [KEY_EXCHANGE].*
```

```
4. [KEY_EXCHANGE_RSP].*
5. [FINISH].*
6. [FINISH_RSP].SPDM Header fields
```

477 **Transcript hash for FINISH_RSP HMAC, mutual authentication**

```
1. `VCA`
2. Hash of the specified certificate chain in DER format (i.e., KEY_EXCHANGE's request Param2) or hash of the publi
3. [KEY_EXCHANGE].*
4. [KEY_EXCHANGE_RSP].*
5. Hash of the specified certificate chain in DER format (i.e., FINISH's Param2) or hash of the public key in its p
6. [FINISH].*
7. [FINISH_RSP].SPDM Header fields
```

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

479 For additional rules, see General ordering rules.

480 10.18 PSK_EXCHANGE request and PSK_EXCHANGE_RSP response messages

481 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 the session key establishment, even if asymmetric-key cryptography is supported.

482 This option requires the Requester and the Responder to have prior knowledge of a common PSK before the handshake. Essentially, the PSK serves as a mutual authentication credential and the base of the session key establishment. As such, only the two endpoints and potentially a trusted third party that provisions the PSK to the two endpoints may 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.

483 A Requester may be paired with multiple Responders. Likewise, a Responder may be paired with multiple Requesters. A pair of Requester and Responder may be provisioned with one or more PSKs. An endpoint may 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.

484 The PSK may be provisioned in a trusted environment, for example, during the secure manufacturing process. In an untrusted environment, the PSK may be agreed upon between the two endpoints using a secure protocol. The mechanism for PSK provisioning is out of scope of this specification. The size of the provisioned PSK is determined by the requirement of security strength of the application, but should be at least 128 bits and recommended to be 256 bits or larger, to resist dictionary attacks especially when the Requester and Responder cannot both contribute sufficient entropy during the exchange. During PSK provisioning, the capabilities of an endpoint and supported algorithms may be communicated to the peer. Therefore, SPDM commands `GET_CAPABILITIES` and `NEGOTIATE_ALGORITHMS` are not required during session key establishment with the PSK option, and `Negotiated State` shall not be supported.

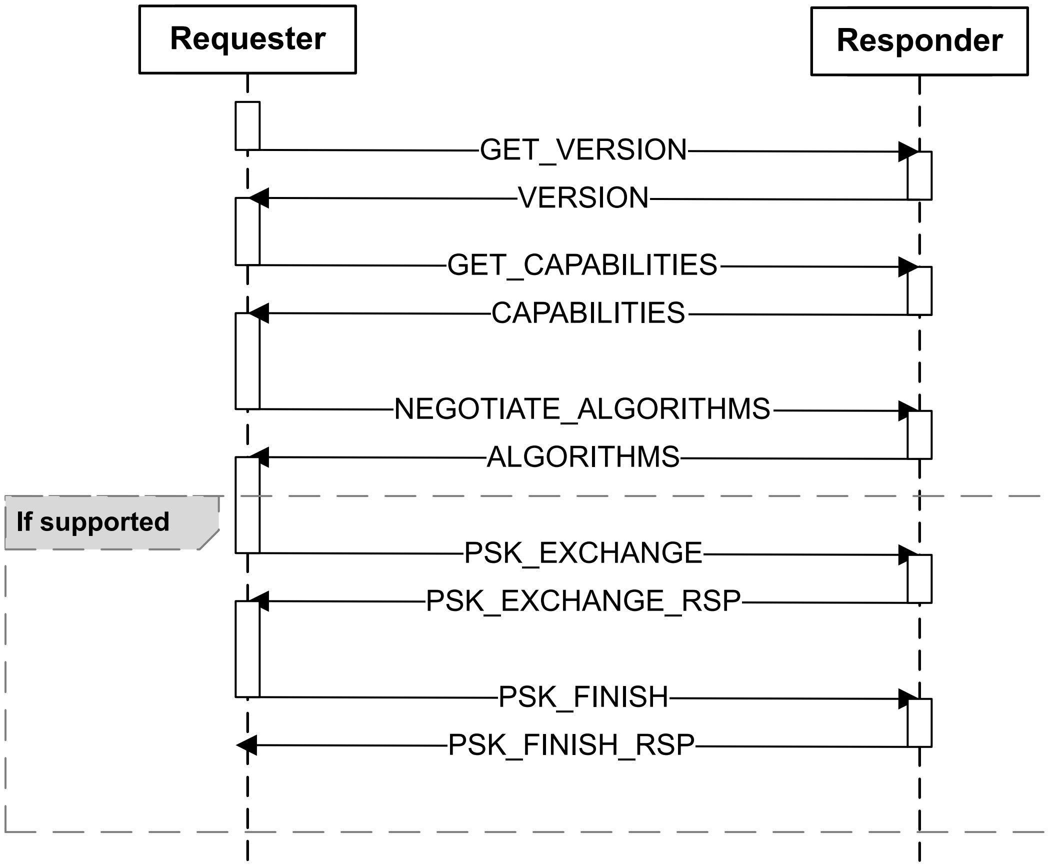
485 Two message pairs are defined for this option: `PSK_EXCHANGE` / `PSK_EXCHANGE_RSP` and `PSK_FINISH` / `PSK_FINISH_RSP`.

486 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.

487 **PSK_EXCHANGE: Example**

488



489 **PSK_EXCHANGE request message format**

Offsets	Field	Size in bytes	Value
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	`RequestResponseCode`	1	`0xE6 = PSK_EXCHANGE`

Offsets	Field	Size in bytes	Value
2	Param1	1	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 `CAPABILITIES` response), the Requester shall set this value to `0x0` .
3	Param2	1	Reserved.
4	ReqSessionID	2	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	Length of `PSKHint` in bytes.
8	R	2	Length of `RequesterContext` in bytes. R shall be equal to or greater than H, where H is the size of the underlying HMAC used in the context of the Requester.
10	`OpaqueDataLength`	2	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	Information required by the Responder to retrieve the PSK. Optional.
12 + P	RequesterContext	R	The context of the Requester. Shall include a nonce (random number or monotonic 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 Requester wishes to pass to the Responder as part of PSK-based key exchange. This field shall conform to the General opaque Data Format.

490 The field `PSKHint` is optional (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 may 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.

491 The `RequesterContext` is the contribution of the Requester to session key derivation. It shall contain a nonce

(random number or monotonic counter) to ensure that the derived session keys are ephemeral to mitigate against replay attacks. If a monotonic counter is used as the nonce, the monotonic counter shall not be reset for the lifetime of the device. The `RequesterContext` may also contain other information from the Requester.

492 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 Key Schedule.
4. Constructs `PSK_EXCHANGE_RSP` response message and sends to the Requester.

493 **PSK_EXCHANGE_RSP response message format**

Offsets	Field	Size in bytes	Value
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	`RequestResponseCode`	1	`0x66 = PSK_EXCHANGE_RSP`
2	Param1	1	HeartbeatPeriod The value of this field shall be zero if Heartbeat is not supported. Otherwise, the value shall be in units of seconds.
3	Param2	1	Reserved.
4	RspSessionID	2	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	Reserved	2	Reserved.
8	Q	2	Length of ResponderContext in bytes.
10	`OpaqueDataLength`	2	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.

Offsets	Field	Size in bytes	Value
12	`MeasurementSummaryHash`	H	If the Responder does not support measurements (`MEAS_CAP=00b` in `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(Concatenation(MeasurementBlock[0], MeasurementBlock[1], ...))` where `MeasurementBlock[x]` denotes a measurement of an element in the TCB. Measurements are concatenated in ascending order based on their measurement index. When the requested `Param1` = `0x1` and 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(Concatenation(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 section for details. If the Responder supports both raw bit stream and digest representations for a given measurement index, then the Responder shall use the digest form. This field shall be in Hash byte order.
12 + H	ResponderContext	Q	Context of the Responder. Optional. If present, shall include a nonce and/or information contributed by the Responder.
12 + H + Q	`OpaqueData`	`OpaqueDataLength`	Optional. If present, the OpaqueData sent by the Responder is used to indicate any parameters that Responder wishes to pass to the Requester as part of PSK-based key exchange. This field shall conform to the General Opaque Data Format.
12 + H + Q + `OpaqueDataLength`	ResponderVerifyData	H	Data to be verified by the Requester using the `finished_key` of the Responder.

494 The `ResponderContext` is the contribution of the Responder to session key derivation. It should contain a nonce (random number or monotonic counter) and other information of the Responder. If a monotonic counter is used as the nonce, the monotonic counter shall not be reset for the lifetime of the device. Because the Responder may be a constrained device that is not able to generate a nonce, `ResponderContext` is optional. However, the Responder is required to use `ResponderContext` if it can generate a nonce.

495 It should be noted 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 - a man-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 or session keys of the prior replayed session are not compromised.

496 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.

497 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
```

498 Note that, even if `CERTIFICATES`, `CHALLENGE_AUTH`, and/or `MEASUREMENTS` were issued, these messages would not be included in the data for calculating `ResponderVerifyData`. In other words, the identity of the signer of `CHALLENGE_AUTH` and/or `MEASUREMENTS` is not bound to identity of the sender of `PSK_EXCHANGE_RSP`. Therefore, to mitigate Responder identity impersonation, the Requester should not issue `PSK_EXCHANGE` if it has received `CHALLENGE_AUTH` and/or `MEASUREMENTS` with a signature from the Responder.

499 Upon receiving PSK_EXCHANGE_RSP, the Requester:

1. Derives the `finished_key` of the Responder by following Key Schedule.
2. Verify `ResponderVerifyData` by calculating the HMAC in the same manner as the Responder. If verification fails, the Requester aborts the session.
3. If the Responder contributes to session key derivation, such as when `PSK_CAP` in the `CAPABILITIES` of the Responder is `10b`, construct `PSK_FINISH` request and send to the Responder.

500 10.19 PSK_FINISH request and PSK_FINISH_RSP response messages

501 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`.

502 **PSK_FINISH request message format**

Offsets	Field	Size in bytes	Value
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	`RequestResponseCode`	1	`0xE7 = PSK_FINISH`
2	Param1	1	Reserved.
3	Param2	1	Reserved.
4	RequesterVerifyData	H	Data to be verified by the Responder by using the `finished_key` of the Requester.

503 To calculate `RequesterVerifyData`, the Requester calculates an HMAC. The key is the `finished_key` of the Requester, as described in Key Schedule. 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
```

504 For additional rules, see General ordering rules.

505 Upon receiving `PSK_FINISH` request, the Responder derives the `finished_key` of the Requester and calculates the HMAC independently in the same manner and verifies the result matches `RequesterVerifyData`. If verified, the Responder constructs `PSK_FINISH_RSP` response and sends to the Requester. Otherwise, the Responder sends `ERROR` response with error code `InvalidRequest` to the Requester.

506 **Successful PSK_FINISH_RSP response message format**

Offsets	Field	Size in bytes	Value
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	`RequestResponseCode`	1	`0x67 = PSK_FINISH_RSP`
2	Param1	1	Reserved.
3	Param2	1	Reserved.

507 10.20 HEARTBEAT request and HEARTBEAT_ACK response messages

508 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 `KEY_EXCHANGE_RSP` or `PSK_EXCHANGE_RSP` response messages. The Responder shall terminate the session if session traffic is not received in twice `HeartbeatPeriod`. Likewise, the Requester shall terminate the session if session traffic, including `ERROR` response, is not received in twice `HeartbeatPeriod`. Session traffic includes encrypted data at the transport layer. How SPDM is informed of encrypted data at the transport layer is outside of the scope of this specification. The Requester may retry `HEARTBEAT` requests. The Requester shall wait `ST1` time for the response before retrying.

509 The timer for the Heartbeat period shall start at the transmission, for Responders, or reception, for Requester, of either the `FINISH_RSP` or `PSK_FINISH_RSP` response messages. When determining the value of HeartbeatPeriod, the Responder should ensure this value is sufficiently greater than `RTT`.

510 For further details of session termination, see Session termination phase.

511 The HEARTBEAT request message format describes the message format.

512 **HEARTBEAT request message format**

Offsets	Field	Size in bytes	Value
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	`RequestResponseCode`	1	`0xE8 = HEARTBEAT` Request
2	Param1	1	Reserved.
3	Param2	1	Reserved.

513 The HEARTBEAT_ACK response message formatdescribes the format for the Heartbeat Response.

514 **HEARTBEAT_ACK response message format**

Offsets	Field	Size in bytes	Value
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	`RequestResponseCode`	1	`0x68 = HEARTBEAT_ACK` Response
2	Param1	1	Reserved.
3	Param2	1	Reserved.

515 ### 10.20.1 Heartbeat additional information

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

517 ## 10.21 KEY_UPDATE request and KEY_UPDATE_ACK response messages

518 To update session keys, this request shall be used. There are many reasons for doing this but an important one is when the per-record nonce will soon reach its maximum value and rollover. The KEY_UPDATE request can be issued by the Responder as well using the GET_ENCAPSULATED_REQUEST mechanism. A KEY_UPDATE request shall update session keys in the direction of the request only. 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. 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.

519 **KEY_UPDATE request message format**

Offsets	Field	Size in bytes	Value
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	`RequestResponseCode`	1	`0xE9 = KEY_UPDATE` Request
2	Param1	1	Key Operation. See KEY_UPDATE Operations Table.
3	Param2	1	Tag. This field shall contain a unique number to aid the Responder in differentiating between the original and retry request. A retry request shall contain the same tag number as the original.

520 **KEY_UPDATE_ACK response message format**

Offsets	Field	Size in bytes	Value
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	`RequestResponseCode`	1	`0x69 = KEY_UPDATE_ACK` Response
2	Param1	1	Key Operation. This field shall reflect the Key Operation field of the request.
3	Param2	1	Tag. This field shall reflect the Tag number in the `KEY_UPDATE` request.

521 **KEY_UPDATE operations**

Value	Operation	Description
0	Reserved	Reserved
1	UpdateKey	Update the single-direction key.
2	UpdateAllKeys	Update keys for both directions.
3	VerifyNewKey	Ensure the key update is successful and the old keys can be safely discarded.
4 - 255	Reserved	Reserved

522 10.21.1 Session key update synchronization

523 For clarity, in the key update process, 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.

524 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 of the scope of this specification.

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

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

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

528 In the case of `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.

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

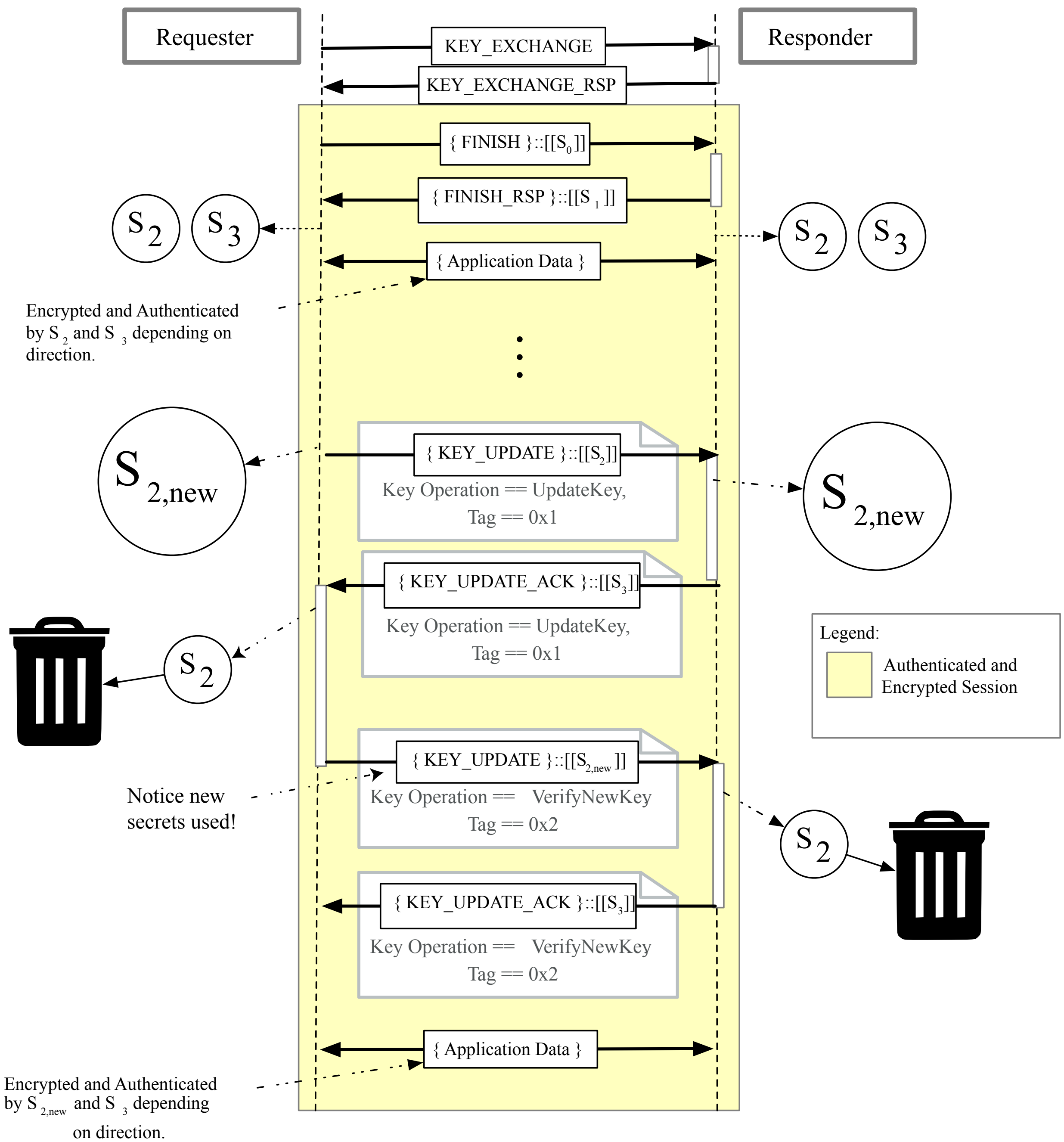
530 Upon the successful reception of the `KEY_UPDATE_ACK`, the sender shall transmit a `KEY_UPDATE` request with `VerifyNewKey` operation using the new session keys. The sender may retry until the corresponding `KEY_UPDATE_ACK` response is received. However, the sender shall be prohibited, at this point, 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. Upon successful reception of the KEY_UPDATE with VerifyNewKey operation, the receiver can now discard the old session keys. After the sender successfully receives the corresponding `KEY_UPDATE_ACK`, the transport layer may start using the new keys.

531 In certain scenarios, the receiver may need additional time to process the `KEY_UPDATE` request such as processing data already in its buffer. Thus, the receiver may reply with an `ERROR` message with `ErrorCode=Busy`. The sender should retry the request after a reasonable period of time with a reasonable amount of retries to prevent premature session termination.

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

533 The KEY_UPDATE protocol example flow figure 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.

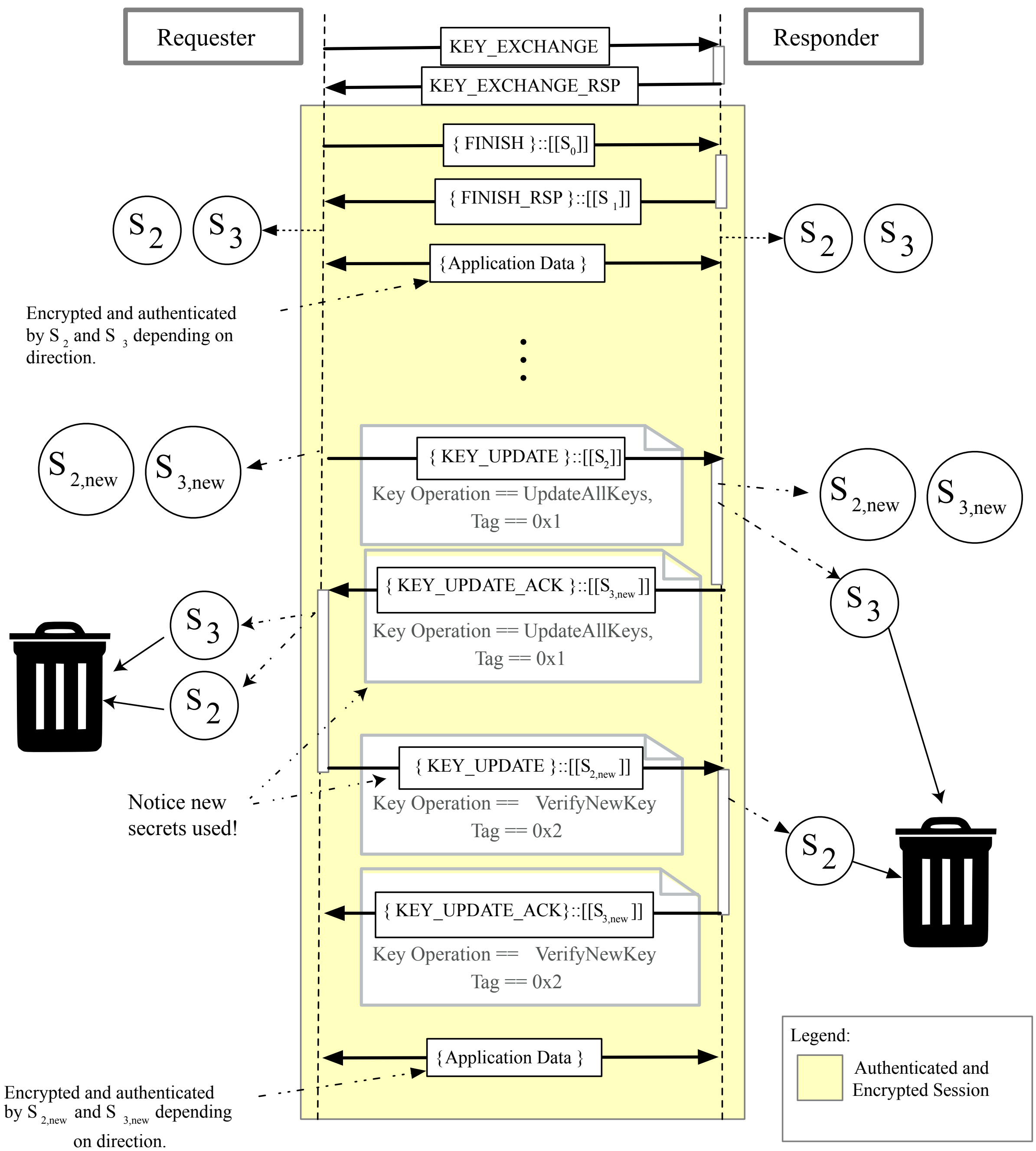
534 **KEY_UPDATE protocol example flow**

535 

536 The 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.

537 **KEY_UPDATE protocol change all keys example flow**

538



539 **10.21.2 KEY_UPDATE transport allowances**

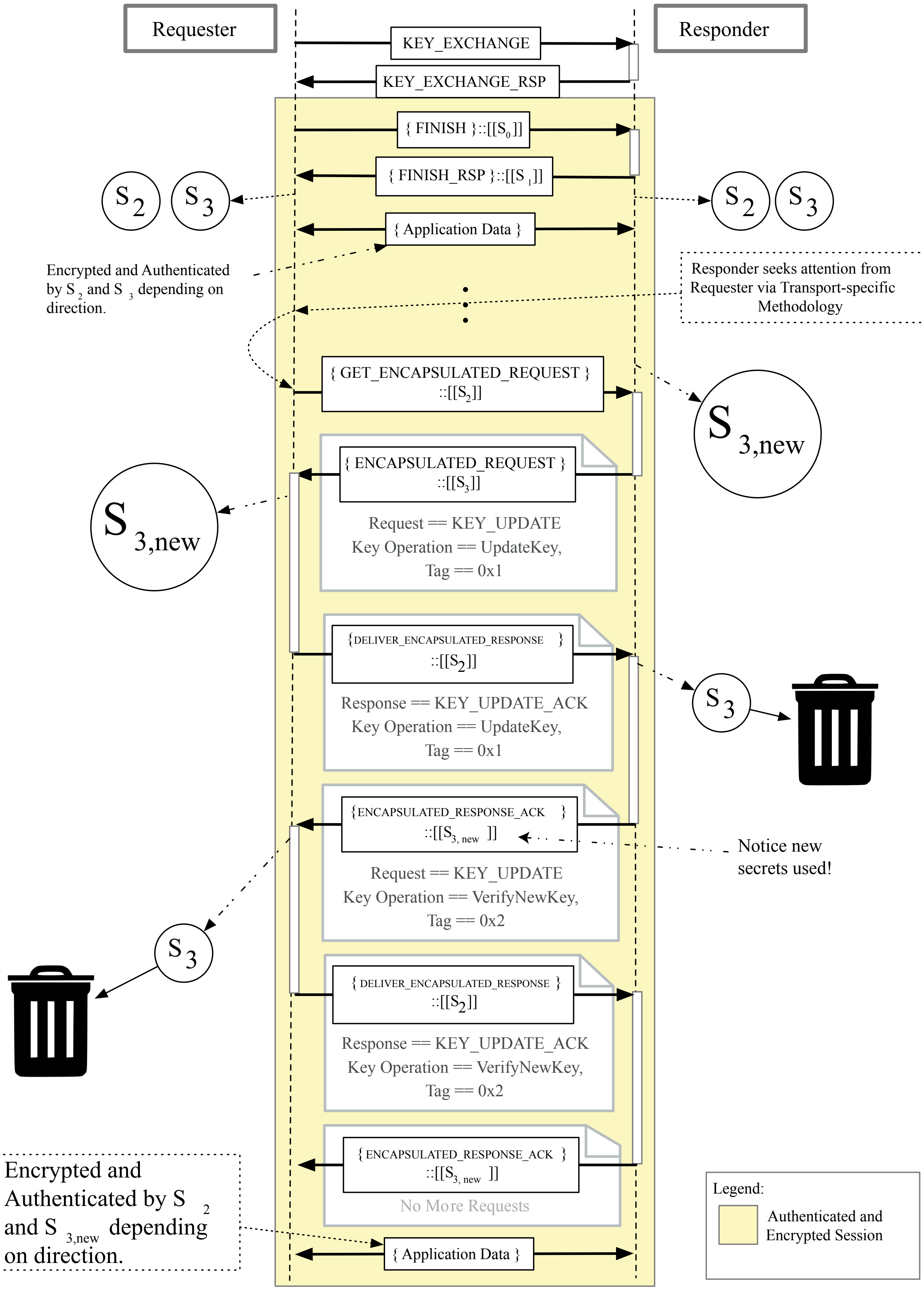
540 On some transports, bidirectional communication can occur asynchronously. On such transports, the transport may allow or disallow the `KEY_UPDATE` to be sent asynchronously without using the `GET_ENCAPSULATED_REQUEST`

mechanism. The actual method to use should be defined by the transport and is outside the scope of this specification.

541 The 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 master) is allowed to initiate all transactions on that bus. This physical transport specifies that a Responder shall alert the Requester via a sideband mechanism and to utilize the `GET_ENCAPSULATED_REQUEST` mechanism to fulfill SPDM-related requirements. Also, in this same example, the Requester and Responder are both capable of encryption and message authentication.

542 **KEY_UPDATE protocol example flow 2**

543



Requester
Responder
KEY_EXCHANGE
KEY_EXCHANGE_RSP
{ FINISH }::[[S0]]
{ FINISH_RSP }::[[S1]]
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]]
Request == KEY_UPDATE
Key Operation == UpdateKey,
Tag == 0x1
{DELIVER_ENCAPSULATED_RESPONSE } ::[[S2]]
Response == KEY_UPDATE_ACK
Key Operation == UpdateKey,
Tag == 0x1
{ENCAPSULATED_RESPONSE_ACK } ::[[S3, new]]
Notice new secrets used!
Request == KEY_UPDATE
Key Operation == VerifyNewKey,
Tag == 0x2
{ DELIVER_ENCAPSULATED_RESPONSE } ::[[S2]]
Response == KEY_UPDATE_ACK
Key Operation == VerifyNewKey,
Tag == 0x2
{ 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

544 10.22 GET_ENCAPSULATED_REQUEST request and ENCAPSULATED_REQUEST response messages

545 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. The Session-based mutual authentication figure and Optimized Session-based mutual authentication example figure are examples that illustrate the use of this scheme.

546 A Requester issues a `GET_ENCAPSULATED_REQUEST` request message to retrieve an encapsulated SPDM request message from the Responder. The response to this message (`ENCAPSULATED_REQUEST`) encapsulates the SPDM request message as if the Responder was acting as a Requester. The request message format is described in `GET_ENCAPSULATED_REQUEST` request format table. The Responder shall use the same SPDM version the Requester used.

547 10.22.1 Encapsulated request flow

548 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 Session-based mutual authentication.

549 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, with the exception of `GET_VERSION` , `RESPOND_IF_READY` and `DELIVER_ENCAPSULATED_RESPONSE` , the Requester shall not issue any other message. If a Responder receives a request other than `DELIVER_ENCAPSULATED_RESPONSE` , `RESPOND_IF_READY` or `GET_VERSION` , the Responder should respond with `ErrorCode=RequestInFlight` .

550 10.22.2 Optimized encapsulated request flow

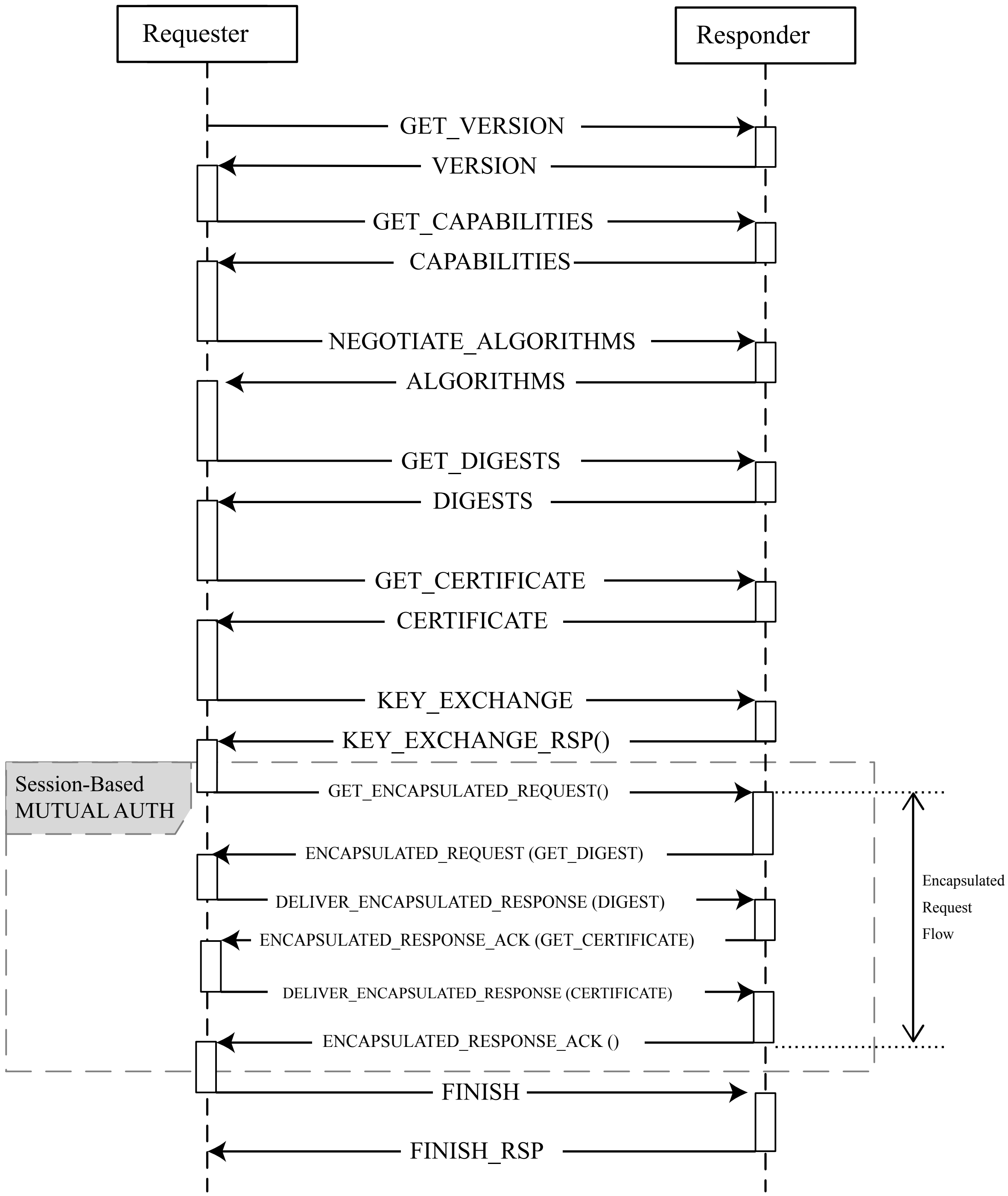
551 The optimized encapsulated request flow is similar to the encapsulated request flow but without the need of `GET_ENCAPSULATED_REQUEST` . This is because the encapsulated request accompanies one of the `Session-Secrets-Exchange` responses; thereby, removing the necessity on the Requester from issuing a `GET_ENCAPSULATED_REQUEST` . When the Responder includes an encapsulated requests with a `Session-Secrets-Exchange` response, the optimized encapsulated request flow shall start. This is also illustrated in Optimized session-based mutual authentication.

552 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`, `GET_VERSION` or Session-Secrets-Exchange, then the Responder should respond with `ErrorCode=RequestInFlight`.

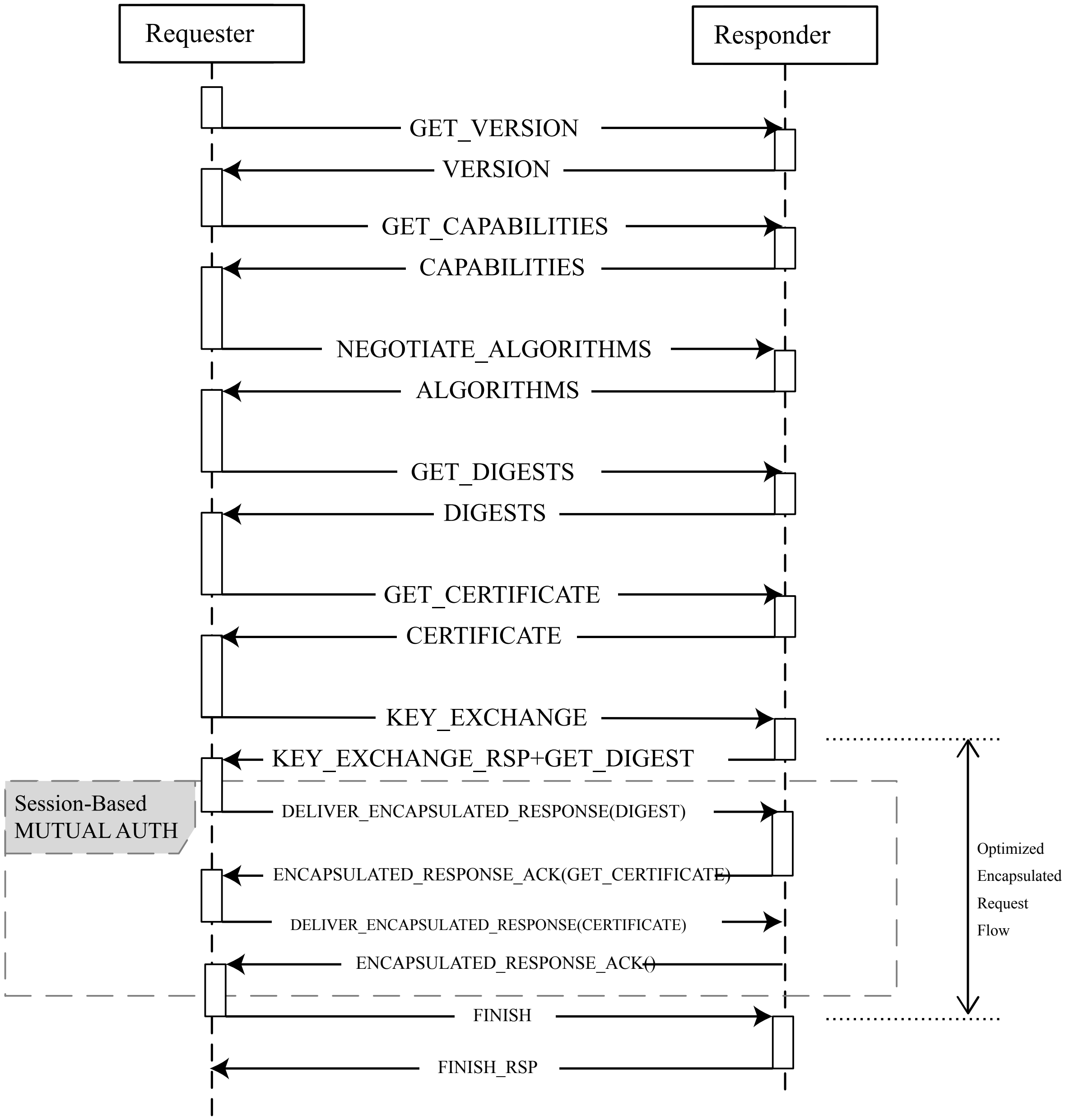
553 **Session-based mutual authentication example**

554



555 **Optimized session-based mutual authentication example**

556



557 **GET_ENCAPSULATED_REQUEST request message format**

Offsets	Field	Size in bytes	Value
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	`RequestResponseCode`	1	`0xEA = GET_ENCAPSULATED_REQUEST`
2	Param1	1	Reserved.
3	Param2	1	Reserved.

558 The ENCAPSULATED_REQUEST response message format describes the format this response.

559 **ENCAPSULATED_REQUEST response message format**

Offsets	Field	Size in bytes	Value
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	`RequestResponseCode`	1	`0x6A = ENCAPSULATED_REQUEST` Response
2	Param1	1	Request ID. This field should be unique to help the Responder match response to request.
3	Param2	1	Reserved.
4	Encapsulated Request	Variable	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 `SPDMVersion` of the `ENCAPSULATED_REQUEST` response. Both `GET_ENCAPSULATED_REQUEST` and `DELIVER_ENCAPSULATED_RESPONSE` shall be invalid requests and the Requester should respond with `ErrorCode=UnexpectedRequest` if these requests are encapsulated.

560 10.22.3 Triggering `GET_ENCAPSULATED_REQUEST`

561 Once a session has been established, the Responder may 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.

562 10.22.4 Additional constraints

563 The `GET_ENCAPSULATED_REQUEST` and `ENCAPSULATED_REQUEST` messages shall only be allowed to encapsulate

certain requests in certain scenarios. For details on these constraints, see the Session, Basic mutual authentication, and KEY_UPDATE request and KEY_UPDATE_ACK response messages clauses.

564 ## 10.23 DELIVER_ENCAPSULATED_RESPONSE request and ENCAPSULATED_RESPONSE_ACK response messages

565 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.

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

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

568 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.

569 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 for more details.

570 The request message format is described in `DELIVER_ENCAPSULATED_RESPONSE` Request Message Format Table.

571 **DELIVER_ENCAPSULATED_RESPONSE request message format**

Offsets	Field	Size (bytes)	Value
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	`RequestResponseCode`	1	`0xEB = DELIVER_ENCAPSULATED_RESPONSE` Request
2	Param1	1	Request ID. The Requester shall use the same `Request ID` (i.e., `Param1`) as provided by the Responder in the corresponding of either `ENCAPSULATED_REQUEST` or `ENCAPSULATED_RESPONSE_ACK`.
3	Param2	1	Reserved.

Offsets	Field	Size (bytes)	Value
4	Encapsulated Response	Variable	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 `SPDMVersion` of the `DELIVER_ENCAPSULATED_RESPONSE` request. Both `ENCAPSULATED_REQUEST` and `ENCAPSULATED_RESPONSE_ACK` shall be invalid responses and the Responder should respond with `ErrorCode=InvalidResponseCode` if these responses are encapsulated.

572 The ENCAPSULATED_RESPONSE_ACK response message format describes the response message format.

573 **ENCAPSULATED_RESPONSE_ACK response message format**

Offsets	Field	Size (bytes)	Value
0	`SPDMVersion`	1	Shall be the `SPDMVersion` as described in SPDM version.
1	`RequestResponseCode`	1	`0x6B = ENCAPSULATED_RESPONSE_ACK`
2	`Param1`	1	Request ID. If `EncapsulatedRequest` is present and `Param2 = 0x01`, then this field should contain a unique, non-zero number to help the Responder match response to request. Otherwise, this field shall be `0x00`.
3	`Param2`	1	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	This field 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 a retry.
5	`Reserved`	3	Reserved.

Offsets	Field	Size (bytes)	Value
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 SPDMVersion of the ENCAPSULATED_REQUEST response. Both GET_ENCAPSULATED_REQUEST and DELIVER_ENCAPSULATED_RESPONSE shall be invalid requests and the Requester shall respond with 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.

574 ### 10.23.1 Additional information

575 Using a unique request ID is highly recommended to aid the Responder in avoiding confusion between a retry and a new DELIVER_ENCAPSULATED_RESPONSE message. For example, if the Responder sent the ENCAPSULATED_RESPONSE_ACK with a new encapsulated request and that 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 when, in fact, it was a retry. This mistake may cause additional mistakes to occur.

576 In general, the response timing for ENCAPSULATED_RESP_ACK shall be subject to the same timing constraints as the encapsulated request. For example, if the encapsulated request was CHALLENGE_AUTH , the Responder, too, shall adhere to CT timing rules when it has a subsequent request. The Requester may return ErrorCode=ResponseNotReady .

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

578 ### 10.23.2 Allowance for encapsulated requests

579 Only certain requests can be encapsulated in any encapsulated request flow. Their corresponding response, including ERROR , can be encapsulated too. Additionally, these requests are only allowed in certain flows, such as Basic Mutual Authentication, and are described in various parts of this specification. The consolidated list of requests allowed to be encapsulated shall be these requests:

- CHALLENGE
- GET_CERTIFICATE
- GET_DIGEST
- KEY_UPDATE

- `SUBSCRIBE_EVENT_GROUP`
- `SEND_EVENT`
- `GET_SUPPORTED_EVENT_GROUPS`

580 If a request is not in the list, then the request and its corresponding response shall be prohibited from being encapsulated.

581 10.23.3 Certain error handling in encapsulated flows

582 These clauses describe special error scenarios and their handling requirements.

583 10.23.3.1 Response not ready

584 In an encapsulated request flow, a Responder may issue an encapsulated request that can take up to `CT` time to fulfill. When the Requester delivers an `ERROR` message with a `ResponseNotReady` error code, 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.

585 The Responder should wait the amount of time indicated in the `ERROR` message for this particular error code.

586 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, the Requester can drop the original response.

587 10.23.3.2 Timeouts

588 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.

589 Both Responder and Request should comply with the timing requirements laid forth in Timing requirements.

590 10.24 END_SESSION request and END_SESSION_ACK response messages

591 This request shall terminate a session. Further communication between the Requester and Responder using the same session ID shall be prohibited. See Session termination phase clause for details.

592 The END_SESSION request message format table describes this format.

593 **END_SESSION request message format**

Offset	Value	Field	Description
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	`RequestResponseCode`	1	`0xEC = END_SESSION`
2	Param1	1	See the End session request attributes table.
3	Param2	1	Reserved.

594 **End session request attributes**

Offset	Value	Field	Description
0	0	Negotiated State Preservation Indicator	If the Responder supports Negotiated State caching (`CACHE_CAP=1`), the Responder shall preserve the Negotiated State. Otherwise, this field shall be ignored.
0	1	Negotiated State Preservation Indicator	If the Responder supports Negotiated State caching (`CACHE_CAP=1`), the Responder shall also clear the Negotiated State as part of session termination. If there is no Negotiated State to be cleared due to a previous `END_SESSION` request message with this field set to 1, this field shall be ignored. If the responder does not support Negotiated State caching (`CACHE_CAP=1`), this field shall be ignored.
[7:1]	Reserved	Reserved	Reserved.

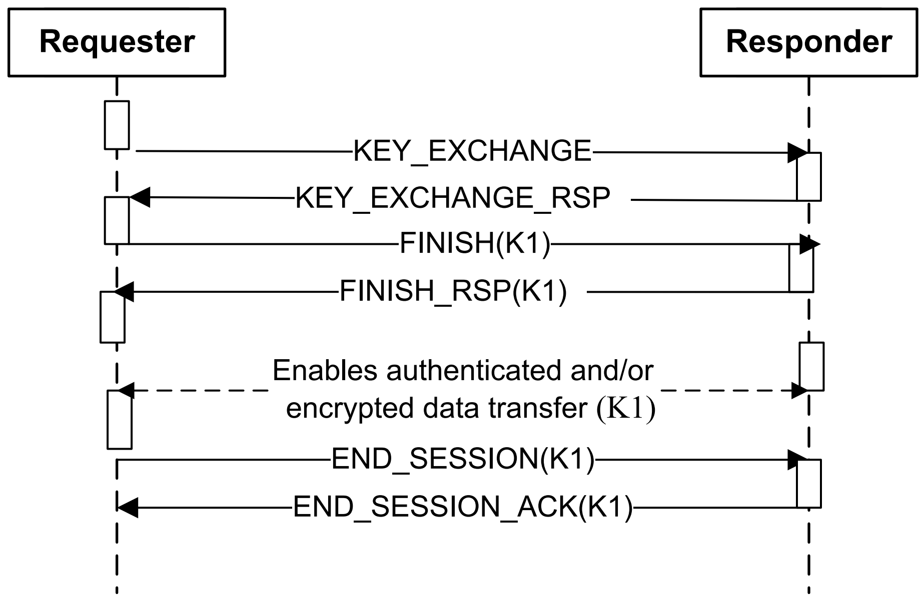
595 The END_SESSION_ACK response message format describes the response message.

596 **END_SESSION_ACK response message format**

Offset	Value	Field	Description
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	`RequestResponseCode`	1	`0x6C = END_SESSION_ACK`
2	Param1	1	Reserved.
3	Param2	1	Reserved.

597 **END_SESSION protocol flow**

598



599 10.25 Certificate Provisioning

600 These clauses describe the request and response messages used for provisioning a device with certificate chains. Provisioning of Slot 0 should be only done in a secure manufacturing environment.

601 10.25.1 GET_CSR request and CSR response messages

602 The `GET_CSR` request message shall retrieve a Certificate Signing Request (CSR) from the Responder. For the provisioning of Slot 0, this command should be run in a secure manufacturing environment. For all additional slots, the Requester shall issue this command inside a secure session. Verification of request authorization for slots 1-7 is outside the scope of the current revision of the specification.

603 A Responder shall only process a `GET_CSR` request if it already possesses an appropriate asymmetric key pair for each of the signature suites (algorithms and associated parameters) it supports. If more than one signature suites are supported, selection of the appropriate signature suite (and thus 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 others contributed by the Responder itself. `RequesterInfo` format shall comply to the PKCS #10 specification in RFC2986, specifically the `CertificationRequestInfo` format. OEM extensions (i.e. OEM OIDs) shall be encoded using the `Attributes` type. The Responder shall return an `ERROR` message with error code `InvalidRequest` if it cannot support all of the fields included in the `RequesterInfo`.

604 The attributes of the resulting CSR and their values shall comply with the clauses presented in the Leaf certificate section.

605 The GET_CSR request message format table shows the `GET_CSR` request message format.

606 The CSR response message format table shows the `CSR` response message format.

607 The resulting CSR contained in a successful `CSR` response will have to 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.

608 **GET_CSR request message format**

Offset	Field	Size (bytes)	Value
0	`SPDMVersion`	1	Shall be the `SPDMVersion` as described in SPDM version.
1	`RequestResponseCode`	1	`0xED=GET_CSR`
2	`Param1`	1	Reserved
3	`Param2`	1	Reserved
4	`RequesterInfoLength`	2	Length of `RequesterInfo` field in bytes provided by the Requester. This field can be 0.
6	`OpaqueDataLength`	2	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`	Optional information provided by the Requester.
8 + `RequesterInfo`	`OpaqueData`	`OpaqueDataLength`	The Requester may include vendor-specific information for the Responder to generate the CSR. This field is optional.

609 **CSR response message format**

Offset	Field	Size (bytes)	Value
0	`SPDMVersion`	1	Shall be the `SPDMVersion` as described in SPDM version.
1	`RequestResponseCode`	1	`0x6D=CSR`
2	`Param1`	1	Reserved
3	`Param2`	1	Reserved
4	`CSRLength`	2	Length of the `CSRdata` in bytes.

Offset	Field	Size (bytes)	Value
6	`Reserved`	2	Reserved
8	`CSRdata`	`CSRLength`	Requested contents of the CSR. DER-encoded.

610 The `CSRdata` format shall comply to the PKCS #10 specification in RFC2986.

611 10.25.2 SET_CERTIFICATE request and SET_CERTIFICATE_RSP response messages

612 For Slot 0 provisioning, the Requester should issue `SET_CERTIFICATE` only in a secure manufacturing environment. The Requester shall issue `SET_CERTIFICATE` inside a secure session for slot 1-7 provisioning. Responder verification of Requester authorization to issue this request is outside the scope of the current revision of the specification. The device may 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 `ErrorCode=ResetRequired` response.

613 The SET_CERTIFICATE request message format table shows the `SET_CERTIFICATE` request message format.

614 The SET_CERTIFICATE_RSP response message format table shows the `SET_CERTIFICATE_RSP` response message format.

615 **SET_CERTIFICATE request message format**

Offset	Field	Size (bytes)	Value
0	`SPDMVersion`	1	Shall be the `SPDMVersion` as described in SPDM version.
1	`RequestResponseCode`	1	`0xEE=SET_CERTIFICATE`
2	`Param1`	1	Bit [7:4] = Reserved. Bit[3:0] = `SlotID` where the new certificate is written The value in this field shall be between 0 and 7 inclusive.
3	`Param2`	1	Reserved
4	`CertChain`	Variable	Contents of target certificate chain, as specified in Certificates and certificate chains.

616 **Successful SET_CERTIFICATE_RSP response message format**

Offset	Field	Size (bytes)	Value
0	`SPDMVersion`	1	Shall be the `SPDMVersion` as described in SPDM version.
1	`RequestResponseCode`	1	`0x6E=SET_CERTIFICATE_RSP`
2	`Param1`	1	Bit [7:4] = Reserved. Bit[3:0] = `SlotID` where the new certificate is written The value in this field shall be between 0 and 7 inclusive.
3	`Param2`	1	Reserved.

617 ## 10.26 Event mechanism

618 An SPDM endpoint may want to be notified of changes from another SPDM endpoint. These changes are called events. The SPDM event mechanism provides a framework for the asynchronous notification of events over a secure session. An SPDM endpoint sending an event is called an Event Notifier and an SPDM endpoint receiving an event is called an Event Recipient. 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.

619 An event is identified by its event group, event type and an event instance ID. An event group is a group of all events from a given standards body or vendor. The event instance ID is a unique numeric value that represents that occurrence of an event.

620 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 subscription is managed by the Event Notifier. An Event Notifier shall only send events that are in the event subscription.

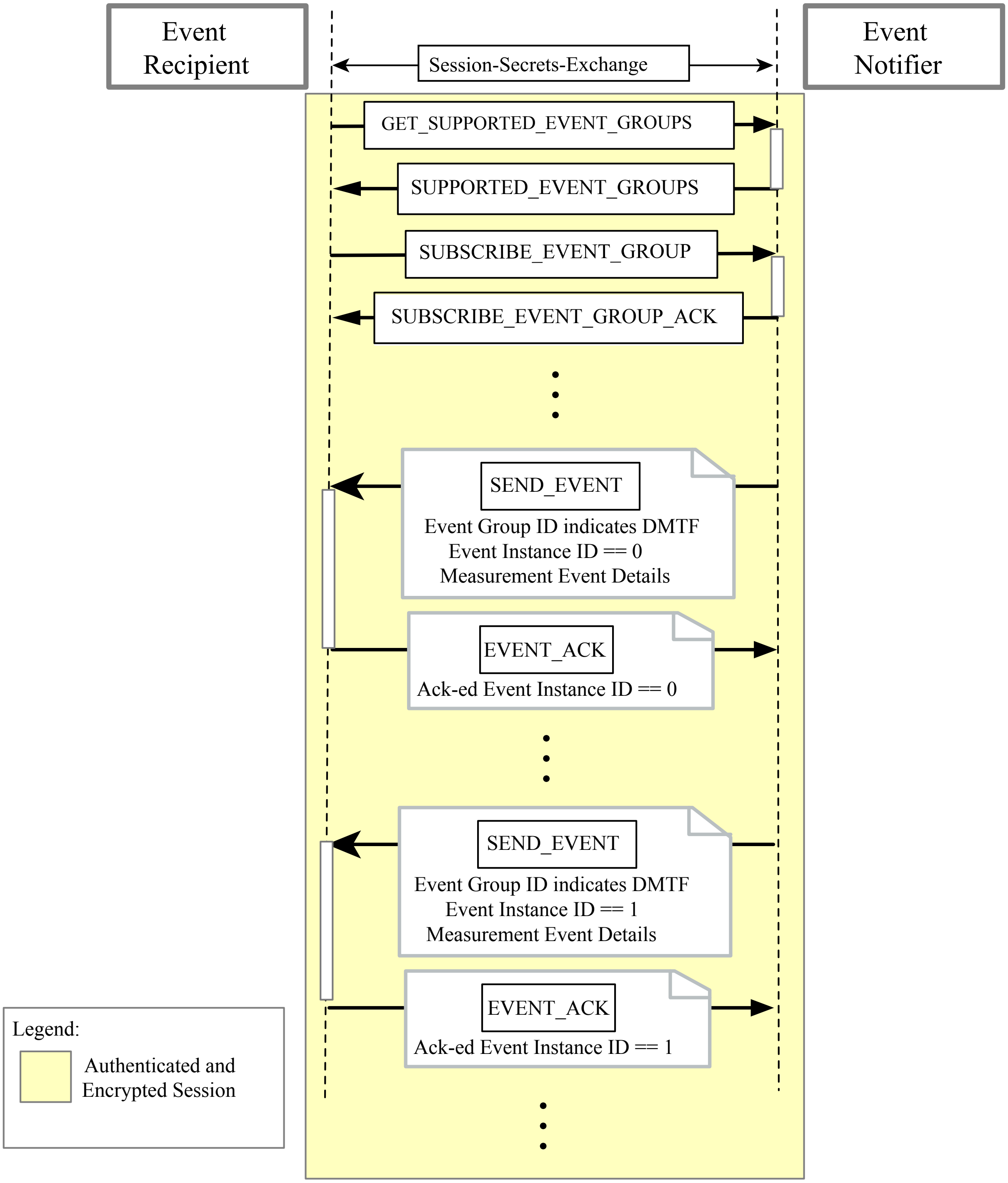
621 At the start of a secure session, an Event Notifier shall not send any events in that session until an Event Recipient subscribes to one or more event groups.

622 Lastly, the Event Notifier shall start with an event instance ID of zero for that secure session.

623 The Event Flow diagram illustrates a typical event flow for event subscription and event delivery over a transport capable of asynchronous bi-directional communication.

624 **Event Flow Diagram**

625



626 When `EVENT_CAP` is set, an Event Notifier shall support `SUBSCRIBE_EVENT_GROUP`, `GET_SUPPORTED_EVENT_GROUPS`, `SEND_EVENT` and their corresponding response messages.

627 **10.26.1 SUBSCRIBE_EVENT_GROUP request and SUBSCRIBE_EVENT_GROUP_ACK response message**

628 This request and response messages allow an Event Recipient to communicate the list of SPDM event groups it is interested in receiving. In addition, the same request and response message can be used to communicate SPDM event groups an Event Recipient is no longer interested in receiving. This request subscribes or unsubscribes all events for a given event group.

629 The event group the Event Recipient is interested in receiving shall be added to the event subscription. Event groups the Event Recipient is no longer interested in receiving shall be removed from the event subscription.

630 An Event Notifier shall be able to begin sending events once the Event Recipient registers at least one event group into the event subscription.

631 To subscribe or unsubscribe to an event group, an Event Recipient shall send the `SUBSCRIBE_EVENT_GROUP` request message. An Event Notifier shall add or remove event types from the event subscription based on the content of this request message. The SUBSCRIBE_EVENT_GROUP request message format describes the message format.

632 The variables `F0` and `F1` are scoped locally within the following table.

633 **SUBSCRIBE_EVENT_GROUP request message format**

Offset	Field	Size (bytes)	Value
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	`RequestResponseCode`	1	`0xEF = SUBSCRIBE_EVENT_GROUP`
2	Param1	1	SubscribeLen. Shall be the number of SVH elements in `SubscribeList`. A value of zero shall indicate the Event Recipient no longer wants to receive any events. This is the equivalent of an empty event subscription or the removal of all event groups in an event subscription.
3	Param2	1	Reserved.
4	SubscribeList	F0	Shall be a list of SVH. If a standard body or vendor is in this list, all events from that standard body or vendor shall be added to the event subscription for that Event Recipient. The size, indicated by F0, of this field shall be the size of this list. This field shall contain the complete list of all event groups the Event Recipient wants to subscribe to. This list shall replace the current event subscription.

634 The SUBSCRIBE_EVENT_GROUP_ACK response message format describes the response format for the `SUBSCRIBE_EVENT_GROUP` request.

635 **SUBSCRIBE_EVENT_GROUP_ACK request message format**

Offsets	Field	Size in bytes	Value
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	`RequestResponseCode`	1	`0x6F = SUBSCRIBE_EVENT_GROUP_ACK` Response
2	Param1	1	`SubscriptionTotal`. This field shall indicate the total number of subscribed event groups for the Event Recipient. This field shall be the same as `SubscribeLen` in `SUBSCRIBE_EVENT_GROUP`.
3	Param2	1	Reserved.

636 For event types defined by this specification, see DMTF event types.

637 10.26.2 GET_SUPPORTED_EVENT_GROUPS request and SUPPORTED_EVENT_GROUPS response message

638 This request and response message is used to retrieve the list of all event groups supported by the Event Notifier.

639 The GET_SUPPORTED_EVENT_GROUPS request message format describes the message format.

640 **GET_SUPPORTED_EVENT_GROUPS request message format**

Offset	Field	Size (bytes)	Value
0	`SPDMVersion`	1	Shall be the `SPDMVersion` as described in SPDM version.
1	`RequestResponseCode`	1	`0xE2 = GET_SUPPORTED_EVENT_GROUPS`
2	Param1	1	Reserved.
3	Param2	1	Reserved.

641 The SUPPORTED_EVENT_GROUPS response message format describes the message format for this response.

642 **SUPPORTED_EVENT_GROUPS response message format**

Offsets	Field	Size in bytes	Value
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.

Offsets	Field	Size in bytes	Value
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.
4	SupportedEventGroupsList	Variable	Shall be a list of all event groups supported by the Event Notifier. This list shall include the DMTF event group. The format of this field shall be a list of SVH to identify the event group. The size of this field shall be the size of this list.

643 **10.26.3 SEND_EVENT request and EVENT_ACK response message**

644 To deliver subscribed events to an Event Recipient, the Event Notifier shall use this request message. More than one event can accompany this request. The maximum size of a request shall be less than or equal to the `DataTransferSize` of the Event Recipient.

645 The SEND_EVENT request message format table describes this request.

646 **SEND_EVENT request message format**

Offset	Field	Size (bytes)	Value
0	`SPDMVersion`	1	Shall be the `SPDMVersion` as described in SPDM version.
1	`RequestResponseCode`	1	`0xF0 = 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.

647 The Event Data table describes the format for details of each event.

648 The variables `F0`, `F1`, and `F2` are scoped locally within the following table.

649 **Event data table**

Offset	Field	Size (bytes)	Value
0	EventInstanceId	4	Shall be the event instance id for the event.
4	EventGroupId	F0	Shall indicate the event group the event type belongs to. The format of this field shall be the SVH format.
4 + F0	Padding	F1	If the size of `EventGroupId` is not a multiple of 4, this field shall be present and have a length of 1, 2 or 3 to ensure the size of `EventGroupId` + `Padding` be a multiple of 4.
4 + F0 + F1	EventType	2	Shall be the event type for the event.
6 + F0 + F1	EventDetailLen	2	Shall be the length of `EventDetail`.
8 + F0 + F1	EventDetail	F2	Shall be the event specific details. This field is specific to the event type in the event group. For the DMTF event group, see Event type details clauses for further information. The size, indicated by F2, shall be the size of this event specific details.
8 + F0 + F1 + F2	EventPadding	0, 1, 2 or 3	Shall be zero-filled. This field shall be a length of zero, one, two or three bytes to ensure the total size of event data is a multiple of four.

650 The EVENT_ACK response message format table describes the format for the response.

651 **EVENT_ACK response message format**

Offsets	Field	Size in bytes	Value
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	`RequestResponseCode`	1	`0x70 = EVENT_ACK` Response
2	Param1	1	Reserved.
3	Param2	1	Reserved.
4	AckEventInstanceId	4	Shall be the highest `EventInstanceId` in the corresponding request.

652 If the Event Recipient does not acknowledge the request, the Event Notifier shall resend the unacknowledged event instance IDs as a new `SEND_EVENT` request at least once. The Event Notifier should resend the unacknowledged event instance IDs at least three times. The Event Notifier should only send unacknowledged event instance IDs. The interval between resending shall be at least 100 ms + `RTT`. The new request may also include new event instance IDs. The Event Notifier can retire the event if it remains unacknowledged. If the event is retired because it is not acknowledged, the Event Notifier shall send an event lost event.

653 The size of SEND_EVENT 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. If the size of a `SEND_EVENT` Request with only one event is greater than the `MaxSPDMmsgSize` of the Event Recipient, an Event Notifier shall, instead, send a `SEND_EVENT` Request with only an Event Lost event (i.e., `EventType` == EventLost) as an indication that the original event was too big in size. To ensure an Event Recipient can receive an Event Lost event, the Event Recipient shall have a `MaxSPDMmsgSize` greater than or equal to 28 bytes. If the `MaxSPDMmsgSize` of the Event Recipient does not meet the minimum size requirement, an Event Notifier shall prohibit an Event Recipient from successfully subscribing to any event groups.

654 10.26.4 Event Instance ID

655 Event Instance ID typically reflects the order of changes in the Event Notifier from a chronologically perspective. The event instance ID shall start at zero and monotonically increase for every new event. This method also allows the Event Recipient to determine if an event was lost.

656 10.27 Large SPDM message transfer mechanism

657 A Large SPDM message is an SPDM message whose size is greater than the `DataTransferSize` of the receiving SPDM endpoint. These clauses provide a transport agnostic mechanism to transfer Large SPDM messages. This mechanism will be used only when the size of an SPDM message exceeds the `DataTransferSize` of the receiving 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 the `DataTransferSize` of the receiving SPDM endpoint, the sending SPDM endpoint shall not utilize this transfer mechanism.

658 This transfer mechanism divides a Large SPDM message into smaller fragments. With the exception of the first and last fragment, all fragments are equal in size. These fragments are called chunks. The chunks shall be numbered and shall transfer in sequence. The chunks and transfer sequence are as such:

- 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 until the last chunk. These numeric values are 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 until 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 in a monotonically increasing chunk sequence number until the last chunk.
- `CHUNK_SEND` , `CHUNK_GET` and their corresponding Responses shall be used to transfer these chunks.

659 The `ChunkSeqNo` fields indicate the chunk sequence number for a given chunk.

660 The Requests and Responses, defined in these clauses, handle the transfer of each chunk.

661 10.27.1 CHUNK_SEND request and CHUNK_SEND_ACK response message

662 `CHUNK_SEND` request and `CHUNK_SEND_ACK` response shall be used to send a request to an SPDM endpoint when the size of the request is greater than the `DataTransferSize` of the receiving SPDM endpoint.

663 The CHUNK_SEND request format table describes the format for the request.

664 **CHUNK_SEND request format table**

Offsets	Field	Size in bytes	Value
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	`RequestResponseCode`	1	`0x85 = CHUNK_SEND` Request
2	Param1	1	Request Attributes. See Chunk Sender Attributes.
3	Param2	1	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 entirely monotonically increase or entirely monotonically 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	2	This field shall identify the chunk number associated with `SPDMChunk`.
6	Reserved	2	Reserved
8	ChunkSize	4	This field shall indicate the size of `SPDMchunk`. See Additional chunk transfer requirements for details.
12	LargeMessageSize	L0 = 0 or 4	This field 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 should be greater than the `DataTransferSize` of the receiving SPDM endpoint.
12 + `L0`	SPDMchunk	Variable	This field shall contain the chunk of the Large SPDM Request associated with `ChunkSeqNo`.

665 **Chunk Sender Attributes**

Bit	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.

666 The CHUNK_SEND_ACK response format table describes the format for the response.

667 **CHUNK_SEND_ACK response format table**

Offsets	Field	Size in bytes	Value
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	`RequestResponseCode`	1	`0x5 = CHUNK_SEND_ACK` Request
2	Param1	1	Response Attributes. See Chunk Receiver Attributes.
3	Param2	1	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 SPDM message.
4	ChunkSeqNo	2	This field shall be the same as `ChunkSeqNo` in the corresponding request.
5+	ResponseToLargeRequest	Variable	This field shall be present on the last chunk (i.e. `LastChunk` is set) or when the `EarlyErrorDetected` bit in `Param1` is set. This field shall contain the Response to the Large SPDM Request. When the `EarlyErrorDetected` bit in `Param1` is set, this field shall contain an `ERROR` message.

668 **Chunk Receiver Attributes**

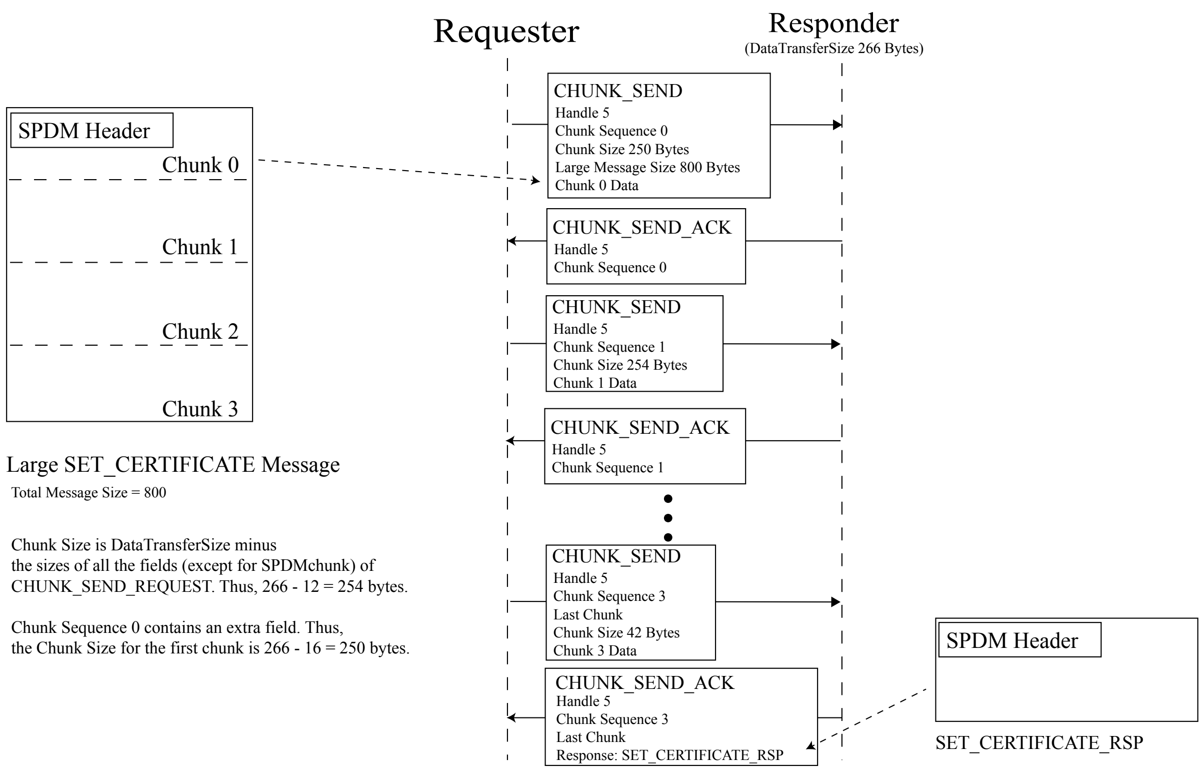
Bit	Field	Description
0	EarlyErrorDetected	If set, the receiver of a Large SPDM Request detected an error in the Request before the last chunk was received. If set, the sender of the Large SPDM Request shall terminate the transfer of any remaining chunks. After addressing the issue, the sender of the failed Large SPDM Request can transfer the fixed Large SPDM Request as a new transfer.
[7:1]	Reserved	Reserved.

669 The CHUNK_SEND_ACK response format table describes the format for the response.

670 Upon reception of the last chunk, the receiving SPDM endpoint shall respond with the response corresponding to the Large SPDM Request in `ResponseToLargeRequest`. If placing the response in `ResponseToLargeRequest` causes the size of the `CHUNK_SEND_ACK` to exceed `DataTransferSize`, the receiving end point shall, instead, respond to `CHUNK_SEND` with an `ERROR` message using `ErrorCode == LargeResponse`. An `ERROR` message with an `ErrorCode == LargeResponse` shall not be allowed in `ResponseToLargeRequest`. An `ERROR` messages with other `ErrorCodes` 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.

671 The Large SET_CERTIFICATE example illustrates the sending of a Large SPDM Request to a Responder.

672 **Large SET_CERTIFICATE example**



673 **10.27.2 CHUNK_GET request and CHUNK_RESPONSE response message**

674 `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.

675 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 using `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 the `ERROR` message with `ErrorCode == LargeResponse`.

676 The CHUNK_GET request format table describes the format for the request.

677 **CHUNK_GET request format table**

Offsets	Field	Size in bytes	Value
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	`RequestResponseCode`	1	`0x86 = CHUNK_GET` Request
2	Param1	1	Reserved.
3	Param2	1	Handle. This field shall be the same value as given in the `Handle` field of the `ERROR` message with `ErrorCode = LargeResponse` .
4	ChunkSeqNo	2	This field shall indicate the desired chunk sequence number of the Large SPDM Response to retrieve.

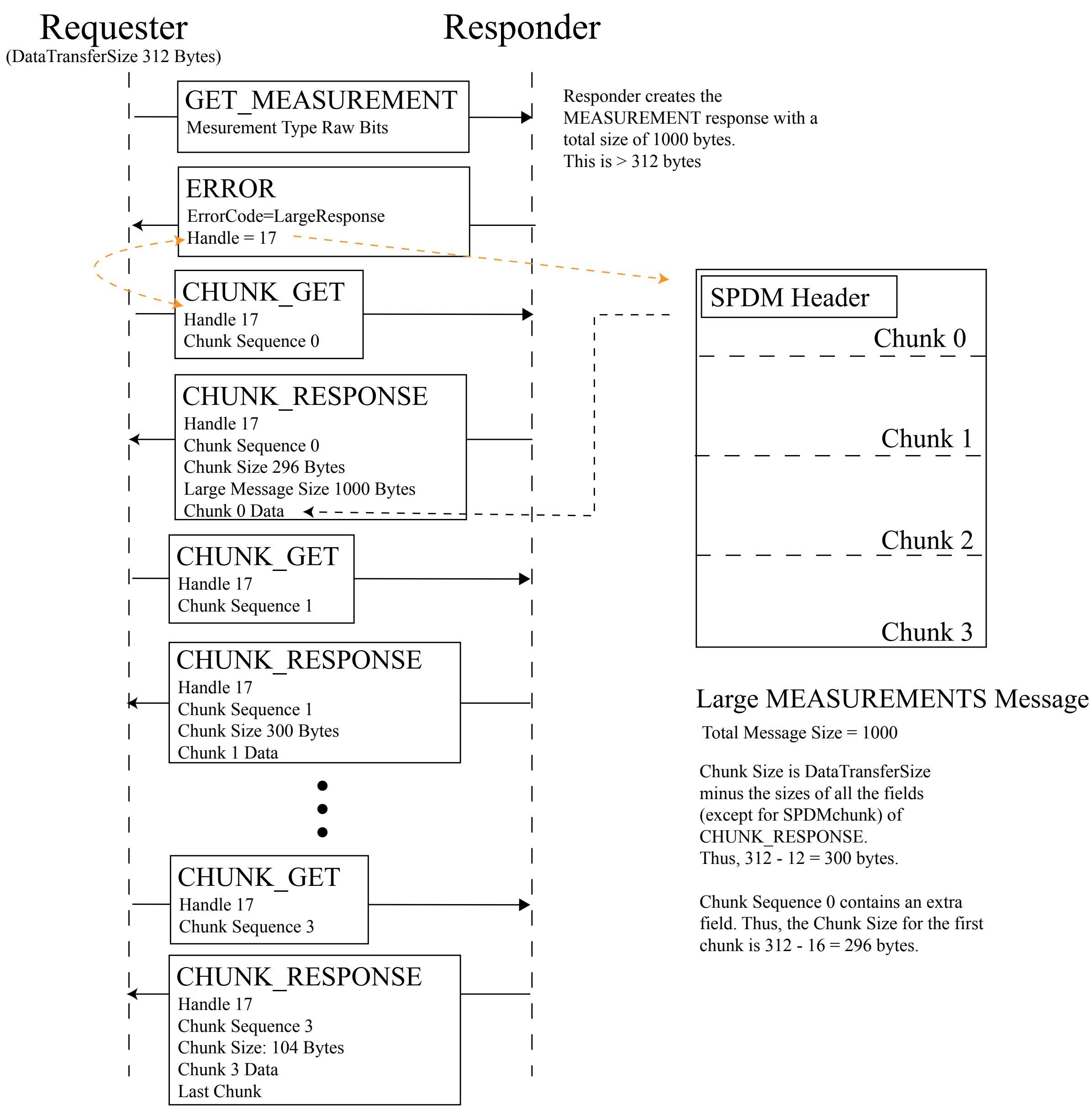
678 The CHUNK_RESPONSE response format table describes the format for the response.

679 **CHUNK_RESPONSE response format table**

Offsets	Field	Size in bytes	Value
0	SPDMVersion	1	Shall be the `SPDMVersion` as described in SPDM version.
1	`RequestResponseCode`	1	`0x85 = CHUNK_RESPONSE` Response
2	Param1	1	Response Attributes. See Chunk Sender Attributes.
3	Param2	1	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	2	This field 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` .
6	Reserved	2	Reserved
8	ChunkSize	4	This field shall indicate the size of `SPDMchunk` . See Additional chunk transfer requirements for details.
12	LargeMessageSize	L0 = 0 or 4	This field 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 should be greater than the `DataTransferSize` of the receiving SPDM endpoint.
12 + `L0`	SPDMchunk	Variable	This field shall contain the chunk of the Large SPDM Request associated with `ChunkSeqNo` .

680 The Large MEASUREMENT example illustrates the sending and retrieval of a Large SPDM Response to a Requester that issued a `GET_MEASUREMENT` request.

681 **Large MEASUREMENT example**



682 10.27.3 Additional chunk transfer requirements

683 When transferring a Large SPDM message, an SPDM endpoint shall be prohibited from transferring a chunk sequence number (i.e. `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 may 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 silent discard the request or respond with an `ERROR` message with `ErrorCode = InvalidRequest`.

684 In general, 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 equal 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.

685 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 reliability of this transfer mechanism and general interoperability, these messages shall be prohibited from being transferred in chunks using this transfer mechanism:

- `GET_VERSION`
- `GET_CAPABILITIES`
- `CAPABILITIES`
- `ERROR`
    - An `ERROR` message with `ErrorCodes` other than `LargeResponse` can be placed in `ResponseToLargeRequest` of `CHUNK_SEND_ACK` response.

686 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 (e.g., message transcript, transcript, etc...) that are used to verify or generate a signature or other cryptographic information. Signature generation, signature verification and other cryptographic computation operate on the Large SPDM messages, themselves, as defined in other parts of this specification.

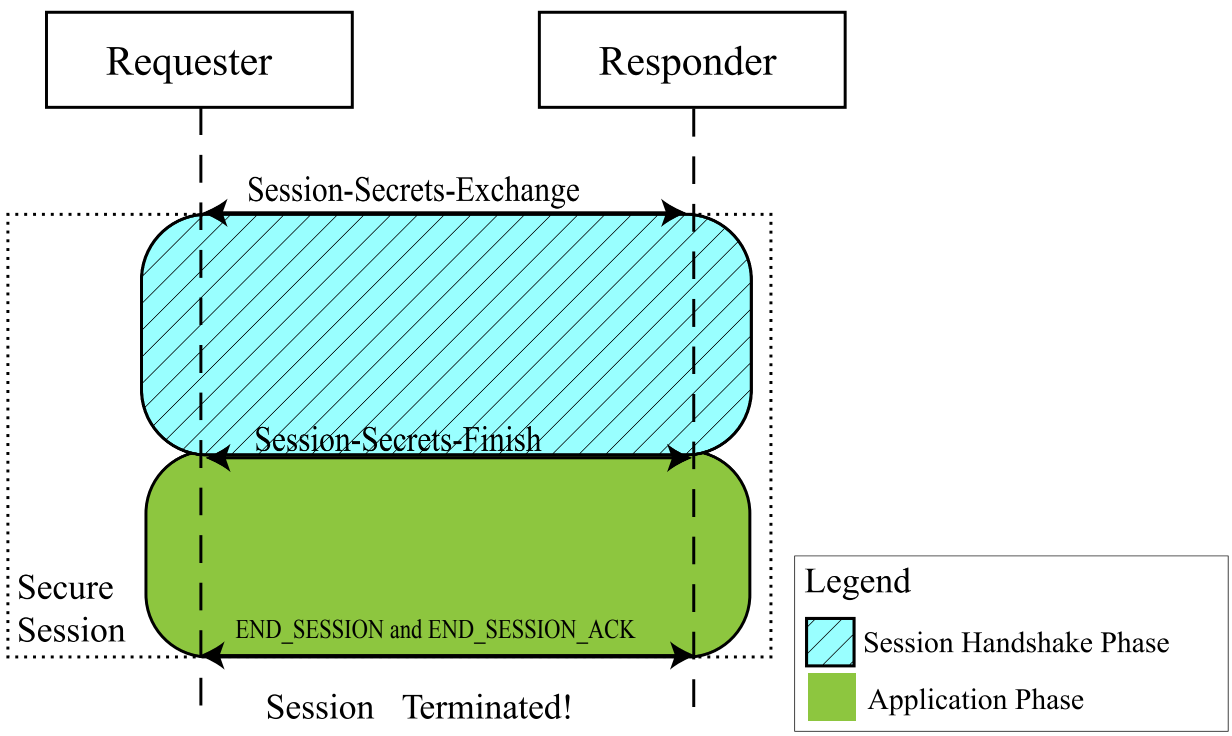
687 The response to a `CHUNK_SEND` or `CHUNK_GET` request, themselves, shall not be `ErrorCode == ResponseNotReady`. However, the `ResponseToLargeRequest` can contain an `ERROR` message with `ErrorCode == ResponseNotReady`.

688 # 11 Session

689 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 or both of encryption or message authentication.

690 There are three phases in a session, as Session phases shows: the handshake, the application, and termination.

691 **Session phases**

692 

693 ## 11.1 Session handshake phase

694 The session handshake phase begins with either `KEY_EXCHANGE` or `PSK_EXCHANGE`. This phase also allows for authentication of the Requester if the Responder indicated that earlier in `ALGORITHMS` response. Furthermore, this phase of the session uses the handshake secrets to secure the communication as described in the Key Schedule.

695 The purpose of this phase is to build trust between the Responder and Requester, first, before either side can send application data. Additionally, it also ensures the integrity of the handshake and to a certain degree, synchronicity with the derived handshake secrets.

696 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 `ALGORITHMS` message. 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.

697 If an error occurs in this phase with `ErrorCode = DecryptError`, the session shall immediately terminate and proceed to session termination.

698 A successful handshake ends with either `FINISH_RSP` or `PSK_FINISH_RSP` and the application phase begins.

699 11.2 Application phase

700 Once the handshake completes and all validation passes, the session reaches the application phase where either the Responder and Requester may send application data.

701 The application phase ends when either the `HEARTBEAT` requirements fail, `END_SESSION` or an `ERROR` message with `ErrorCode = DecryptError`. The next phase is the session termination phase.

702 11.3 Session termination phase

703 This phase signals the end of the Application phase and the enactment of internal clean-up procedures by the endpoints. Requesters and Responders may have various reasons for terminating a session, outside the scope of this specification.

704 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 may simply be an internal phase with no explicit SPDM messages sent or received.

705 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 may have other internal data associated with a session that they should also clean up.

706 11.4 Simultaneous active sessions

707 If a Responder supports key exchanges, the maximum number of simultaneous active sessions shall be a minimum of one. If the `KEY_EXCHANGE` or `PSK_EXCHANGE` request will exceed the maximum number of simultaneous active sessions of the Responder, the Responder shall respond with an `Errorcode = SessionLimitExceeded`.

708 This specification does not prohibit concurrent sessions in which the same Requester and Responder reverses role. 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. Since these two sessions are distinct and separate, the two endpoints should ensure they do not mix sessions. To ensure proper session handling, each endpoint should ensure their portion of the session IDs are unique at time of Session-Secrets-Exchange. This would form a final unique session ID for that new session. Additionally, the endpoints may use information at the transport layer to further ensure proper handling of sessions.

709 11.5 Records and session ID

710 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 or both encrypted or authenticated. 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.

711 The actual format and other details of a record is outside the scope of this specification. It is generally assumed that the transport protocol will define the format and other details of the record.

712 # 12 Key schedule

713 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 `HMAC` as defined by RFC2104 and `HKDF-Expand` as described in RFC5869. SPDM defines the following additional functions:

```
BinConcat(Length, Version, Label, Context)
```

714 where `BinConcat` shall be the concatenation of binary data, in the order shown in BinConcat Details Table:

715 **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	Little	Hash.Length

716 If Context is null, then `BinConcat` is the concatenation of the first three components only.

717 **Version details**

SPDM version	Version text
SPDM 1.1	"spdm1.1 "

718 The `HKDF-Expand` function prototype, as used by the default SPDM key schedule, is as follows:

```
    HKDF-Expand(secret, context, Hash.Length)
```

719 The `HMAC-Hash` function prototype is described as follows:

```
HMAC-Hash(salt, IKM);
```

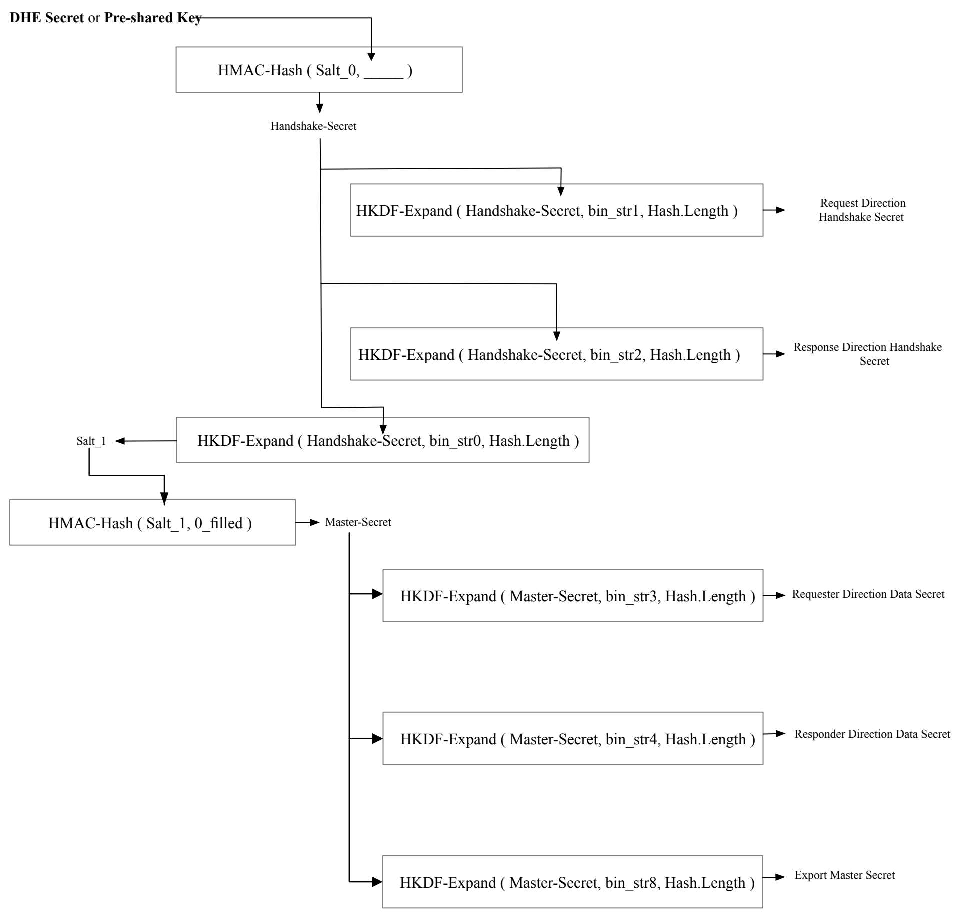
720 where IKM is the Input Keying Material and HMAC-Hash uses `HMAC` as defined in RFC2104.

721 For `HKDF-Expand` and `HMAC-Hash`, the hash function shall be the selected hash function in `ALGORITHMS` response. `Hash.Length` shall be the length of the output of the hash function selected by the `ALGORITHMS` response.

722 Both Responder and Requester shall use the key schedule shown in the Key Schedule Figure.

723 **Key schedule**

724



725 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 a name for clarity.

726 The Key Schedule table accompanies the figure to complete the Key Schedule. The Responder and Requester shall also adhere to the definition of this table.

727 **Key schedule**

Variable	Definition
Salt_0	A zero filled array of Hash.Length length.
0_filled	A zero filled array of Hash.Length length.
bin_str0	BinConcat(Hash.Length, Version, "derived", NULL).
bin_str1	BinConcat(Hash.Length, Version, "req hs data", TH1).
bin_str2	BinConcat(Hash.Length, Version, "rsp hs data", TH1).
bin_str3	BinConcat(Hash.Length, Version, "req app data", TH2)
bin_str4	BinConcat(Hash.Length, Version, "rsp app data", TH2)
DHE Secret	This shall be the secret derived from `KEY_EXCHANGE/KEY_EXCHANGE_RSP`
Pre-shared Key	PSK

728 Note: With common hash functions, any label longer than 12 characters requires an additional iteration of the hash function to compute. As in RFC8446 the labels defined above have all been chosen to fit within this limit.

729 12.1 DHE secret computation

730 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.

731 For `ffdhe2048`, `ffdhe4096`, `secp256r1`, `secp384r1` and `secp521r1`, the format and computation of the DHE secret shall be the shared secret as defined in section 7.4 of RFC 8446.

732 For `SM2_P256`, 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 in order to arrive to the same secret value. Furthermore, K_A and K_B utilizes a KDF, also defined by GB/T 32918.3-2016, that allows for a flexible hash algorithm. This hash algorithm shall be the selected hashing algorithm in `BashHashSel` or `ExtHashSel`.

733 12.2 Transcript hash in key derivation

734 There are two transcript hashes used in the key schedule, namely, **TH1** and **TH2**.

735 ## 12.3 TH1 definition

736 If the Requester and Responder used `KEY_EXCHANGE/KEY_EXCHANGE_RSP` to exchange initial keying information, then **TH1** shall be the output of applying the negotiated hash function to the concatenation of the following:

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

737 If the Requester and Responder used `PSK_EXCHANGE/PSK_EXCHANGE_RSP` to exchange initial keying information, then **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 the `ResponderVerifyData` field

738 ## 12.4 TH2 definition

739 If the Requester and Responder used `KEY_EXCHANGE/KEY_EXCHANGE_RSP` to exchange initial keying information, then **TH2** shall be the output of applying the negotiated hash function to the concatenation of the following:

1. `VCA`
2. Hash of the specified certificate chain in DER format (i.e., KEY_EXCHANGE Param2) or hash of the public key in its provisioned format, if a certificate is not used
3. `[KEY_EXCHANGE].*`
4. `[KEY_EXCHANGE_RSP].*`
5. Hash of the specified certificate chain in DER format (i.e., FINISH's Param2) or hash of the public key in its provisioned format, if a certificate is not used. (Valid only in mutual authentication)
6. `[FINISH].*`
7. `[FINISH_RSP].*`

740 If the Requester and Responder used `PSK_EXCHANGE/PSK_EXCHANGE_RSP` to exchange initial keying information, then **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)

741 12.5 Key schedule major secrets

742 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)

743 Each secret applies in a certain direction of transmission and only valid during a certain time frame. These four major secrets, each, will be used to derive their respective encryption key and IV to be used in the AEAD function as selected in the `ALGORITHMS` response.

744 12.5.1 Request-direction handshake secret

745 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`.

746 12.5.2 Response-direction handshake secret

747 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`.

748 12.5.3 Requester-direction data secret

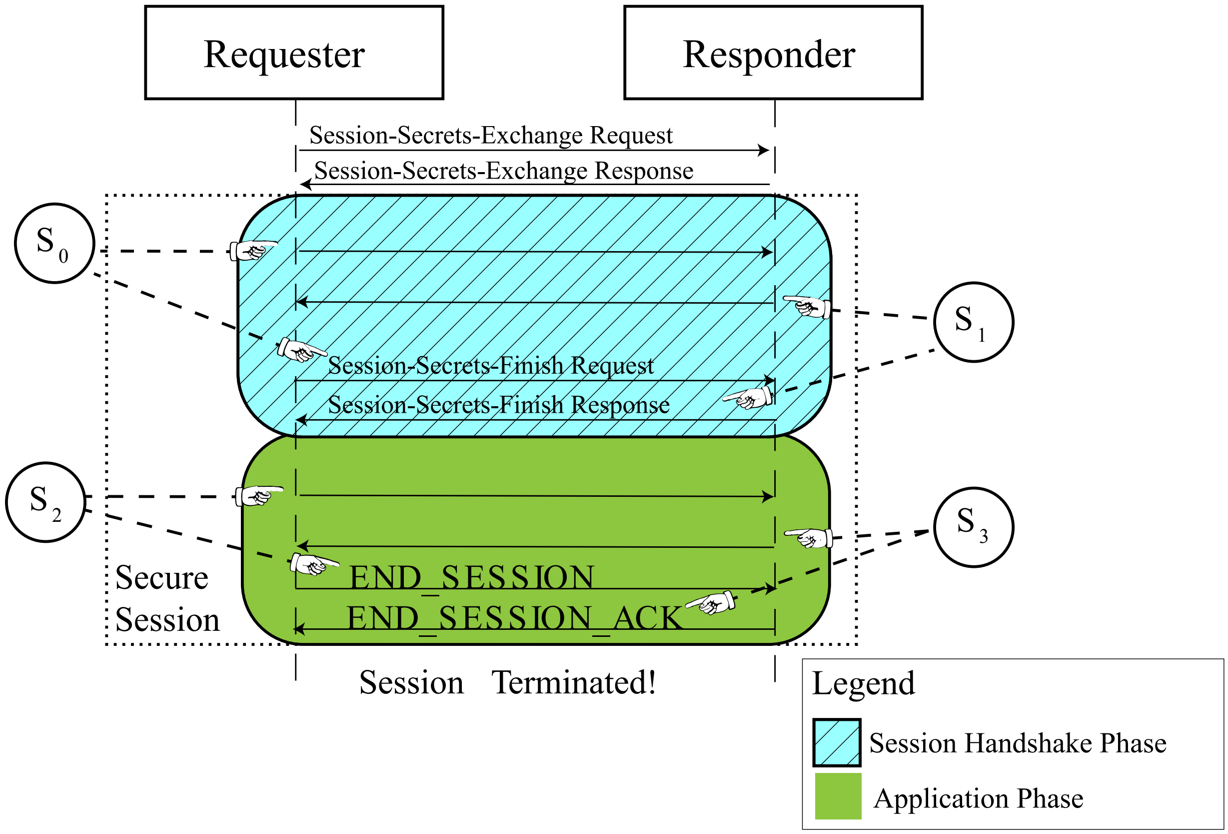
749 This secret shall be used for any data transmitted during the application phase of the session. This secret shall only be applied for all data traveling from the Requester to the Responder.

750 12.5.4 Responder-direction data secret

751 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.

752 The Secrets Usage Figure illustrates where each of the major secrets are used as described previously.

753 **Secrets usage**

754 

755 12.6 Encryption key and IV derivation

756 For each key schedule major secret, the following function shall be applied to obtain the encryption key and IV value.

```
EncryptionKey = HDKF-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);
```

757 Both `key_length` and `iv_length` shall be the lengths associated with the selected AEAD algorithm in `ALGORITHMS` message.

758 12.7 finished_key derivation

759 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);
```

760 The handshake-secret shall either be request-direction handshake secret or response-direction handshake secret.

761 12.8 Deriving additional keys from the Export Master Secret

762 After a successful SPDM key exchange, additional keys can be derived from the Export Master Secret. How keys are derived 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);
```

763 12.9 Major secrets update

764 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.

765 The major secrets are re-keyed as a result of this. 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);
```

766 In computing the new secret, `current_secret` shall either be the current Requester-Direction Data Secret or Responder-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.

767 # 13 Application data

768 SPDM utilizes authenticated encryption with associated data (AEAD) cipher algorithms in much the same way that TLS 1.3 does to protect both the confidentiality and integrity of data that shall remain secret, as well as the integrity of data that need to be transmitted in the clear, such as protocol headers, but shall be protected from manipulation. AEAD algorithms provide both encryption and message authentication. Each algorithm specifies the 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.

769 AEAD functions shall provide the following functions and comply with the requirements defined in RFC5116:

```
AEAD_Encrypt(encryption_key, nonce, associated_data, plaintext);
AEAD_Decrypt(encryption_key, nonce, associated_data, ciphertext);
```

770 where

Value	Description
`AEAD_Encrypt`	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`	Function that verifies the MAC and if validation is successful, fully decrypts the `ciphertext` and produces the original `plaintext`.
`encryption_key`	Derived encryption key for the respective direction. For details, see the Key schedule clause.
`nonce`	Nonce computation. For details, see the Nonce derivation clause.
`associated_data`	Associated data.
`plaintext`	Data to encrypt.
`ciphertext`	Data to decrypt.

771 ## 13.1 Nonce derivation

772 Certain AEAD ciphers have specific requirements on nonce construction, as their security properties may be compromised by the accidental reuse of a nonce value. Implementations should follow the requirements, such as those provided in RFC5116 for nonce derivation.

773 # 14 General opaque data

774 The General opaque data format allows for a mixture of vendors, standard organizations or transport-specific data to accompany an SPDM message without namespace collisions.

775 The General opaque data table defines the format for opaque data fields in this specification. All opaque data fields in SPDM messages shall utilize the format defined by the General opaque data.

776 **General opaque data table**

Offset	Field	Length (bytes)	Description
0	TotalElements	1	Shall be the total number of elements in `OpaqueList`.
2	Reserved	3	Reserved
8+	OpaqueList	Variable	Shall be a list of Opaque Elements.

777 The Opaque element table defines the format for each element in `OpaqueList`.

778 **Opaque element table**

Offset	Field	Length (bytes)	Description
0	ID	1	Shall be one of the values in the `ID` column of Registry or standards body ID.
1	VendorLen	1	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` column of Registry and standards body ID table for the respective `ID`.
2	VendorID	VendorLen	If `VendorLen` is greater than zero, this field shall be the ID of the vendor corresponding to the `ID` field. Otherwise, this field shall be absent.
2 + VendorLen	OpaqueElementDataLen	2	Shall be the length of `OpaqueElementData`.
X : 4 + VendorLen	OpaqueElementData	Variable	Shall be the data defined by the vendor or standards body.

Offset	Field	Length (bytes)	Description
Y : X + 1	AlignPadding	1, 2 or 3	If `X` does not fall on a 4-byte boundary, this field shall be present and of the correct length to ensure `Y` ends on a 4-byte boundary. This field shall be all zeros.

779 15 Signature generation

780 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 `ALGORITHMS` message.

781 The signature generation function takes this form:

```
signature = SPDMsign(PrivKey, data_to_be_signed, context);
```

782 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

783 The function shall output a signature using `PrivKey` and a selected cryptographic signing algorithm.

784 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" and the string form of the version of this specification. This will form a 64 character string.
2. Create `spdm_context`. If the Requester is generating the signature, then `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`.

785 Here is an example, named Example 1:

786 If the version of this specification is 1.4.0, the Responder is generating a signature and `context` is "my example context". The `sdpm_prefix` is "dmtf-spdm-v1.4.0dmtf-spdm-v1.4.0dmtf-spdm-v1.4.0dmtf-spdm-v1.4.0". The `spdm_context` is "responder-my example context".

787 Next, form `combined_spdm_prefix`. The `combined_spdm_prefix` shall be the concatenation of `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 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.

788 Continuing Example 1, the Combined SPDM Prefix table 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 long.

Furthermore, a number surrounded by double quotation marks indicates the ASCII value of that number is used. See Text or string encoding for encoding rules.

789 **Combined SPDM Prefix table**

790 | Offset | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0xA | 0xB | 0xC | 0xD | 0xE | 0xF | |--------|-----|-----|--- |--- |--- |-----|--- |-----|--- |---|-----|-----|----- |-----|-----| | 0 | d | m | t | f | - | s | p | d | m | - | v | "1" | . | "4" | . | "0" | | 0x10 | d | m | t | f | - | s | p | d | m | - | v | "1" | . | "4" | . | "0" | | 0x20 | d | m | t | f | - | s | p | d | m | - | v | "1" | . | "4" | . | "0" | | 0x30 | d | m | t | f | - | s | p | d | m | - | v | "1" | . | "4" | . | "0" | | 0x40 | 0x0 | 0x0 | 0x0 | 0x0 | 0x0 | 0x0 | 0x0 | 0x0 | r | e | s | p | o | n | d | e | | 0x50 | r | - | m | y | space (0x20) | e | x | a | m | p | l | | e | space (0x20) | c | o | n | | 0x60 | t | e | x | t | | | | | | | | | | | | |

791 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`.

792 If the Responder is generating the signature, the selected cryptographic signing algorithm is indicated in exactly one of `BaseAsymSel` or `ExtAsymSel` in `ALGORITHMS` message. If the Requester is generating the signature, the selected cryptographic signing algorithm is indicated in `ReqBaseAsymAlg` of `RespAlgStruct` in `ALGORITHMS` message.

793 Because each cryptographic signing algorithm is vastly different, these clauses define the binding of `SPDMsign` to those algorithms.

794 15.1 Signing algorithms in extensions

795 If an algorithm is selected in either the `ExtAsymSel` or `AlgExternal` of `ReqBaseAsymAlg` of `RespAlgStruct` in `ALGORITHMS` response, its binding is out of scope of this specification.

796 15.2 RSA and ECDSA signing algorithms

797 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`.

798 The private key, defined by the specification for these algorithms, shall be `PrivKey`.

799 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`.

800 For ECDSA algorithms, these algorithms shall follow section 6 of FIPS PUB 186-4.

801 15.3 EdDSA signing algorithms

802 These algorithms are described in RFC 8032.

803 The private key, defined by RFC 8032, shall be `PrivKey`.

804 In the specification for these algorithms, the letter `M` denotes the message to be signed.

805 15.3.1 Ed25519 sign

806 This specification only defines Ed25519 usage and not its variants.

807 `M` shall be the concatenation of `combined_spdm_prefix` and `message_hash`.

808 15.3.2 Ed448 sign

809 This specification only defines Ed448 usage and not its variants.

810 `M` shall be the concatenation of `combined_spdm_prefix` and `message_hash`.

811 Ed448 defines a context string, `C`. `C` shall be the `spdm_context`.

812 15.4 SM2 signing algorithm

813 This algorithm is described in GB/T 32918.2-2016. GB/T 32918.2-2016 also defines the variable `M` and ID_A.

814 The private key, defined by GB/T 32918.2-2016, shall be `PrivKey`.

815 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`.

816 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`.

817 Lastly, SM2 expects a distinguishing identifier, which identifies the signer, and is indicated by the variable ID_A. If `DMTFOthername` is present in the leaf certificate, ID_A shall be the concatenation of `ub-DMTF-device-info` and any transport specific identity. If `DMTFOtherName` is not present in the leaf certificate, the ID_A shall be the transport specific identity. The transport should specify the transport specific identity.

818 16 Signature verification

819 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 `ALGORITHMS` message.

820 The signature verification function takes this form:

```
SPDMsignatureVerify(PubKey, signature, unverified_data, context);
```

821 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

822 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.

823 The verifier of the signature shall create `spdm_prefix` , `spdm_context` and `combined_spdm_context` as described in Signature generation.

824 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` .

825 If the Responder generated the signature, the selected cryptographic signature verification algorithm is indicated in exactly one of `BaseAsymSel` or `ExtAsymSel` in `ALGORITHMS` message. If the Requester generated the signature, the selected cryptographic signature verification algorithm is indicated in `ReqBaseAsymAlg` of `RespAlgStruct` in `ALGORITHMS` message.

826 Because each cryptographic signature verification algorithm is vastly different, these clauses define the binding of `SPDMsignatureVerify` to those algorithms.

827 16.1 Signature verification algorithms in extensions

828 If an algorithm is selected in either the `ExtAsymSel` or `AlgExternal` of `ReqBaseAsymAlg` of `RespAlgStruct` in `ALGORITHMS` response, its binding is out of scope of this specification.

829 16.2 RSA and ECDSA signature verification algorithms

830 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`.

831 The public key, defined in the specification for these algorithms, shall be `PubKey`.

832 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`.

833 For ECDSA algorithms, these algorithms shall follow section 6 of FIPS PUB 186-4.

834 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.

835 For ECDSA algorithms, `SPDMsignatureVerify` shall return success when the output of "Verification with the Public Key", as defined in ANSI X9.62-2005, is "valid". Otherwise, `SPDMsignatureVerify` shall return failure.

836 16.3 EdDSA signature verification algorithms

837 These algorithms are described in RFC 8032. RFC 8032, also, defines the variable `M`, `PH` and `C`.

838 The public key, also defined in RFC 8032, shall be `PubKey`.

839 In the specification for these algorithms, the letter `M` denotes the message to be signed.

840 16.3.1 Ed25519 verify

841 `M` shall be the concatenation of `combined_spdm_prefix` and `unverified_message_hash`.

842 `SPDMsignatureVerify` shall return success when step 1 does not result in an invalid signature and 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.

843 16.3.2 Ed448 verify

844 `M` shall be the concatenation of `combined_spdm_prefix` and `unverified_message_hash`.

845 Ed448 defines a context string, `C`. `C` shall be the `spdm_context`.

846 `SPDMsignatureVerify` shall return success when step 1 does not result in an invalid signature and 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.

847 16.4 SM2 signature verification algorithm

848 This algorithm is described in GB/T 32918.2-2016, which also defines the variable `M` and ID_A.

849 The public key, also defined in GB/T 32918.2-2016, shall be `PubKey`.

850 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`.

851 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`.

852 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.

853 `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.

854 17 General ordering rules

855 With the exception of `GET_VERSION`, a Responder shall either return an `ERROR` message with `ErrorCode=UnexpectedRequest` or silently discard the request if the request is sent out of order. Additionally, the Responder may continue to silently discard all requests or return an `ERROR` message with `ErrorCode=RequestResynch` until the Requester issues a `GET_VERSION`. A Requester may retry messages but the retries shall be identical to the first, excluding transport variances. However, if the Responder sees two or more non-identical `GET_CAPABILITIES` or `NEGOTIATE_ALGORITHMS`, the Responder shall return an `ERROR` message with `ErrorCode=UnexpectedRequest` or silently discard non-identical messages. Furthermore, the Responder may continue to silently discard all messages or return an `ERROR` message until the Requester issues a `GET_VERSION`.

856 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 may lead to an authentication failure, session handshake failures and other failures. The response to a retried request should be identical to the original response.

857 18 DMTF Event Types

858 The DMTF Event Types table shows the supported DMTF event types for the DMTF event group.

859 **DMTF event types table**

Event Type	Event Name	Requirement	Description
0	Reserved	Reserved	Reserved.
1	MeasurementEvent	Optional	A measurement changed.
2	EventLost	Mandatory	Events were lost.
All others	Reserved	Reserved	Reserved.

860 18.1 Event type details

861 Each DMTF event type has its own event specific information, referred to as `EventDetail`, to describe the event. These clauses describes the format for each DMTF event type. The event types are listed in the DMTF event types table.

862 18.1.1 Measurement Event

863 The measurement event (`EventType == MeasurementEvent`) notifies the Event Recipient when a certain measurement has changed and its new measurement. The `EventDetail` format for this measurement is the same format as the Measurement block format.

864 For this event type, the `MeasurementSpecification` field of the measurement block shall be the same measurement specification as selected by the Responder in the `MeasurementSpecificationSel` field of `ALGORITHMS` response.

865 When the `MeasurementSpecification` is DMTF and the measurement event is for a raw bit stream, the size of `DMTFSpecMeasurementValue` shall be from one to 100 bytes, inclusively. The Event Recipient is expected to retrieve the raw bit stream using `GET_MEASUREMENT` Request.

866 18.1.2 Event Lost

867 This event (`EventType == EventLost`) notifies the Event Recipient that certain events are lost. The reasons for event lost are varied and numerous but some examples are lost in transport or lost due to insufficient resources. This event shall always be resent indefinitely until the Event Recipient acknowledges it. Resending this event means this event was not acknowledged previously.

868 The Event lost format table describes the format for `EventDetails` .

869 **Event lost format**

Offset	Field	Size (bytes)	Value
0	LastAckedEventID	4	Shall be the last event ID acknowledged by the Event Recipient.
4	CurrentEventID	4	Shall be the current event ID.

870 If the Event Notifier cannot or can no longer track the information in Event lost format, then `LastAckedEventID` and `CurrentEventID` shall be both 0xFFFF_FFFF.

871 When resending an event lost event, the Event Notifier can update the fields in Event lost format if new events are lost since the previous send. The `LostEventList` shall be cleared when the Event Recipient acknowledges this event.

872 # 19 ANNEX A (informative) TLS 1.3

873 This specification heavily models TLS 1.3. TLS 1.3 and consequently this specification assumes the transport layers provide these 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.

874 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 DTLS 1.3, which at the time of this specification is still in draft form.

875 # 20 ANNEX B (normative) Device certificate example

876 The Device certificate example shows an example device certificate:

877 **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 = w0123456
        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-----
```

878 21 ANNEX C (informative) OID reference

879 The following table lists all Object Identifiers (OIDs) defined in this specification.

OID	Identifier Name	Definition	Use
{ 1 3 6 1 4 1 412 274 1 }	`id-DMTF-device-info`	Leaf certificate	Certificate device information.
{ 1 3 6 1 4 1 412 274 2 }	`id-DMTF-hardware-identity`	Identity provisioning	Hardware certificate identifier.
{ 1 3 6 1 4 1 412 274 3 }	`id-DMTF-eku-responder-auth`	Leaf certificate	Certificate Extended Key Usage - SPDM Responder Authentication.
{ 1 3 6 1 4 1 412 274 4 }	`id-DMTF-eku-requester-auth`	Leaf certificate	Certificate Extended Key Usage - SPDM Requester Authentication.
{ 1 3 6 1 4 1 412 274 5 }	`id-DMTF-mutable-certificate`	Identity provisioning	Mutable certificate identifier.

880

22 ANNEX D (informative) variable name reference

881 Throughout this document, various sizes and offsets are referred to by a variable. The following table lists variables used in this document, the definition of the variable, and the location in this document that shows how the variable is set.

Symbol	Definition	Set location
A	Number of Requester-supported extended asymmetric key signature algorithms.	NEGOTIATE_ALGORITHMS request message format
A'	Number of extended asymmetric key signature algorithms selected by the Requester.	Successful ALGORITHMS response message format
D	The size of D (and C for ECDHE) is derived from the selected DHE group.	KEY_EXCHANGE request message format
E	Number of Requester-supported extended hashing algorithms.	NEGOTIATE_ALGORITHMS request message format
E'	The number of extended hashing algorithms selected requested by the Requester.	Successful ALGORITHMS response message format
F0	The length of the `SubscribeList`.	SUBSCRIBE_EVENT_GROUP request message format
F1	The length of `Padding` added after `SubscribeList`.	SUBSCRIBE_EVENT_GROUP request message format
F2	The size of the event specific details.	Event Details
H	The output size, in bytes, of the hash algorithm agreed upon in `NEGOTIATE_ALGORITHMS`.	Successful ALGORITHMS response message format
J	The length of the `UnsubscribeList`.	SUBSCRIBE_EVENT_GROUP request message format
MS	The length of the cryptographic hash or raw bit stream, as indicated in `DMTFSpecMeasurementValueType[7]`.	DMTF measurement specification format
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.	Successful VERSION response message format
Q	Length of the ResponderContext.	PSK_EXCHANGE_RSP response message format

Symbol	Definition	Set location
P	Length of the PSKHint.	PSK_EXCHANGE request message format
R	Length of the RequesterContext.	PSK_EXCHANGE request message format
SigLen	The size of the asymmetric-signing algorithm output, in bytes, that the Responder selected through the last ALGORITHMS response message.	Successful ALGORITHMS response message format

882 # 23 ANNEX E (informative) change log

883 ## 23.1 Version 1.0.0 (2019-10-16)

- Initial Release

884 ## 23.2 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 are 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.

885 ## 23.3 Version 1.2.0 (Pending)

- Fix improper reference in `DMTFSpecMeasurementValue` field in "Measurement field format when MeasurementSpecification field is Bit 0 = DMTF" table.
- Certificate digests in `DIGEST` calculation clarified.

- Format of certificate in `CertChain` parameter of `CERTIFICATE` message clarified.
- Validity period of X.509v3 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 occurs 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 SPDM request and response messages validity table.
- 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 cental location in this specification where it explains the appropriate value to populate for this field.
- Clarified use case for `Token` field in `ResponseNotReady`.
- Renamed `Measurement field format when MeasurementSpecification field is Bit 0 = DMTF` table to 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.
- 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.
- Allow Responder to specify hash algorithm for each index of measurement.
- 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.
- New:

- Added support for `AliasCert` s.
  - Compliant Requesters must support a Responder that uses either `DeviceCerts` or `AliasCert` s.
- Added Certain error handling in encapsulated flows
- Added Slot 0 certificate provisioning methodology.
- Added Allowance for encapsulated requests.
- Added Event mechanism and DMTF event type.
- 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 (Leaf certificate)
  - Certificate Extended Key Usage - SPDM Requester Authentication (Leaf certificate)
  - 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.

886 24 Bibliography

887 DMTF DSP4014, *DMTF Process for Working Bodies 2.6*.