Trusted Platform Module Library

Part 3: Commands

Family “2.0”

Level 00 Revision 01.59

November 8, 2019

Published

Contact: admin@trustedcomputinggroup.org

TCG Published

TCG

CONTENTS

Tables

Trusted Platform Module Library
Part 3: Commands

1 Scope

This TPM 2.0 Part 3 of the *Trusted Platform Module Library* specification contains the definitions of the TPM commands. These commands make use of the constants, flags, structures, and union definitions defined in TPM 2.0 Part 2.

The detailed description of the operation of the commands is written in the C language with extensive comments. The behavior of the C code in this TPM 2.0 Part 3 is normative but does not fully describe the behavior of a TPM. The combination of this TPM 2.0 Part 3 and TPM 2.0 Part 4 is sufficient to fully describe the required behavior of a TPM.

The code in parts 3 and 4 is written to define the behavior of a compliant TPM. In some cases (e.g., firmware update), it is not possible to provide a compliant implementation. In those cases, any implementation provided by the vendor that meets the general description of the function provided in TPM 2.0 Part 3 would be compliant.

The code in parts 3 and 4 is not written to meet any particular level of conformance nor does this specification require that a TPM meet any particular level of conformance.

2 Terms and Definitions

For the purposes of this document, the terms and definitions given in TPM 2.0 Part 1 apply.

3 Symbols and abbreviated terms

For the purposes of this document, the symbols and abbreviated terms given in TPM 2.0 Part 1 apply.

4 Notation

4.1 Introduction

For the purposes of this document, the notation given in TPM 2.0 Part 1 applies.

Command and response tables use various decorations to indicate the fields of the command and the allowed types. These decorations are described in this clause.

4.2 Table Decorations

The symbols and terms in the Notation column of Table 1 are used in the tables for the command schematics. These values indicate various qualifiers for the parameters or descriptions with which they are associated.

Table 1 — Command Modifiers and Decoration

Notation	Meaning
+	A Type decoration – When appended to a value in the Type column of a command, this symbol indicates that the parameter is allowed to use the “null” value of the data type (see in TPM 2.0 Part 2, *Conditional Types*). The null value is usually TPM_RH_NULL for a handle or TPM_ALG_NULL for an algorithm selector. NOTE This decoration is not appended to response parameters.
@	A Name decoration – When this symbol precedes a handle parameter in the “Name” column, it indicates that an authorization session is required for use of the entity associated with the handle. If a handle does not have this symbol, then an authorization session is not allowed.
+PP	A Description modifier – This modifier may follow TPM_RH_PLATFORM in the “Description” column to indicate that Physical Presence is required when *platformAuth*/*platformPolicy* is provided.
+{PP}	A Description modifier – This modifier may follow TPM_RH_PLATFORM to indicate that Physical Presence may be required when *platformAuth*/*platformPolicy* is provided. The commands with this notation may be in the *setList* or *clearList* of TPM2_PP_Commands().
{NV}	A Description modifier – This modifier may follow the *commandCode* in the “Description” column to indicate that the command may result in an update of NV memory and be subject to rate throttling by the TPM. If the command code does not have this notation, then a write to NV memory does not occur as part of the command actions. NOTE Any command that uses authorization may cause a write to NV if there is an authorization failure. A TPM may use the occasion of command execution to update the NV copy of clock.
{F}	A Description modifier – This modifier indicates that the “flushed” attribute will be SET in the TPMA_CC for the command. The modifier may follow the *commandCode* in the “Description” column to indicate that any transient handle context used by the command will be flushed from the TPM when the command completes. This may be combined with the {NV} modifier but not with the {E} modifier. EXAMPLE 1 {NV F} EXAMPLE 2 TPM2_SequenceComplete() will flush the context associated with the *sequenceHandle*.
{E}	A Description modifier – This modifier indicates that the “extensive” attribute will be SET in the TPMA_CC for the command. This modifier may follow the *commandCode* in the “Description” column to indicate that the command may flush many objects and re-enumeration of the loaded context likely will be required. This may be combined with the {NV} modifier but not with the {F} modifier. EXAMPLE 1 {NV E} EXAMPLE 2 TPM2_Clear() will flush all contexts associated with the Storage hierarchy and the Endorsement hierarchy.

Notation	Meaning
Auth Index:	A Description modifier – When a handle has a “@” decoration, the “Description” column will contain an “Auth Index:” entry for the handle. This entry indicates the number of the authorization session. The authorization sessions associated with handles will occur in the session area in the order of the handles with the “@” modifier. Sessions used only for encryption/decryption or only for audit will follow the handles used for authorization.
Auth Role:	A Description modifier – This will be in the “Description” column of a handle with the “@” decoration. It may have a value of USER, ADMIN or DUP. If the handle has the Auth Role of USER and the handle is an Object, the type of authorization is determined by the setting of *userWithAuth* in the Object's attributes. If the handle is TPM_RH_OWNER, TPM_RH_ENDORSEMENT, or TPM_RH_PLATFORM, operation is as if *userWithAuth* is SET. If the handle references an NV Index, then the allowed authorizations are determined by the settings of the attributes of the NV Index as described in TPM 2.0 Part 2, "TPMA_NV (NV Index Attributes)." If the Auth Role is ADMIN and the handle is an Object, the type of authorization is determined by the setting of *adminWithPolicy* in the Object's attributes. If the handle is TPM_RH_OWNER, TPM_RH_ENDORSEMENT, or TPM_RH_PLATFORM, operation is as if *adminWithPolicy* is SET. If the handle is an NV index, operation is as if *adminWithPolicy* is SET (see 5.6 e)2)). If the DUP role is selected, authorization may only be with a policy session (DUP role only applies to Objects). When either ADMIN or DUP role is selected, a policy command that selects the command being authorized is required to be part of the policy. EXAMPLE TPM2_Certify requires the ADMIN role for the first handle (*objectHandle*). The policy authorization for objectHandle is required to contain TPM2_PolicyCommandCode(*commandCode* == TPM_CC_Certify). This sets the state of the policy so that it can be used for ADMIN role authorization in TPM2_Certify().

4.3 Handle and Parameter Demarcation

The demarcations between the header, handle, and parameter parts are indicated by:

Table 2 — Separators

Separator	Meaning
	the values immediately following are in the handle area
	the values immediately following are in the parameter area

4.4 AuthorizationSize and ParameterSize

Authorization sessions are not shown in the command or response schematics. When the tag of a command or response is TPM_ST_SESSIONS, then a 32-bit value will be present in the command/response buffer to indicate the size of the authorization field or the parameter field. This value shall immediately follow the handle area (which may contain no handles). For a command, this value (*authorizationSize*) indicates the size of the Authorization Area and shall have a value of 9 or more. For a response, this value (*parameterSize*) indicates the size of the parameter area and may have a value of zero.

If the *authorizationSize* field is present in the command, *parameterSize* will be present in the response, but only if the *responseCode* is TPM_RC_SUCCESS.

When authorization is required to use the TPM entity associated with a handle, then at least one session will be present. To indicate this, the command *tag* Description field contains TPM_ST_SESSIONS. Addional sessions for audit, encrypt, and decrypt may be present.

When the command *tag* Description field contains TPM_ST_NO_SESSIONS, then no sessions are allowed and the *authorizationSize* field is not present.

When a command allows use of sessions when not required, the command *tag* Description field will indicate the types of sessions that may be used with the command.

4.5 Return Code Alias

For the RC_FMT1 return codes that may add a parameter, handle, or session number, the prefix TPM_RCS_ is an alias for TPM_RC_.

TPM_RC_n is added, where n is the parameter, handle, or session number. In addition, TPM_RC_H is added for handle, TPM_RC_P for parameter, and TPM_RC_S for session errors.

NOTE TPM_RCS_ is a programming convention. Programmers should only add numbers to TPM_RCS_ return codes, never TPM_RC_ return codes. Only return codes that can have a number added have the TPM_RCS_ alias defined. Attempting to use a TPM_RCS_ return code that does not have the TPM_RCS_ alias will cause a compiler error.

EXAMPLE 1 Since TPM_RC_VALUE can have a number added, TPM_RCS_VALUE is defined. A program can use the construct "TPM_RCS_VALUE + number". Since TPM_RC_SIGNATURE cannot have a number added, TPM_RCS_SIGNATURE is not defined. A program using the construct "TPM_RCS_SIGNATURE + number" will not compile, alerting the programmer that the construct is incorrect.

By convention, the number to be added is of the form RC_CommandName_ParameterName where CommmandName is the name of the command with the TPM2_ prefix removed. The parameter name alone is insufficient because the same parameter name could be in a different position in different commands.

EXAMPLE 2 TPM2_HMAC_Start with parameters that result in TPM_ALG_NULL as the hash algorithm will returns TPM_RC_VALUE plus the parameter number. Since *hashAlg* is the second parameter, This code results:

```
#define RC_HMAC_Start_hashAlg          (TPM_RC_P + TPM_RC_2)

return TPM_RCS_VALUE + RC_HMAC_Start_hashAlg;
```

5 Command Processing

5.1 Introduction

This clause defines the command validations that are required of any implementation and the response code returned if the indicated check fails. Unless stated otherwise, the order of the checks is not normative and different TPM may give different responses when a command has multiple errors.

In the description below, some statements that describe a check may be followed by a response code in parentheses. This is the normative response code should the indicated check fail. A normative response code may also be included in the statement.

5.2 Command Header Validation

Before a TPM may begin the actions associated with a command, a set of command format and consistency checks shall be performed. These checks are listed below and should be performed in the indicated order.

a) The TPM shall successfully unmarshal a TPMI_ST_COMMAND_TAG and verify that it is either TPM_ST_SESSIONS or TPM_ST_NO_SESSIONS (TPM_RC_BAD_TAG).

b) The TPM shall successfully unmarshal a UINT32 as the *commandSize*. If the TPM has an interface buffer that is loaded by some hardware process, the number of octets in the input buffer for the command reported by the hardware process shall exactly match the value in *commandSize* (TPM_RC_COMMAND_SIZE).

NOTE A TPM may have direct access to system memory and unmarshal directly from that memory.

c) The TPM shall successfully unmarshal a TPM_CC and verify that the command is implemented (TPM_RC_COMMAND_CODE).

5.3 Mode Checks

The following mode checks shall be performed in the order listed:

a) If the TPM is in Failure mode, then the *commandCode* is TPM_CC_GetTestResult or TPM_CC_GetCapability (TPM_RC_FAILURE) and the command *tag* is TPM_ST_NO_SESSIONS (TPM_RC_FAILURE).

NOTE 1 In Failure mode, the TPM has no cryptographic capability and processing of sessions is not supported.

b) The TPM is in Field Upgrade mode (FUM), the *commandCode* is TPM_CC_FieldUpgradeData (TPM_RC_UPGRADE).

c) If the TPM has not been initialized (TPM2_Startup()), then the *commandCode* is TPM_CC_Startup (TPM_RC_INITIALIZE).

NOTE 2 The TPM may enter Failure mode during _TPM_Init processing, before TPM2_Startup(). Since the platform firmware cannot know that the TPM is in Failure mode without accessing it, and since the first command is required to be TPM2_Startup(), the expected sequence will be that platform firmware (the CRTM) will issue TPM2_Startup() and receive TPM_RC_FAILURE indicating that the TPM is in Failure mode.

There may be failures where a TPM cannot record that it received TPM2_Startup(). In those cases, a TPM in failure mode may process TPM2_GetTestResult(), TPM2_GetCapability(), or the field upgrade commands. As a side effect, that TPM may process TPM2_GetTestResult(), TPM2_GetCapability() or the field upgrade commands before TPM2_Startup().

This is a corner case exception to the rule that TPM2_Startup() must be the first command.

The mode checks may be performed before or after the command header validation.

5.4 Handle Area Validation

After successfully unmarshaling and validating the command header, the TPM shall perform the following checks on the handles and sessions. These checks may be performed in any order.

NOTE 1 A TPM is required to perform the handle area validation before the authorization checks because an authorization cannot be performed unless the authorization values and attributes for the referenced entity are known by the TPM. For them to be known, the referenced entity must be in the TPM and accessible.

a) The TPM shall successfully unmarshal the number of handles required by the command and validate that the value of the handle is consistent with the command syntax. If not, the TPM shall return TPM_RC_VALUE.

NOTE 2 The TPM may unmarshal a handle and validate that it references an entity on the TPM before unmarshaling a subsequent handle.

NOTE 3 If the submitted command contains fewer handles than required by the syntax of the command, the TPM may continue to read into the next area and attempt to interpret the data as a handle.

b) For all handles in the handle area of the command, the TPM will validate that the referenced entity is present in the TPM.

   1) If the handle references a transient object, the handle shall reference a loaded object (TPM_RC_REFERENCE_H0 + N where N is the number of the handle in the command).

      NOTE 4 If the hierarchy for a transient object is disabled, then the transient objects will be flushed so this check will fail.

   2) If the handle references a persistent object, then

      i) the hierarchy associated with the object (platform or storage, based on the handle value) is enabled (TPM_RC_HANDLE);

      ii) the handle shall reference a persistent object that is currently in TPM non-volatile memory (TPM_RC_HANDLE);

      iii) if the handle references a persistent object that is associated with the endorsement hierarchy, that the endorsement hierarchy is not disabled (TPM_RC_HANDLE); and

         NOTE 5 The reference implementation keeps an internal attribute, passed down from a primary key to its descendents, indicating the object's hierarchy.

      iv) if the TPM implementation moves a persistent object to RAM for command processing then sufficient RAM space is available (TPM_RC_OBJECT_MEMORY).

   3) If the handle references an NV Index, then

      i) an Index exists that corresponds to the handle (TPM_RC_HANDLE); and

      ii) the hierarchy associated with the existing NV Index is not disabled (TPM_RC_HANDLE).

      iii) If the command requires write access to the index data then TPMA_NV_WRITELOCKED is not SET (TPM_RC_NV_LOCKED)

      iv) If the command requires read access to the index data then TPMA_NV_READLOCKED is not SET (TPM_RC_NV_LOCKED)

   4) If the handle references a session, then the session context shall be present in TPM memory (TPM_RC_REFERENCE_H0 + N).

   5) If the handle references a primary seed for a hierarchy (TPM_RH_ENDORSEMENT, TPM_RH_OWNER, or TPM_RH_PLATFORM) then the enable for the hierarchy is SET (TPM_RC_HIERARCHY).

   6) If the handle references a PCR, then the value is within the range of PCR supported by the TPM (TPM_RC_VALUE)

      NOTE 6 In the reference implementation, this TPM_RC_VALUE is returned by the unmarshaling code for a TPMI_DH_PCR.

5.5 Session Area Validation

a) If the tag is TPM_ST_SESSIONS and the command requires TPM_ST_NO_SESSIONS, the TPM will return TPM_RC_AUTH_CONTEXT.

b) If the tag is TPM_ST_NO_SESSIONS and the command requires TPM_ST_SESSIONS, the TPM will return TPM_RC_AUTH_MISSING.

c) If the tag is TPM_ST_SESSIONS, the TPM will attempt to unmarshal an *authorizationSize* and return TPM_RC_AUTHSIZE if the value is not within an acceptable range.

   1) The minimum value is (sizeof(TPM_HANDLE) + sizeof(UINT16) + sizeof(TPMA_SESSION) + sizeof(UINT16)).

2) The maximum value of authorizationSize is equal to commandSize – (sizeof(TPM_ST) + sizeof(UINT32) + sizeof(TPM_CC) + (N * sizeof(TPM_HANDLE)) + sizeof(UINT32)) where N is the number of handles associated with the *commandCode* and may be zero.

NOTE 1 (sizeof(TPM_ST) + sizeof(UINT32) + sizeof(TPM_CC)) is the size of a command header. The last UINT32 contains the authorizationSize octets, which are not counted as being in the authorization session area.

d) The TPM will unmarshal the authorization sessions and perform the following validations:

1) If the session handle is not a handle for an HMAC session, a handle for a policy session, or, TPM_RS_PW then the TPM shall return TPM_RC_HANDLE.

2) If the session is not loaded, the TPM will return the warning TPM_RC_REFERENCE_S0 + N where N is the number of the session. The first session is session zero, N = 0.

NOTE 2 If the HMAC and policy session contexts use the same memory, the type of the context must match the type of the handle.

3) If the maximum allowed number of sessions have been unmarshaled and fewer octets than indicated in *authorizationSize* were unmarshaled (that is, *authorizationSize* is too large), the TPM shall return TPM_RC_AUTHSIZE.

4) The consistency of the authorization session attributes is checked.

i) Only one session is allowed for:

(a) session auditing (TPM_RC_ATTRIBUTES) – this session may be used for encrypt or decrypt but may not be a session that is also used for authorization;

(b) decrypting a command parameter (TPM_RC_ATTRIBUTES) – this may be any of the authorization sessions, or the audit session, or a session may be added for the single purpose of decrypting a command parameter, as long as the total number of sessions does not exceed three; and

(c) encrypting a response parameter (TPM_RC_ATTRIBUTES) – this may be any of the authorization sessions, or the audit session if present, ora session may be added for the single purpose of encrypting a response parameter, as long as the total number of sessions does not exceed three.

NOTE 3 A session used for decrypting a command parameter may also be used for encrypting a response parameter.

ii) If a session is not being used for authorization, at least one of decrypt, encrypt, or audit must be SET. (TPM_RC_ATTRIBUTES).

5) An authorization session is present for each of the handles with the "@" decoration (TPM_RC_AUTH_MISSING).

5.6 Authorization Checks

After unmarshaling and validating the handles and the consistency of the authorization sessions, the authorizations shall be checked. Authorization checks only apply to handles if the handle in the command schematic has the "@" decoration. Authorization checks must be performed in this order.

a) The public and sensitive portions of the object shall be present on the TPM (TPM_RC_AUTH_UNAVAILABLE).

b) If the associated handle is TPM_RH_PLATFORM, and the command requires confirmation with physical presence, then physical presence is asserted (TPM_RC_PP).

c) If the object or NV Index is subject to DA protection, and the authorization is with an HMAC or password, then the TPM is not in lockout (TPM_RC_LOCKOUT).

NOTE 1 An object is subject to DA protection if its *noDA* attribute is CLEAR. An NV Index is subject to DA protection if its TPMA_NV_NO_DA attribute is CLEAR.

NOTE 2 An HMAC or password is required in a policy session when the policy contains TPM2_PolicyAuthValue() or TPM2_PolicyPassword().

d) If the command requires a handle to have DUP role authorization, then the associated authorization session is a policy session (TPM_RC_AUTH_TYPE).

e) If the command requires a handle to have ADMIN role authorization:

 1) If the entity being authorized is an object and its *adminWithPolicy* attribute is SET, or a hierarchy, then the authorization session is a policy session (TPM_RC_AUTH_TYPE).

 NOTE 3 If adminWithPolicy is CLEAR, then any type of authorization session is allowed.

 2) If the entity being authorized is an NV Index, then the associated authorization session is a policy session.

 NOTE 4 The only commands that are currently defined that require use of ADMIN role authorization are commands that operate on objects and NV Indices.

f) If the command requires a handle to have USER role authorization:

 1) If the entity being authorized is an object and its *userWithAuth* attribute is CLEAR, then the associated authorization session is a policy session (TPM_RC_POLICY_FAIL).

 NOTE 5 There is no check for a hierarchy, because a hierarchy operates as if userWithAuth is SET.

 2) If the entity being authorized is an NV Index;

  i) if the authorization session is a policy session;

   (a) the TPMA_NV_POLICYWRITE attribute of the NV Index is SET if the command modifies the NV Index data (TPM_RC_AUTH_UNAVAILABLE);

   (b) the TPMA_NV_POLICYREAD attribute of the NV Index is SET if the command reads the NV Index data (TPM_RC_AUTH_UNAVAILABLE);

  ii) if the authorization is an HMAC session or a password;

   (a) the TPMA_NV_AUTHWRITE attribute of the NV Index is SET if the command modifies the NV Index data (TPM_RC_AUTH_UNAVAILABLE);

   (b) the TPMA_NV_AUTHREAD attribute of the NV Index is SET if the command reads the NV Index data (TPM_RC_AUTH_UNAVAILABLE).

g) If the authorization is provided by a policy session, then:

 1) if *policySession→timeOut* has been set, the session shall not have expired (TPM_RC_EXPIRED);

 2) if *policySession→cpHash* has been set, it shall match the *cpHash* of the command (TPM_RC_POLICY_FAIL);

 3) if *policySession→commandCode* has been set, then *commandCode* of the command shall match (TPM_RC_POLICY_CC);

 4) *policySession→policyDigest* shall match the *authPolicy* associated with the handle (TPM_RC_POLICY_FAIL);

 5) if *policySession→pcrUpdateCounter* has been set, then it shall match the value of *pcrUpdateCounter* (TPM_RC_PCR_CHANGED);

 6) if *policySession→commandLocality* has been set, it shall match the locality of the command (TPM_RC_LOCALITY),

7) if *policySession→cpHash* contains a template, and the command is TPM2_Create(), TPM2_CreatePrimary(), or TPM2_CreateLoaded(), then the *inPublic* parmeter matches the contents of *policySession→cpHash;* and

8) if the policy requires that an authValue be provided in order to satisfy the policy, then *session.hmac* is not an Empty Buffer.

h) If the authorization uses an HMAC, then the HMAC is properly constructed using the *authValue* associated with the handle and/or the session secret (TPM_RC_AUTH_FAIL or TPM_RC_BAD_AUTH).

NOTE 6 A policy session may require proof of knowledge of the authValue of the object being authorized.

i) If the authorization uses a password, then the password matches the *authValue* associated with the handle (TPM_RC_AUTH_FAIL or TPM_RC_BAD_AUTH).

If the TPM returns an error other than TPM_RC_AUTH_FAIL then the TPM shall not alter any TPM state. If the TPM return TPM_RC_AUTH_FAIL, then the TPM shall not alter any TPM state other than *lockoutCount*.

NOTE 7 The TPM may decrease failedTries regardless of any other processing performed by the TPM. That is, the TPM may exit Lockout mode, regardless of the return code.

5.7 Parameter Decryption

If an authorization session has the TPMA_SESSION.*decrypt* attribute SET, and the command does not allow a command parameter to be encrypted, then the TPM will return TPM_RC_ATTRIBUTES. Otherwise, the TPM will decrypt the parameter using the values associated with the session before parsing parameters.

NOTE The size of the parameter to be encrypted can be zero.

5.8 Parameter Unmarshaling

5.8.1 Introduction

The detailed actions for each command assume that the input parameters of the command have been unmarshaled into a command-specific structure with the structure defined by the command schematic. Additionally, a response-specific output structure is assumed which will receive the values produced by the detailed actions.

NOTE An implementation is not required to process parameters in this manner or to separate the parameter parsing from the command actions. This method was chosen for the specification so that the normative behavior described by the detailed actions would be clear and unencumbered.

Unmarshaling is the process of processing the parameters in the input buffer and preparing the parameters for use by the command-specific action code. No data movement need take place but it is required that the TPM validate that the parameters meet the requirements of the expected data type as defined in TPM 2.0 Part 2.

5.8.2 Unmarshaling Errors

When an error is encountered while unmarshaling a command parameter, an error response code is returned and no command processing occurs. A table defining a data type may have response codes embedded in the table to indicate the error returned when the input value does not match the parameters of the table.

NOTE In the reference implementation, a parameter number is added to the response code so that the offending parameter can be isolated. This is optional.

In many cases, the table contains no specific response code value and the return code will be determined as defined in Table 3.

Table 3 — Unmarshaling Errors

Response Code	Meaning
TPM_RC_ASYMMETRIC	a parameter that should be an asymmetric algorithm selection does not have a value that is supported by the TPM
TPM_RC_BAD_TAG	a parameter that should be a command tag selection has a value that is not supported by the TPM
TPM_RC_COMMAND_CODE	a parameter that should be a command code does not have a value that is supported by the TPM
TPM_RC_HASH	a parameter that should be a hash algorithm selection does not have a value that is supported by the TPM
TPM_RC_INSUFFICIENT	the input buffer did not contain enough octets to allow unmarshaling of the expected data type;
TPM_RC_KDF	a parameter that should be a key derivation scheme (KDF) selection does not have a value that is supported by the TPM
TPM_RC_KEY_SIZE	a parameter that is a key size has a value that is not supported by the TPM
TPM_RC_MODE	a parameter that should be a symmetric encryption mode selection does not have a value that is supported by the TPM
TPM_RC_RESERVED	a non-zero value was found in a reserved field of an attribute structure (TPMA_)
TPM_RC_SCHEME	a parameter that should be signing or encryption scheme selection does not have a value that is supported by the TPM
TPM_RC_SIZE	the value of a size parameter is larger or smaller than allowed
TPM_RC_SYMMETRIC	a parameter that should be a symmetric algorithm selection does not have a value that is supported by the TPM
TPM_RC_TAG	a parameter that should be a structure tag has a value that is not supported by the TPM
TPM_RC_TYPE	The type parameter of a TPMT_PUBLIC or TPMT_SENSITIVE has a value that is not supported by the TPM
TPM_RC_VALUE	a parameter does not have one of its allowed values

In some commands, a parameter may not be used because of various options of that command. However, the unmarshaling code is required to validate that all parameters have values that are allowed by the TPM 2.0 Part 2 definition of the parameter type even if that parameter is not used in the command actions.

5.9 Command Post Processing

When the code that implements the detailed actions of the command completes, it returns a response code. If that code is not TPM_RC_SUCCESS, the post processing code will not update any session or audit data and will return a 10-octet response packet.

If the command completes successfully, the tag of the command determines if any authorization sessions will be in the response. If so, the TPM will encrypt the first parameter of the response if indicated by the authorization attributes. The TPM will then generate a new nonce value for each session and, if appropriate, generate an HMAC.

If authorization HMAC computations are performed on the response, the HMAC keys used in the response will be the same as the HMAC keys used in processing the HMAC in the command.

NOTE 1 This primarily affects authorizations associated with a first write to an NV Index using a bound session. The computation of the HMAC in the response is performed as if the Name of the Index did not change as a consequence of the command actions. The session binding to the NV Index will not persist to any subsequent command.

NOTE 2 The authorization attributes were validated during the session area validation to ensure that only one session was used for parameter encryption of the response and that the command allowed encryption in the response.

NOTE 3 No session nonce value is used for a password authorization but the session data is present.

Additionally, if the command is being audited by Command Audit, the audit digest is updated with the *cpHash* of the command and *rpHash* of the response.

6 Response Values

6.1 Tag

When a command completes successfully, the *tag* parameter in the response shall have the same value as the *tag* parameter in the command (TPM_ST_SESSIONS or TPM_ST_NO_SESSIONS). When a command fails (the responseCode is not TPM_RC_SUCCESS), then the *tag* parameter in the response shall be TPM_ST_NO_SESSIONS.

A special case exists when the command *tag* parameter is not an allowed value (TPM_ST_SESSIONS or TPM_ST_NO_SESSIONS). For this case, it is assumed that the system software is attempting to send a command formatted for a TPM 1.2 but the TPM is not capable of executing TPM 1.2 commands. So that the TPM 1.2 compatible software will have a recognizable response, the TPM sets *tag* to TPM_ST_RSP_COMMAND, *responseSize* to 00 00 00 $0A_{16}$ and *responseCode* to TPM_RC_BAD_TAG. This is the same response as the TPM 1.2 fatal error for TPM_BADTAG.

6.2 Response Codes

The normal response for any command is TPM_RC_SUCCESS. Any other value indicates that the command did not complete and the state of the TPM is unchanged. An exception to this general rule is that the logic associated with dictionary attack protection is allowed to be modified when an authorization failure occurs.

Commands have response codes that are specific to that command, and those response codes are enumerated in the detailed actions of each command. The codes associated with the unmarshaling of parameters are documented Table 3. Another set of response code values are not command specific and indicate a problem that is not specific to the command. That is, if the indicated problem is remedied, the same command could be resubmitted and may complete normally.

The response codes that are not command specific are listed and described in

Table 4.

The reference code for the command actions may have code that generates specific response codes associated with a specific check but the listing of responses may not have that response code listed.

Table 4 — Command-Independent Response Codes

Response Code	Meaning
TPM_RC_CANCELED	This response code may be returned by a TPM that supports command cancel. When the TPM receives an indication that the current command should be cancelled, the TPM may complete the command or return this code. If this code is returned, then the TPM state is not changed and the same command may be retried.
TPM_RC_CONTEXT_GAP	This response code can be returned for commands that manage session contexts. It indicates that the gap between the lowest numbered active session and the highest numbered session is at the limits of the session tracking logic. The remedy is to load the session context with the lowest number so that its tracking number can be updated.
TPM_RC_LOCKOUT	This response indicates that authorizations for objects subject to DA protection are not allowed at this time because the TPM is in DA lockout mode. The remedy is to wait or to exeucte TPM2_DictionaryAttackLockoutReset().
TPM_RC_MEMORY	A TPM may use a common pool of memory for objects, sessions, and other purposes. When the TPM does not have enough memory available to perform the actions of the command, it may return TPM_RC_MEMORY. This indicates that the TPM resource manager may flush either sessions or objects in order to make memory available for the command execution. A TPM may choose to return TPM_RC_OBJECT_MEMORY or TPM_RC_SESSION_MEMORY if it needs contexts of a particular type to be flushed.
TPM_RC_NV_RATE	This response code indicates that the TPM is rate-limiting writes to the NV memory in order to prevent wearout. This response is possible for any command that explicity writes to NV or commands that incidentally use NV such as a command that uses authorization session that may need to update the dictionary attack logic.
TPM_RC_NV_UNAVAILABLE	This response code is similar to TPM_RC_NV_RATE but indicates that access to NV memory is currently not available and the command is not allowed to proceed until it is. This would occur in a system where the NV memory used by the TPM is not exclusive to the TPM and is a shared system resource.
TPM_RC_OBJECT_HANDLES	This response code indicates that the TPM has exhausted its handle space and no new objects can be loaded unless the TPM is rebooted. This does not occur in the reference implementation because of the way that object handles are allocated. However, other implementations are allowed to assign each object a unique handle each time the object is loaded. A TPM using this implementation would be able to load 2^{24} objects before the object space is exhausted.
TPM_RC_OBJECT_MEMORY	This response code can be returned by any command that causes the TPM to need an object 'slot'. The most common case where this might be returned is when an object is loaded (TPM2_Load, TPM2_CreatePrimary(), or TPM2_ContextLoad()). However, the TPM implementation is allowed to use object slots for other reasons. In the reference implementation, the TPM copies a referenced persistent object into RAM for the duration of the commannd. If all the slots are previously occupied, the TPM may return this value. A TPM is allowed to use object slots for other purposes and return this value. The remedy when this response is returned is for the TPM resource manager to flush a transient object.
TPM_RC_REFERENCE_Hx	This response code indicates that a handle in the handle area of the command is not associated with a loaded object. The value of 'x' is in the range 0 to 6 with a value of 0 indicating the 1st handle and 6 representing the 7th. Upper values are provided for future use. The TPM resource manager needs to find the correct object and load it. It may then adjust the handle and retry the command. NOTE Usually, this error indicates that the TPM resource manager has a corrupted database.

Response Code	Meaning
TPM_RC_REFERENCE_Sx	This response code indicates that a handle in the session area of the command is not associated with a loaded session. The value of 'x' is in the range 0 to 6 with a value of 0 indicating the 1st session handle and 6 representing the 7th. Upper values are provided for future use. The TPM resource manager needs to find the correct session and load it. It may then retry the command. NOTE Usually, this error indicates that the TPM resource manager has a corrupted database.
TPM_RC_RETRY	the TPM was not able to start the command
TPM_RC_SESSION_HANDLES	This response code indicates that the TPM does not have a handle to assign to a new session. This respose is only returned by TPM2_StartAuthSession(). It is listed here because the command is not in error and the TPM resource manager can remedy the situation by flushing a session (TPM2_FlushContext().
TPM_RC_SESSION_MEMORY	This response code can be returned by any command that causes the TPM to need a session 'slot'. The most common case where this might be returned is when a session is loaded (TPM2_StartAuthSession() or TPM2_ContextLoad()). However, the TPM implementation is allowed to use object slots for other purposes. The remedy when this response is returned is for the TPM resource manager to flush a transient object.
TPM_RC_SUCCESS	Normal completion for any command. If the responseCode is TPM_RC_SUCCESS, then the rest of the response has the format indicated in the response schematic. Otherwise, the response is a 10 octet value indicating an error.
TPM_RC_TESTING	This response code indicates that the TPM is performing tests and cannot respond to the request at this time. The command may be retried.
TPM_RC_YIELDED	the TPM has suspended operation on the command; forward progress was made and the command may be retried. See TPM 2.0 Part 1, “Multi-tasking.” NOTE This cannot occur on the reference implementation.

7 Implementation Dependent

The actions code for each command makes assumptions about the behavior of various sub-systems. There are many possible implementations of the subsystems that would achieve equivalent results. The actions code is not written to anticipate all possible implementations of the sub-systems. Therefore, it is the responsibility of the implementer to ensure that the necessary changes are made to the actions code when the sub-system behavior changes.

8 Detailed Actions Assumptions

8.1 Introduction

The C code in the Detailed Actions for each command is written with a set of assumptions about the processing performed before the action code is called and the processing that will be done after the action code completes.

8.2 Pre-processing

Before calling the command actions code, the following actions have occurred.

- Verification that the handles in the handle area reference entities that are resident on the TPM.
- NOTE If a handle is in the parameter portion of the command, the associated entity does not have to be loaded, but the handle is required to be the correct type.
- If use of a handle requires authorization, the Password, HMAC, or Policy session associated with the handle has been verified.
- If a command parameter was encrypted using parameter encryption, it was decrypted before being unmarshaled.
- If the command uses handles or parameters, the calling stack contains a pointer to a data structure (*in*) that holds the unmarshaled values for the handles and command parameters. If the response has handles or parameters, the calling stack contains a pointer to a data structure (*out*) to hold the handles and response parameters generated by the command.
- All parameters of the *in* structure have been validated and meet the requirements of the parameter type as defined in TPM 2.0 Part 2.
- Space set aside for the out structure is sufficient to hold the largest *out* structure that could be produced by the command

8.3 Post Processing

When the function implementing the command actions completes,

- response parameters that require parameter encryption will be encrypted after the command actions complete;
- audit and session contexts will be updated if the command response is TPM_RC_SUCCESS; and
- the command header and command response parameters will be marshaled to the response buffer.

9 Start-up

9.1 Introduction

This clause contains the commands used to manage the startup and restart state of a TPM.

9.2 _TPM_Init

9.2.1 General Description

_TPM_Init initializes a TPM.

Initialization actions include testing code required to execute the next expected command. If the TPM is in FUM, the next expected command is TPM2_FieldUpgradeData(); otherwise, the next expected command is TPM2_Startup().

NOTE 1 If the TPM performs self-tests after receiving _TPM_Init() and the TPM enters Failure mode before receiving TPM2_Startup() or TPM2_FieldUpgradeData(), then the TPM may be able to accept TPM2_GetTestResult() or TPM2_GetCapability().

The means of signaling _TPM_Init shall be defined in the platform-specific specifications that define the physical interface to the TPM. The platform shall send this indication whenever the platform starts its boot process and only when the platform starts its boot process.

There shall be no software method of generating this indication that does not also reset the platform and begin execution of the CRTM.

NOTE 2 In the reference implementation, this signal causes an internal flag (*s_initialized*) to be CLEAR. While this flag is CLEAR, the TPM will only accept the next expected command described above.

9.2.2 Detailed Actions

```
#include "Tpm.h"
#include "_TPM_Init_fp.h"
 // This function is used to process a _TPM_Init indication.
LIB_EXPORT void
_TPM_Init(
    void
    )
{
    g_powerWasLost = g_powerWasLost | _plat__WasPowerLost();

#if SIMULATION && DEBUG
    // If power was lost and this was a simulation, put canary in RAM used by NV
    // so that uninitialized memory can be detected more easily
    if(g_powerWasLost)
    {
        memset(&gc, 0xbb, sizeof(gc));
        memset(&gr, 0xbb, sizeof(gr));
        memset(&gp, 0xbb, sizeof(gp));
        memset(&go, 0xbb, sizeof(go));
    }
#endif

#if SIMULATION
    // Clear the flag that forces failure on self-test
    g_forceFailureMode = FALSE;
#endif

    // Disable the tick processing
    _plat__ACT_EnableTicks(FALSE);

    // Set initialization state
    TPMInit();

    // Set g_DRTMHandle as unassigned
    g_DRTMHandle = TPM_RH_UNASSIGNED;

    // No H-CRTM, yet.
    g_DrtmPreStartup = FALSE;

    // Initialize the NvEnvironment.
    g_nvOk = NvPowerOn();

    // Initialize cryptographic functions
    g_inFailureMode = (CryptInit() == FALSE);
    if(!g_inFailureMode)
    {
        // Load the persistent data
        NvReadPersistent();

        // Load the orderly data (clock and DRBG state).
        // If this is not done here, things break
        NvRead(&go, NV_ORDERLY_DATA, sizeof(go));

        // Start clock. Need to do this after NV has been restored.
        TimePowerOn();
    }
    return;
}
```

9.3 TPM2_Startup

9.3.1 General Description

TPM2_Startup() is always preceded by _TPM_Init, which is the physical indication that TPM initialization is necessary because of a system-wide reset. TPM2_Startup() is only valid after _TPM_Init. Additional TPM2_Startup() commands are not allowed after it has completed successfully. If a TPM requires TPM2_Startup() and another command is received, or if the TPM receives TPM2_Startup() when it is not required, the TPM shall return TPM_RC_INITIALIZE.

NOTE 1 See 9.2.1 for other command options for a TPM supporting field upgrade mode.

NOTE 2 _TPM_Hash_Start, _TPM_Hash_Data, and _TPM_Hash_End are not commands and a platform-specific specification may allow these indications between _TPM_Init and TPM2_Startup().

If in Failure mode, the TPM shall accept TPM2_GetTestResult() and TPM2_GetCapability() even if TPM2_Startup() is not completed successfully or processed at all.

A platform-specific specification may restrict the localities at which TPM2_Startup() may be received.

A Shutdown/Startup sequence determines the way in which the TPM will operate in response to TPM2_Startup(). The three sequences are:

1) TPM Reset – This is a Startup(CLEAR) preceded by either Shutdown(CLEAR) or no TPM2_Shutdown(). On TPM Reset, all variables go back to their default initialization state.

   NOTE 3 Only those values that are specified as having a default initialization state are changed by TPM Reset. Persistent values that have no default initialization state are not changed by this command. Values such as seeds have no default initialization state and only change due to specific commands.

2) TPM Restart – This is a Startup(CLEAR) preceded by Shutdown(STATE). This preserves much of the previous state of the TPM except that PCR and the controls associated with the Platform hierarchy are all returned to their default initialization state;

3) TPM Resume – This is a Startup(STATE) preceded by Shutdown(STATE). This preserves the previous state of the TPM including the static Root of Trust for Measurement (S-RTM) PCR and the platform controls other than the *phEnable*.

If a TPM receives Startup(STATE) and that was not preceded by Shutdown(STATE), the TPM shall return TPM_RC_VALUE.

If, during TPM Restart or TPM Resume, the TPM fails to restore the state saved at the last Shutdown(STATE), the TPM shall enter Failure Mode and return TPM_RC_FAILURE.

On any TPM2_Startup(),

- *phEnable* shall be SET;
- all transient contexts (objects, sessions, and sequences) shall be flushed from TPM memory;

NOTE 4 See Part 1 Time for a description of the TPMS_TIME_INFO.*time* behaviour.

- use of *lockoutAuth* shall be enabled if *lockoutRecovery* is zero.

Additional actions are performed based on the Shutdown/Startup sequence.

On TPM Reset:

- *platformAuth* and *platformPolicy* shall be set to the Empty Buffer,
- For each NV Index with TPMA_NV_WRITEDEFINE CLEAR or TPMA_NV_WRITTEN CLEAR, TPMA_NV_WRITELOCKED shall be CLEAR,
- For each NV Index with TPMA_NV_ORDERLY SET, TPMA_NV_WRITTEN shall be CLEAR unless the type is TPM_NT_COUNTER,
- On a disorderly reset, advance the orderly counters,
- For each NV Index with TPMA_NV_CLEAR_STCLEAR SET, TPMA_NV_WRITTEN shall be CLEAR,
- tracking data for saved session contexts shall be set to its initial value,
- the object context sequence number is reset to zero,
- a new context encryption key shall be generated,
- TPMS_CLOCK_INFO.*restartCount* shall be reset to zero,
- TPMS_CLOCK_INFO.*resetCount* shall be incremented,
- the PCR Update Counter shall be clear to zero,

NOTE 5 Because the PCR update counter may be incremented when a PCR is reset, the PCR resets performed as part of this command can result in the PCR update counter being non-zero at the end of this command.

- *phEnableNV, shEnable* and *ehEnable* shall be SET, and
- PCR in all banks are reset to their default initial conditions as determined by the relevant platform-specific specification and the H-CRTM state (for exceptions, see TPM 2.0 Part 1*, H-CRTM before TPM2_Startup() and TPM2_Startup without H-CRTM*),
- For each ACT the timeout is reset to zero, the *signaled* attribute is set to CLEAR (if *preserveSignaled* is CLEAR), and the *authPolicy* is set to the Empty Buffer and its hashAlg is set to TPM_ALG_NULL.

NOTE 6 PCR may be initialized any time between _TPM_Init and the end of TPM2_Startup(). PCR that are preserved by TPM Resume will need to be restored during TPM2_Startup().

NOTE 7 See "Initializing PCR" in TPM 2.0 Part 1 for a description of the default initial conditions for a PCR.

On TPM Restart:

- TPMS_CLOCK_INFO.*restartCount* shall be incremented,
- *phEnableNV, shEnable* and *ehEnable* shall be SET,
- *platformAuth* and *platformPolicy* shall be set to the Empty Buffer,
- For each NV index with TPMA_NV_WRITEDEFINE CLEAR or TPMA_NV_WRITTEN CLEAR, TPMA_NV_WRITELOCKED shall be CLEAR,
- For each NV index with TPMA_NV_CLEAR_STCLEAR SET, TPMA_NV_WRITTEN shall be CLEAR, and
- PCR in all banks are reset to their default initial conditions.
- If an H-CRTM Event Sequence is active, extend the PCR designated by the platform-specific specification.
- For each ACT the timeout is reset to zero, the *signaled* attribute is set to CLEAR (if *preserveSignaled* is CLEAR), and the *authPolicy* is set to the Empty Buffer and its hashAlg is set to TPM_ALG_NULL.

On TPM Resume:

- the H-CRTM startup method is the same for this TPM2_Startup() as for the previous TPM2_Startup(); (TPM_RC_LOCALITY)
- TPMS_CLOCK_INFO.*restartCount* shall be incremented; and
- PCR that are specified in a platform-specific specification to be preserved on TPM Resume are restored to their saved state and other PCR are set to their initial value as determined by a platform-specific specification. For constraints, see TPM 2.0 Part 1, *H-CRTM before TPM2_Startup() and TPM2_Startup without H-CRTM.*
- The ACT timeout, the ACT *signaled* attribute and the ACT specific *authPolicy* values are preserved.

Other TPM state may change as required to meet the needs of the implementation.

If the *startupType* is TPM_SU_STATE and the TPM requires TPM_SU_CLEAR, then the TPM shall return TPM_RC_VALUE.

NOTE 8 The TPM will require TPM_SU_CLEAR when no shutdown was performed or after Shutdown(CLEAR).

NOTE 9 If *startupType* is neither TPM_SU_STATE nor TPM_SU_CLEAR, then the unmarshaling code returns TPM_RC_VALUE.

9.3.2 Command and Response

Table 5 — TPM2_Startup Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_Startup {NV}
TPM_SU	startupType	TPM_SU_CLEAR or TPM_SU_STATE

Table 6 — TPM2_Startup Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

9.3.3 Detailed Actions

```
#include "Tpm.h"
#include "Startup_fp.h"
#if CC_Startup  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_LOCALITY	a Startup(STATE) does not have the same H-CRTM state as the previous Startup() or the locality of the startup is not 0 pr 3
TPM_RC_NV_UNINITIALIZED	the saved state cannot be recovered and a Startup(CLEAR) is required.
TPM_RC_VALUE	start up type is not compatible with previous shutdown sequence

```
TPM_RC
TPM2_Startup(
    Startup_In      *in             // IN: input parameter list
    )
{
    STARTUP_TYPE         startup;
    BYTE                 locality = _plat__LocalityGet();
    BOOL                 OK = TRUE;
//
    // The command needs NV update.
    RETURN_IF_NV_IS_NOT_AVAILABLE;

    // Get the flags for the current startup locality and the H-CRTM.
    // Rather than generalizing the locality setting, this code takes advantage
    // of the fact that the PC Client specification only allows Startup()
    // from locality 0 and 3. To generalize this probably would require a
    // redo of the NV space and since this is a feature that is hardly ever used
    // outside of the PC Client, this code just support the PC Client needs.

// Input Validation
    // Check that the locality is a supported value
    if(locality != 0 && locality != 3)
        return TPM_RC_LOCALITY;
    // If there was a H-CRTM, then treat the locality as being 3
    // regardless of what the Startup() was. This is done to preserve the
    // H-CRTM PCR so that they don't get overwritten with the normal
    // PCR startup initialization. This basically means that g_StartupLocality3
    // and g_DrtmPreStartup can't both be SET at the same time.
    if(g_DrtmPreStartup)
        locality = 0;
    g_StartupLocality3 = (locality == 3);

#if USE_DA_USED
    // If there was no orderly shutdown, then their might have been a write to
    // failedTries that didn't get recorded but only if g_daUsed was SET in the
    // shutdown state
    g_daUsed = (gp.orderlyState == SU_DA_USED_VALUE);
    if(g_daUsed)
        gp.orderlyState = SU_NONE_VALUE;
#endif

    g_prevOrderlyState = gp.orderlyState;

    // If there was a proper shutdown, then the startup modifiers are in the
    // orderlyState. Turn them off in the copy.
    if(IS_ORDERLY(g_prevOrderlyState))
        g_prevOrderlyState &=  ~(PRE_STARTUP_FLAG | STARTUP_LOCALITY_3);
```

```
    // If this is a Resume,
    if(in->startupType == TPM_SU_STATE)
    {
        // then there must have been a prior TPM2_ShutdownState(STATE)
        if(g_prevOrderlyState != TPM_SU_STATE)
            return TPM_RCS_VALUE + RC_Startup_startupType;
        // and the part of NV used for state save must have been recovered
        // correctly.
        // NOTE: if this fails, then the caller will need to do Startup(CLEAR). The
        // code for Startup(Clear) cannot fail if the NV can't be read correctly
        // because that would prevent the TPM from ever getting unstuck.
        if(g_nvOk == FALSE)
            return TPM_RC_NV_UNINITIALIZED;
        // For Resume, the H-CRTM has to be the same as the previous boot
        if(g_DrtmPreStartup != ((gp.orderlyState & PRE_STARTUP_FLAG) != 0))
            return TPM_RCS_VALUE + RC_Startup_startupType;
        if(g_StartupLocality3 != ((gp.orderlyState & STARTUP_LOCALITY_3) != 0))
            return TPM_RC_LOCALITY;
    }
    // Clean up the gp state
    gp.orderlyState = g_prevOrderlyState;

// Internal Date Update
    if((gp.orderlyState == TPM_SU_STATE) && (g_nvOk == TRUE))
    {
        // Always read the data that is only cleared on a Reset because this is not
        // a reset
        NvRead(&gr, NV_STATE_RESET_DATA, sizeof(gr));
        if(in->startupType == TPM_SU_STATE)
        {
            // If this is a startup STATE (a Resume) need to read the data
            // that is cleared on a startup CLEAR because this is not a Reset
            // or Restart.
            NvRead(&gc, NV_STATE_CLEAR_DATA, sizeof(gc));
            startup = SU_RESUME;
        }
        else
            startup = SU_RESTART;
    }
    else
        // Will do a TPM reset if Shutdown(CLEAR) and Startup(CLEAR) or no shutdown
        // or there was a failure reading the NV data.
        startup = SU_RESET;
    // Startup for cryptographic library. Don't do this until after the orderly
    // state has been read in from NV.
    OK = OK && CryptStartup(startup);

    // When the cryptographic library has been started, indicate that a TPM2_Startup
    // command has been received.
    OK = OK && TPMRegisterStartup();

    // Read the platform unique value that is used as VENDOR_PERMANENT
    // authorization value
    g_platformUniqueDetails.t.size
        = (UINT16)_plat__GetUnique(1, sizeof(g_platformUniqueDetails.t.buffer),
                                   g_platformUniqueDetails.t.buffer);

// Start up subsystems
    // Start set the safe flag
    OK = OK && TimeStartup(startup);

    // Start dictionary attack subsystem
    OK = OK && DAStartup(startup);

    // Enable hierarchies
    OK = OK && HierarchyStartup(startup);
```

```

    // Restore/Initialize PCR
    OK = OK && PCRStartup(startup, locality);

    // Restore/Initialize command audit information
    OK = OK && CommandAuditStartup(startup);

    // Restore the ACT
    OK = OK && ActStartup(startup);

//// The following code was moved from Time.c where it made no sense
    if(OK)
    {
        switch(startup)
        {
            case SU_RESUME:
                // Resume sequence
                gr.restartCount++;
                break;
            case SU_RESTART:
                // Hibernate sequence
                gr.clearCount++;
                gr.restartCount++;
                break;
            default:
                // Reset object context ID to 0
                gr.objectContextID = 0;
                // Reset clearCount to 0
                gr.clearCount = 0;

                // Reset sequence
                // Increase resetCount
                gp.resetCount++;

                // Write resetCount to NV
                NV_SYNC_PERSISTENT(resetCount);

                gp.totalResetCount++;
                // We do not expect the total reset counter overflow during the life
                // time of TPM.  if it ever happens, TPM will be put to failure mode
                // and there is no way to recover it.
                // The reason that there is no recovery is that we don't increment
                // the NV totalResetCount when incrementing would make it 0. When the
                // TPM starts up again, the old value of totalResetCount will be read
                // and we will get right back to here with the increment failing.
                if(gp.totalResetCount == 0)
                    FAIL(FATAL_ERROR_INTERNAL);

                // Write total reset counter to NV
                NV_SYNC_PERSISTENT(totalResetCount);

                // Reset restartCount
                gr.restartCount = 0;

                break;
        }
    }
    // Initialize session table
    OK = OK && SessionStartup(startup);

    // Initialize object table
    OK = OK && ObjectStartup();

    // Initialize index/evict data.  This function clears read/write locks
    // in NV index
    OK = OK && NvEntityStartup(startup);
```

```

    // Initialize the orderly shut down flag for this cycle to SU_NONE_VALUE.
    gp.orderlyState = SU_NONE_VALUE;

    OK = OK && NV_SYNC_PERSISTENT(orderlyState);

    // This can be reset after the first completion of a TPM2_Startup() after
    // a power loss. It can probably be reset earlier but this is an OK place.
    if(OK)
        g_powerWasLost = FALSE;

    return (OK) ? TPM_RC_SUCCESS : TPM_RC_FAILURE;
}
#endif // CC_Startup
```

9.4 TPM2_Shutdown

9.4.1 General Description

This command is used to prepare the TPM for a power cycle. The *shutdownType* parameter indicates how the subsequent TPM2_Startup() will be processed.

For a *shutdownType* of any type, the volatile portion of Clock is saved to NV memory and the orderly shutdown indication is SET. NV Indexes with the TPMA_NV_ORDERLY attribute will be updated.

For a *shutdownType* of TPM_SU_STATE, the following additional items are saved:

- tracking information for saved session contexts;
- the session context counter;
- PCR that are designated as being preserved by TPM2_Shutdown(TPM_SU_STATE);
- the PCR Update Counter;
- flags associated with supporting the TPMA_NV_WRITESTCLEAR and TPMA_NV_READSTCLEAR attributes;
- the counter value and authPolicy for each ACT; and

  NOTE If a counter has not been updated since the last TPM2_Startup(), then the saved value will be one half of the current counter value.

- the command audit digest and count.

The following items shall not be saved and will not be in TPM memory after the next TPM2_Startup:

- TPM-memory-resident session contexts;
- TPM-memory-resident transient objects; or
- TPM-memory-resident hash contexts created by TPM2_HashSequenceStart().

Some values may be either derived from other values or saved to NV memory.

This command saves TPM state but does not change the state other than the internal indication that the context has been saved. The TPM shall continue to accept commands. If a subsequent command changes TPM state saved by this command, then the effect of this command is nullified. The TPM MAY nullify this command for any subsequent command rather than check whether the command changed state saved by this command. If this command is nullified. and if no TPM2_Shutdown() occurs before the next TPM2_Startup(), then the next TPM2_Startup() shall be TPM2_Startup(CLEAR).

9.4.2 Command and Response

Table 7 — TPM2_Shutdown Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_Shutdown {NV}
TPM_SU	shutdownType	TPM_SU_CLEAR or TPM_SU_STATE

Table 8 — TPM2_Shutdown Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

9.4.3 Detailed Actions

```
#include "Tpm.h"
#include "Shutdown_fp.h"
#if CC_Shutdown  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_TYPE	if PCR bank has been re-configured, a CLEAR StateSave() is required

```
TPM_RC
TPM2_Shutdown(
    Shutdown_In     *in             // IN: input parameter list
    )
{
    // The command needs NV update.  Check if NV is available.
    // A TPM_RC_NV_UNAVAILABLE or TPM_RC_NV_RATE error may be returned at
    // this point
    RETURN_IF_NV_IS_NOT_AVAILABLE;

// Input Validation

    // If PCR bank has been reconfigured, a CLEAR state save is required
    if(g_pcrReConfig && in->shutdownType == TPM_SU_STATE)
        return TPM_RCS_TYPE + RC_Shutdown_shutdownType;

// Internal Data Update

    gp.orderlyState = in->shutdownType;

    // PCR private date state save
    PCRStateSave(in->shutdownType);

    // Save the ACT state
    ActShutdown(in->shutdownType);

    // Save RAM backed NV index data
    NvUpdateIndexOrderlyData();

#if ACCUMULATE_SELF_HEAL_TIMER
    // Save the current time value
    go.time = g_time;
#endif

    // Save all orderly data
    NvWrite(NV_ORDERLY_DATA, sizeof(ORDERLY_DATA), &go);

    if(in->shutdownType == TPM_SU_STATE)
    {
        // Save STATE_RESET and STATE_CLEAR data
        NvWrite(NV_STATE_CLEAR_DATA, sizeof(STATE_CLEAR_DATA), &gc);
        NvWrite(NV_STATE_RESET_DATA, sizeof(STATE_RESET_DATA), &gr);

        // Save the startup flags for resume
        if(g_DrtmPreStartup)
            gp.orderlyState = TPM_SU_STATE | PRE_STARTUP_FLAG;
        else if(g_StartupLocality3)
            gp.orderlyState = TPM_SU_STATE | STARTUP_LOCALITY_3;
    }
    // only two shutdown options.
    else if(in->shutdownType != TPM_SU_CLEAR)
        return TPM_RCS_VALUE + RC_Shutdown_shutdownType;
```



```

    NV_SYNC_PERSISTENT(orderlyState);

    return TPM_RC_SUCCESS;
}
#endif // CC_Shutdown
```

10 Testing

10.1 Introduction

Compliance to standards for hardware security modules may require that the TPM test its functions before the results that depend on those functions may be returned. The TPM may perform operations using testable functions before those functions have been tested as long as the TPM returns no value that depends on the correctness of the testable function.

EXAMPLE TPM2_PCR_Extend() may be executed before the hash algorithms have been tested. However, until the hash algorithms have been tested, the contents of a PCR may not be used in any command if that command may result in a value being returned to the TPM user. This means that TPM2_PCR_Read() or TPM2_PolicyPCR() could not complete until the hashes have been checked but other TPM2_PCR_Extend() commands may be executed even though the operation uses previous PCR values.

If a command is received that requires return of a value that depends on untested functions, the TPM shall test the required functions before completing the command.

Once the TPM has received TPM2_SelfTest() and before completion of all tests, the TPM is required to return TPM_RC_TESTING for any command that uses a function that requires a test.

If a self-test fails at any time, the TPM will enter Failure mode. While in Failure mode, the TPM will return TPM_RC_FAILURE for any command other than TPM2_GetTestResult() and TPM2_GetCapability(). The TPM will remain in Failure mode until the next _TPM_Init.

10.2 TPM2_SelfTest

10.2.1 General Description

This command causes the TPM to perform a test of its capabilities. If the *fullTest* is YES, the TPM will test all functions. If *fullTest* = NO, the TPM will only test those functions that have not previously been tested.

If any tests are required, the TPM shall either

- return TPM_RC_TESTING and begin self-test of the required functions, or

  NOTE 1 If *fullTest* is NO, and all functions have been tested, the TPM shall return TPM_RC_SUCCESS.

- perform the tests and return the test result when complete. On failure, the TPM shall return TPM_RC_FAILURE.

If the TPM uses option a), the TPM shall return TPM_RC_TESTING for any command that requires use of a testable function, even if the functions required for completion of the command have already been tested.

NOTE 2 This command may cause the TPM to continue processing after it has returned the response. So that software can be notified of the completion of the testing, the interface may include controls that would allow the TPM to generate an interrupt when the "background" processing is complete. This would be in addition to the interrupt that may be available for signaling normal command completion. It is not necessary that there be two interrupts, but the interface should provide a way to indicate the nature of the interrupt (normal command or deferred command).

NOTE 3 The PC Client platform specific TPM, in response to *fullTest* YES, will not return TPM_RC_TESTING. It will block until all tests are complete.

10.2.2 Command and Response

Table 9 — TPM2_SelfTest Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_SelfTest {NV}
TPMI_YES_NO	fullTest	YES if full test to be performed NO if only test of untested functions required

Table 10 — TPM2_SelfTest Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

10.2.3 Detailed Actions

```
#include "Tpm.h"
#include "SelfTest_fp.h"
#if CC_SelfTest  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_CANCELED	the command was canceled (some incremental process may have been made)
TPM_RC_TESTING	self test in process

```
TPM_RC
TPM2_SelfTest(
    SelfTest_In     *in             // IN: input parameter list
    )
{
// Command Output

    // Call self test function in crypt module
    return CryptSelfTest(in->fullTest);
}
#endif // CC_SelfTest
```

10.3 TPM2_IncrementalSelfTest

10.3.1 General Description

This command causes the TPM to perform a test of the selected algorithms.

NOTE 1 The *toTest* list indicates the algorithms that software would like the TPM to test in anticipation of future use. This allows tests to be done so that a future commands will not be delayed due to testing.

The implementation may treat algorithms on the *toTest* list as either 'test each completely' or 'test this combination.'

EXAMPLE If the *toTest* list includes AES and CTR mode, it may be interpreted as a request to test only AES in CTR mode. Alternatively, it may be interpreted as a request to test AES in all modes and CTR mode for all symmetric algorithms.

If *toTest* contains an algorithm that has already been tested, it will not be tested again.

NOTE 2 The only way to force retesting of an algorithm is with TPM2_SelfTest(*fullTest* = YES).

The TPM will return in *toDoList* a list of algorithms that are yet to be tested. This list is not the list of algorithms that are scheduled to be tested but the algorithms/functions that have not been tested. Only the algorithms on the *toTest* list are scheduled to be tested by this command.

NOTE 3 An algorithm remains on the *toDoList* while any part of it remains untested.

EXAMPLE A symmetric algorithm remains untested until it is tested with all its modes.

Making *toTest* an empty list allows the determination of the algorithms that remain untested without triggering any testing.

If *toTest* is not an empty list, the TPM shall return TPM_RC_SUCCESS for this command and then return TPM_RC_TESTING for any subsequent command (including TPM2_IncrementalSelfTest()) until the requested testing is complete.

NOTE 4 If *toDoList* is empty, then no additional tests are required and TPM_RC_TESTING will not be returned in subsequent commands and no additional delay will occur in a command due to testing.

NOTE 5 If none of the algorithms listed in *toTest* is in the *toDoList*, then no tests will be performed.

NOTE 6 The TPM cannot return TPM_RC_TESTING for the first call to this command even when testing is not complete, because response parameters can only returned with the TPM_RC_SUCCESS return code.

If all the parameters in this command are valid, the TPM returns TPM_RC_SUCCESS and the *toDoList* (which may be empty).

NOTE 7 An implementation may perform all requested tests before returning TPM_RC_SUCCESS, or it may return TPM_RC_SUCCESS for this command and then return TPM_RC_TESTING for all subsequence commands (including TPM2_IncrementatSelfTest()) until the requested tests are complete.

10.3.2 Command and Response

Table 11 — TPM2_IncrementalSelfTest Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_IncrementalSelfTest {NV}
TPML_ALG	toTest	list of algorithms that should be tested

Table 12 — TPM2_IncrementalSelfTest Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPML_ALG	toDoList	list of algorithms that need testing

10.3.3 Detailed Actions

```
#include "Tpm.h"
#include "IncrementalSelfTest_fp.h"
#if CC_IncrementalSelfTest  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_CANCELED	the command was canceled (some tests may have completed)
TPM_RC_VALUE	an algorithm in the *toTest* list is not implemented

```
TPM_RC
TPM2_IncrementalSelfTest(
    IncrementalSelfTest_In      *in,            // IN: input parameter list
    IncrementalSelfTest_Out     *out            // OUT: output parameter list
    )
{
    TPM_RC                       result;
// Command Output

    // Call incremental self test function in crypt module. If this function
    // returns TPM_RC_VALUE, it means that an algorithm on the 'toTest' list is
    // not implemented.
    result = CryptIncrementalSelfTest(&in->toTest, &out->toDoList);
    if(result == TPM_RC_VALUE)
        return TPM_RCS_VALUE + RC_IncrementalSelfTest_toTest;
    return result;
}
#endif // CC_IncrementalSelfTest
```


10.4 TPM2_GetTestResult

10.4.1 General Description

This command returns manufacturer-specific information regarding the results of a self-test and an indication of the test status.

If TPM2_SelfTest() has not been executed and a testable function has not been tested, *testResult* will be TPM_RC_NEEDS_TEST. If TPM2_SelfTest() has been received and the tests are not complete, *testResult* will be TPM_RC_TESTING.

If testing of all functions is complete without functional failures, *testResult* will be TPM_RC_SUCCESS. If any test failed, *testResult* will be TPM_RC_FAILURE.

This command will operate when the TPM is in Failure mode so that software can determine the test status of the TPM and so that diagnostic information can be obtained for use in failure analysis. If the TPM is in Failure mode, then *tag* is required to be TPM_ST_NO_SESSIONS or the TPM shall return TPM_RC_FAILURE.

NOTE The reference implementation may return a 32-bit value *s_failFunction*. This simply gives a unique value to each of the possible places where a failure could occur. It is not intended to provide a pointer to the function. __func__ is a pointer to a character string but the failure mode code can only return 32-bit values. It is expected that the manufacturer can disambiguate this value if a customer's TPM goes into failure mode.

10.4.2 Command and Response

Table 13 — TPM2_GetTestResult Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_GetTestResult

Table 14 — TPM2_GetTestResult Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_MAX_BUFFER	outData	test result data contains manufacturer-specific information
TPM_RC	testResult	

10.4.3 Detailed Actions

```
#include "Tpm.h"
#include "GetTestResult_fp.h"
#if CC_GetTestResult  // Conditional expansion of this file
```

In the reference implementation, this function is only reachable if the TPM is not in failure mode meaning that all tests that have been run have completed successfully. There is not test data and the test result is TPM_RC_SUCCESS.

```
TPM_RC
TPM2_GetTestResult(
    GetTestResult_Out   *out            // OUT: output parameter list
    )
{
// Command Output

    // Call incremental self test function in crypt module
    out->testResult = CryptGetTestResult(&out->outData);

    return TPM_RC_SUCCESS;
}
#endif // CC_GetTestResult
```

11 Session Commands

11.1 TPM2_StartAuthSession

11.1.1 General Description

This command is used to start an authorization session using alternative methods of establishing the session key (*sessionKey*). The session key is then used to derive values used for authorization and for encrypting parameters.

This command allows injection of a secret into the TPM using either asymmetric or symmetric encryption. The type of *tpmKey* determines how the value in *encryptedSalt* is encrypted. The decrypted secret value is used to compute the *sessionKey*.

NOTE 1 If *tpmKey* Is TPM_RH_NULL, then *encryptedSalt* is required to be an Empty Buffer.

The label value of "SECRET" (see "Terms and Definitions" in TPM 2.0 Part 1) is used in the recovery of the secret value.

The TPM generates the *sessionKey* from the recovered secret value.

No authorization is required for *tpmKey* or *bind*.

NOTE 2 The justification for using *tpmKey* without providing authorization is that the result of using the key is not available to the caller, except indirectly through the *sessionKey*. This does not represent a point of attack on the value of the key. If the caller attempts to use the session without knowing the *sessionKey* value, it is an authorization failure that will trigger the dictionary attack logic.

The entity referenced with the *bind* parameter contributes an authorization value to the *sessionKey* generation process.

If both *tpmKey* and *bind* are TPM_RH_NULL, then *sessionKey* is set to the Empty Buffer. If *tpmKey* is not TPM_RH_NULL, then *encryptedSalt* is used in the computation of *sessionKey*. If *bind* is not TPM_RH_NULL, the *authValue* of *bind* is used in the *sessionKey* computation.

If *symmetric* specifies a block cipher, then TPM_ALG_CFB is the only allowed value for the *mode* field in the *symmetric* parameter (TPM_RC_MODE).

This command starts an authorization session and returns the session handle along with an initial *nonceTPM* in the response.

If the TPM does not have a free slot for an authorization session, it shall return TPM_RC_SESSION_HANDLES.

If the TPM implements a "gap" scheme for assigning *contextID* values, then the TPM shall return TPM_RC_CONTEXT_GAP if creating the session would prevent recycling of old saved contexts (See "Context Management" in TPM 2.0 Part 1).

If *tpmKey* is not TPM_ALG_NULL then *encryptedSalt* shall be a TPM2B_ENCRYPTED_SECRET of the proper type for *tpmKey*. The TPM shall return TPM_RC_HANDLE if the sensitive portion of *tpmKey* is not loaded. The TPM shall return TPM_RC_VALUE if:

a) *tpmKey* references an RSA key and
    1) the size of *encryptedSalt* is not the same as the size of the public modulus of *tpmKey*,
    2) *encryptedSalt* has a value that is greater than the public modulus of *tpmKey*,
    3) *encryptedSalt* is not a properly encoded OAEP value, or
    4) the decrypted *salt* value is larger than the size of the digest produced by the *nameAlg* of *tpmKey*; or

NOTE 3 The asymScheme of the key object is ignored in this case and TPM_ALG_OAEP is used, even if asymScheme is set to TPM_ALG_NULL.

b) *tpmKey* references an ECC key and *encryptedSalt*

1) does not contain a TPMS_ECC_POINT or

2) is not a point on the curve of *tpmKey*;

NOTE 4 When ECC is used, the point multiply process produces a value (Z) that is used in a KDF to produce the final secret value. The size of the secret value is an input parameter to the KDF and the result will be set to be the size of the digest produced by the *nameAlg* of *tpmKey*.

The TPM shall return TPM_RC_KEY if *tpmkey* does not reference an asymmetric key. The TPM shall return TPM_RC_VALUE if the scheme of the key is not TPM_ALG_OAEP or TPM_ALG_NULL. The TPM shall return TPM_RC_ATTRIBUTES if tpmKey does not have the *decrypt* attribute SET.

NOTE While TPM_RC_VALUE is preferred, TPM_RC_SCHEME is acceptable.

If *bind* references a transient object, then the TPM shall return TPM_RC_HANDLE if the sensitive portion of the object is not loaded.

For all session types, this command will cause initialization of the *sessionKey* and may establish binding between the session and an object (the *bind* object). If *sessionType* is TPM_SE_POLICY or TPM_SE_TRIAL, the additional session initialization is:

- set *policySession→policyDigest* to a Zero Digest (the digest size for *policySession→policyDigest* is the size of the digest produced by *authHash*);
- authorization may be given at any locality;
- authorization may apply to any command code;
- authorization may apply to any command parameters or handles;
- the authorization has no time limit;
- an authValue is not needed when the authorization is used;
- the session is not bound;
- the session is not an audit session; and
- the time at which the policy session was created is recorded.

Additionally, if *sessionType* is TPM_SE_TRIAL, the session will not be usable for authorization but can be used to compute the *authPolicy* for an object.

NOTE 5 Although this command changes the session allocation information in the TPM, it does not invalidate a saved context. That is, TPM2_Shutdown() is not required after this command in order to re-establish the orderly state of the TPM. This is because the created context will occupy an available slot in the TPM and sessions in the TPM do not survive any TPM2_Startup(). However, if a created session is context saved, the orderly state does change.

The TPM shall return TPM_RC_SIZE if *nonceCaller* is less than 16 octets or is greater than the size of the digest produced by *authHash*.

11.1.2 Command and Response

Table 15 — TPM2_StartAuthSession Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit, decrypt, or encrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_StartAuthSession
TPMI_DH_OBJECT+	tpmKey	handle of a loaded decrypt key used to encrypt *salt* may be TPM_RH_NULL Auth Index: None
TPMI_DH_ENTITY+	bind	entity providing the *authValue* may be TPM_RH_NULL Auth Index: None
TPM2B_NONCE	nonceCaller	initial *nonceCaller,* sets nonceTPM size for the session shall be at least 16 octets
TPM2B_ENCRYPTED_SECRET	encryptedSalt	value encrypted according to the type of *tpmKey* If *tpmKey* is TPM_RH_NULL, this shall be the Empty Buffer.
TPM_SE	sessionType	indicates the type of the session; simple HMAC or policy (including a trial policy)
TPMT_SYM_DEF+	symmetric	the algorithm and key size for parameter encryption may select TPM_ALG_NULL
TPMI_ALG_HASH	authHash	hash algorithm to use for the session Shall be a hash algorithm supported by the TPM and not TPM_ALG_NULL

Table 16 — TPM2_StartAuthSession Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPMI_SH_AUTH_SESSION	sessionHandle	handle for the newly created session
TPM2B_NONCE	nonceTPM	the initial nonce from the TPM, used in the computation of the *sessionKey*

11.1.3 Detailed Actions

```
#include "Tpm.h"
#include "StartAuthSession_fp.h"
#if CC_StartAuthSession  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_ATTRIBUTES	*tpmKey* does not reference a decrypt key
TPM_RC_CONTEXT_GAP	the difference between the most recently created active context and the oldest active context is at the limits of the TPM
TPM_RC_HANDLE	input decrypt key handle only has public portion loaded
TPM_RC_MODE	*symmetric* specifies a block cipher but the mode is not TPM_ALG_CFB.
TPM_RC_SESSION_HANDLES	no session handle is available
TPM_RC_SESSION_MEMORY	no more slots for loading a session
TPM_RC_SIZE	nonce less than 16 octets or greater than the size of the digest produced by *authHash*
TPM_RC_VALUE	secret size does not match decrypt key type; or the recovered secret is larger than the digest size of the *nameAlg* of *tpmKey*; or, for an RSA decrypt key, if *encryptedSecret* is greater than the public modulus of *tpmKey*.

```
TPM_RC
TPM2_StartAuthSession(
    StartAuthSession_In      *in,            // IN: input parameter buffer
    StartAuthSession_Out     *out            // OUT: output parameter buffer
    )
{
    TPM_RC                    result = TPM_RC_SUCCESS;
    OBJECT                   *tpmKey;                 // TPM key for decrypt salt
    TPM2B_DATA                salt;

// Input Validation

    // Check input nonce size.  IT should be at least 16 bytes but not larger
    // than the digest size of session hash.
    if(in->nonceCaller.t.size < 16
       || in->nonceCaller.t.size > CryptHashGetDigestSize(in->authHash))
        return TPM_RCS_SIZE + RC_StartAuthSession_nonceCaller;

    // If an decrypt key is passed in, check its validation
    if(in->tpmKey != TPM_RH_NULL)
    {
        // Get pointer to loaded decrypt key
        tpmKey = HandleToObject(in->tpmKey);

        // key must be asymmetric with its sensitive area loaded. Since this
        // command does not require authorization, the presence of the sensitive
        // area was not already checked as it is with most other commands that
        // use the sensitive are so check it here
        if(!CryptIsAsymAlgorithm(tpmKey->publicArea.type))
            return TPM_RCS_KEY + RC_StartAuthSession_tpmKey;
        // secret size cannot be 0
        if(in->encryptedSalt.t.size == 0)
            return TPM_RCS_VALUE + RC_StartAuthSession_encryptedSalt;
        // Decrypting salt requires accessing the private portion of a key.
        // Therefore, tmpKey can not be a key with only public portion loaded
```



```
        if(tpmKey->attributes.publicOnly)
            return TPM_RCS_HANDLE + RC_StartAuthSession_tpmKey;
        // HMAC session input handle check.
        // tpmKey should be a decryption key
        if(!IS_ATTRIBUTE(tpmKey->publicArea.objectAttributes, TPMA_OBJECT, decrypt))
            return TPM_RCS_ATTRIBUTES + RC_StartAuthSession_tpmKey;
        // Secret Decryption.  A TPM_RC_VALUE, TPM_RC_KEY or Unmarshal errors
        // may be returned at this point
        result = CryptSecretDecrypt(tpmKey, &in->nonceCaller, SECRET_KEY,
                                    &in->encryptedSalt, &salt);
        if(result != TPM_RC_SUCCESS)
            return TPM_RCS_VALUE + RC_StartAuthSession_encryptedSalt;
    }
    else
    {
        // secret size must be 0
        if(in->encryptedSalt.t.size != 0)
            return TPM_RCS_VALUE + RC_StartAuthSession_encryptedSalt;
        salt.t.size = 0;
    }
    switch(HandleGetType(in->bind))
    {
        case TPM_HT_TRANSIENT:
        {
            OBJECT      *object = HandleToObject(in->bind);
            // If the bind handle references a transient object, make sure that we
            // can get to the authorization value. Also, make sure that the object
            // has a proper Name (nameAlg != TPM_ALG_NULL). If it doesn't, then
            // it might be possible to bind to an object where the authValue is
            // known. This does not create a real issue in that, if you know the
            // authorization value, you can actually bind to the object. However,
            // there is a potential
            if(object->attributes.publicOnly == SET)
                return TPM_RCS_HANDLE + RC_StartAuthSession_bind;
            break;
        }
        case TPM_HT_NV_INDEX:
        // a PIN index can't be a bind object
        {
            NV_INDEX       *nvIndex = NvGetIndexInfo(in->bind, NULL);
            if(IsNvPinPassIndex(nvIndex->publicArea.attributes)
               || IsNvPinFailIndex(nvIndex->publicArea.attributes))
                return TPM_RCS_HANDLE + RC_StartAuthSession_bind;
            break;
        }
        default:
            break;
    }
    // If 'symmetric' is a symmetric block cipher (not TPM_ALG_NULL or TPM_ALG_XOR)
    // then the mode must be CFB.
    if(in->symmetric.algorithm != TPM_ALG_NULL
       && in->symmetric.algorithm != TPM_ALG_XOR
       && in->symmetric.mode.sym != TPM_ALG_CFB)
        return TPM_RCS_MODE + RC_StartAuthSession_symmetric;

// Internal Data Update and command output

    // Create internal session structure.  TPM_RC_CONTEXT_GAP, TPM_RC_NO_HANDLES
    // or TPM_RC_SESSION_MEMORY errors may be returned at this point.
    //
    // The detailed actions for creating the session context are not shown here
    // as the details are implementation dependent
    // SessionCreate sets the output handle and nonceTPM
    result = SessionCreate(in->sessionType, in->authHash, &in->nonceCaller,
                           &in->symmetric, in->bind, &salt, &out->sessionHandle,
                           &out->nonceTPM);
```

```
    return result;
}
#endif // CC_StartAuthSession
```

11.2 TPM2_PolicyRestart

11.2.1 General Description

This command allows a policy authorization session to be returned to its initial state. This command is used after the TPM returns TPM_RC_PCR_CHANGED. That response code indicates that a policy will fail because the PCR have changed after TPM2_PolicyPCR() was executed. Restarting the session allows the authorizations to be replayed because the session restarts with the same *nonceTPM*. If the PCR are valid for the policy, the policy may then succeed.

This command does not reset the policy ID or the policy start time.

11.2.2 Command and Response

Table 17 — TPM2_PolicyRestart Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PolicyRestart
TPMI_SH_POLICY	sessionHandle	the handle for the policy session

Table 18 — TPM2_PolicyRestart Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

11.2.3 Detailed Actions

```
#include "Tpm.h"
#include "PolicyRestart_fp.h"
#if CC_PolicyRestart  // Conditional expansion of this file
TPM_RC
TPM2_PolicyRestart(
    PolicyRestart_In    *in             // IN: input parameter list
    )
{
    // Initialize policy session data
    SessionResetPolicyData(SessionGet(in->sessionHandle));

    return TPM_RC_SUCCESS;
}
#endif // CC_PolicyRestart
```

12 Object Commands

12.1 TPM2_Create

12.1.1 General Description

This command is used to create an object that can be loaded into a TPM using TPM2_Load(). If the command completes successfully, the TPM will create the new object and return the object's creation data (*creationData*), its public area (*outPublic*), and its encrypted sensitive area (*outPrivate*). Preservation of the returned data is the responsibility of the caller. The object will need to be loaded (TPM2_Load()) before it may be used. The only difference between the *inPublic* TPMT_PUBLIC template and the *outPublic* TPMT_PUBLIC object is in the *unique* field.

NOTE 1 This command may require temporary use of a transient resource, even though the object does not remain loaded after the command. See Part 1 Transient Resources.

TPM2B_PUBLIC template (*inPublic*) contains all of the fields necessary to define the properties of the new object. The setting for these fields is defined in "Public Area Template" in Part 1 of this specification and in "TPMA_OBJECT" in Part 2 of this specification. The size of the *unique* field shall not be checked for consistency with the other object parameters.

NOTE 2 For interoperability, the *unique* field should not be set to a value that is larger than allowed by object parameters, so that the unmarshaling will not fail. A size of zero is recommended. After unmarshaling, the TPM does not use the input *unique* field. It is, however, used in TPM2_CreatePrimary() and TPM2_CreateLoaded.

EXAMPLE 1 A TPM_ALG_RSA object with a *keyBits* of 2048 in the object's parameters should have a *unique* field that is no larger than 256 bytes.

EXAMPLE 2 TPM_ALG_KEYEDHASH or a TPM_ALG_SYMCIPHER object should have a *unique* field that is no larger than the digest produced by the object's *nameAlg*.

The *parentHandle* parameter shall reference a loaded decryption key that has both the public and sensitive area loaded.

When defining the object, the caller provides a template structure for the object in a TPM2B_PUBLIC structure (*inPublic*), an initial value for the object's *authValue (inSensitive.userAuth),* and, if the object is a symmetric object, an optional initial data value (*inSensitive.data*). The TPM shall validate the consistency of the attributes of *inPublic* according to the Creation rules in "TPMA_OBJECT" in TPM 2.0 Part 2.

The *inSensitive* parameter may be encrypted using parameter encryption.

The methods in this clause are used by both TPM2_Create() and TPM2_CreatePrimary(). When a value is indicated as being TPM-generated, the value is filled in by bits from the RNG if the command is TPM2_Create() and with values from **KDFa**() if the command is TPM2_CreatePrimary(). The parameters of each creation value are specified in TPM 2.0 Part 1.

The *sensitiveDataOrigin* attribute of *inPublic* shall be SET if *inSensitive.data* is an Empty Buffer and CLEAR if *inSensitive.data* is not an Empty Buffer or the TPM shall return TPM_RC_ATTRIBUTES.

If the Object is a not a *keyedHash* object, and the *sign* and *encrypt* attributes are CLEAR, the TPM shall return TPM_RC_ATTRIBUTES.

The TPM will create new data for the sensitive area and compute a TPMT_PUBLIC.*unique* from the sensitive area based on the object type:

a) For a symmetric key:

   1) If *inSensitive*.sensitive.*data* is the Empty Buffer, a TPM-generated key value is placed in the new object's TPMT_SENSITIVE.*sensitive.sym*. The size of the key will be determined by *inPublic.publicArea.parameters*.

2) If *inSensitive*.sensitive.*data* is not the Empty Buffer, the TPM will validate that the size of *inSensitive.data* is no larger than the key size indicated in the *inPublic template* (TPM_RC_SIZE) and copy the *inSensitive.data* to TPMT_SENSITIVE.*sensitive*.*sym* of the new object.
3) A TPM-generated obfuscation value is placed in TPMT_SENSITIVE.*sensitive.seedValue*. The size of the obfuscation value is the size of the digest produced by the nameAlg in *inPublic*. This value prevents the public *unique* value from leaking information about the *sensitive* area.
4) The TPMT_PUBLIC.*unique.sym* value for the new object is then generated, as shown in equation (1) below, by hashing the key and obfuscation values in the TPMT_SENSITIVE with the *nameAlg* of the object.

$$unique \coloneqq \mathbf{H}_{nameAlg}(sensitive.seedValue.buffer \mid\mid sensitive.any.buffer) \tag{1}$$

b) If the Object is an asymmetric key:

1) If *inSensitive.sensitive.data* is not the Empty Buffer, then the TPM shall return TPM_RC_VALUE.
2) A TPM-generated private key value is created with the size determined by the parameters of inPublic.publicArea.parameters.
3) If the key is a Storage Key, a TPM-generated TPMT_SENSITIVE.*seedValue* value is created; otherwise, TPMT_SENSITIVE.*seedValue.size* is set to zero.

NOTE 3 An Object that is not a storage key has no child Objects to encrypt, so it does not need a symmetric key.

4) The public *unique* value is computed from the private key according to the methods of the key type.
5) If the key is an ECC key and the scheme required by the curveID is not the same as *scheme* in the public area of the template, then the TPM shall return TPM_RC_SCHEME.
6) If the key is an ECC key and the KDF required by the curveID is not the same as *kdf* in the pubic area of the template, then the TPM shall return TPM_RC_KDF.

NOTE 4 There is currently no command in which the caller may specify the KDF to be used with an ECC decryption key. Since there is no use for this capability, the reference implementation requires that the *kdf* in the template be set to TPM_ALG_NULL or TPM_RC_KDF is returned.

c) If the Object is a *keyedHash* object:

1) If *inSensitive.sensitive.data* is an Empty Buffer, and both *sign* and *decrypt* are CLEAR in the attributes of *inPublic*, the TPM shall return TPM_RC_ATTRIBUTES. This would be a data object with no data.

NOTE 5 Revisions 134 and earlier reference code did not check the error case of *sensitiveDataOrigin* SET and an Empty Buffer. Thus, some TPM implementations may also not have included this error check.

2) If *sign* and *decrypt* are both CLEAR, or if *sign* and *decrypt* are both SET and the *scheme* in the public area of the template is not TPM_ALG_NULL, the TPM shall return TPM_RC_SCHEME.

NOTE 6 Revisions 138 and earlier did not enforce this error case.

3) If *inSensitive.sensitive.data* is not an Empty Buffer, the TPM will copy the *inSensitive.sensitive.data* to TPMT_SENSITIVE.*sensitive.bits* of the new object.

NOTE 7 The size of inSensitive.sensitive.data is limited to be no larger than MAX_SYM_DATA.

4) If *inSensitive.sensitive.data* is an Empty Buffer, a TPM-generated key value that is the size of the digest produced by the *nameAlg* in *inPublic* is placed in TPMT_SENSITIVE.*sensitive.bits*.

5) A TPM-generated obfuscation value that is the size of the digest produced by the *nameAlg* of *inPublic* is placed in TPMT_SENSITIVE.*seedValue*.

6) The TPMT_PUBLIC.*unique.keyedHash* value for the new object is then generated, as shown in equation (1) above, by hashing the key and obfuscation values in the TPMT_SENSITIVE with the *nameAlg* of the object.

For TPM2_Load(), the TPM will apply normal symmetric protections to the created TPMT_SENSITIVE to create *outPublic*.

NOTE 8 The encryption key is derived from the symmetric seed in the sensitive area of the parent.

In addition to *outPublic* and *outPrivate*, the TPM will build a TPMS_CREATION_DATA structure for the object. TPMS_CREATION_DATA.*outsideInfo* is set to *outsideInfo*. This structure is returned in *creationData*. Additionally, the digest of this structure is returned in *creationHash*, and, finally, a TPMT_TK_CREATION is created so that the association between the creation data and the object may be validated by TPM2_CertifyCreation().

If the object being created is a Storage Key and *fixedParent* is SET in the attributes of *inPublic*, then the symmetric algorithms and parameters of *inPublic* are required to match those of the parent. The algorithms that must match are *inPublic.nameAlg*, and the values in *inPublic.parameters* that select the symmetric scheme. If *inPublic.nameAlg* does not match, the TPM shall return TPM_RC_HASH.If the symmetric scheme of the key does not match, the parent, the TPM shall return TPM_RC_SYMMETRIC. The TPM shall not use different response code to differentiate between mismatches of the components of *inPublic.parameters*. However, after this verification, when using the scheme to encrypt child objects, the TPM ignores the symmetric mode and uses TPM_ALG_CFB.

NOTE 9 The symmetric scheme is a TPMT_SYM_DEF_OBJECT. In a symmetric block ciphier, it is at *inPublic.parameters.symDetail.sym* and in an asymmetric object is at *inPublic.parameters.asymDetail.symmetric.*

NOTE 10 Prior to revision 01.34, the parent asymmetric algorithms were also checked for *fixedParent* storage keys.

12.1.2 Command and Response

Table 19 — TPM2_Create Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_Create
TPMI_DH_OBJECT	@parentHandle	handle of parent for new object Auth Index: 1 Auth Role: USER
TPM2B_SENSITIVE_CREATE	inSensitive	the sensitive data
TPM2B_PUBLIC	inPublic	the public template
TPM2B_DATA	outsideInfo	data that will be included in the creation data for this object to provide permanent, verifiable linkage between this object and some object owner data
TPML_PCR_SELECTION	creationPCR	PCR that will be used in creation data

Table 20 — TPM2_Create Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_PRIVATE	outPrivate	the private portion of the object
TPM2B_PUBLIC	outPublic	the public portion of the created object
TPM2B_CREATION_DATA	creationData	contains a TPMS_CREATION_DATA
TPM2B_DIGEST	creationHash	digest of *creationData* using *nameAlg* of *outPublic*
TPMT_TK_CREATION	creationTicket	ticket used by TPM2_CertifyCreation() to validate that the creation data was produced by the TPM

12.1.3 Detailed Actions

```
#include "Tpm.h"
#include "Object_spt_fp.h"
#include "Create_fp.h"
#if CC_Create  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_ATTRIBUTES	*sensitiveDataOrigin* is CLEAR when *sensitive.data* is an Empty Buffer, or is SET when *sensitive.data* is not empty; *fixedTPM*, *fixedParent*, or *encryptedDuplication* attributes are inconsistent between themselves or with those of the parent object; inconsistent *restricted*, *decrypt* and *sign* attributes; attempt to inject sensitive data for an asymmetric key;
TPM_RC_HASH	non-duplicable storage key and its parent have different name algorithm
TPM_RC_KDF	incorrect KDF specified for decrypting keyed hash object
TPM_RC_KEY	invalid key size values in an asymmetric key public area or a provided symmetric key has a value that is not allowed
TPM_RC_KEY_SIZE	key size in public area for symmetric key differs from the size in the sensitive creation area; may also be returned if the TPM does not allow the key size to be used for a Storage Key
TPM_RC_OBJECT_MEMORY	a free slot is not available as scratch memory for object creation
TPM_RC_RANGE	the exponent value of an RSA key is not supported.
TPM_RC_SCHEME	inconsistent attributes *decrypt*, *sign*, or *restricted* and key's scheme ID; or hash algorithm is inconsistent with the scheme ID for keyed hash object
TPM_RC_SIZE	size of public *authPolicy* or sensitive *authValue* does not match digest size of the name algorithm sensitive data size for the keyed hash object is larger than is allowed for the scheme
TPM_RC_SYMMETRIC	a storage key with no symmetric algorithm specified; or non-storage key with symmetric algorithm different from TPM_ALG_NULL
TPM_RC_TYPE	unknown object type; *parentHandle* does not reference a restricted decryption key in the storage hierarchy with both public and sensitive portion loaded
TPM_RC_VALUE	exponent is not prime or could not find a prime using the provided parameters for an RSA key; unsupported name algorithm for an ECC key
TPM_RC_OBJECT_MEMORY	there is no free slot for the object

```
TPM_RC
TPM2_Create(
    Create_In       *in,            // IN: input parameter list
    Create_Out      *out            // OUT: output parameter list
    )
{
    TPM_RC                   result = TPM_RC_SUCCESS;
    OBJECT                  *parentObject;
    OBJECT                  *newObject;
    TPMT_PUBLIC             *publicArea;

// Input Validation
    parentObject = HandleToObject(in->parentHandle);
```



```
    pAssert(parentObject != NULL);

    // Does parent have the proper attributes?
    if(!ObjectIsParent(parentObject))
        return TPM_RCS_TYPE + RC_Create_parentHandle;

    // Get a slot for the creation
    newObject = FindEmptyObjectSlot(NULL);
    if(newObject == NULL)
        return TPM_RC_OBJECT_MEMORY;
    // If the TPM2B_PUBLIC was passed as a structure, marshal it into is canonical
    // form for processing

    // to save typing.
    publicArea = &newObject->publicArea;

    // Copy the input structure to the allocated structure
    *publicArea = in->inPublic.publicArea;

    // Check attributes in input public area. CreateChecks() checks the things that
    // are unique to creation and then validates the attributes and values that are
    // common to create and load.
    result = CreateChecks(parentObject, publicArea,
                          in->inSensitive.sensitive.data.t.size);
    if(result != TPM_RC_SUCCESS)
        return RcSafeAddToResult(result, RC_Create_inPublic);
    // Clean up the authValue if necessary
    if(!AdjustAuthSize(&in->inSensitive.sensitive.userAuth, publicArea->nameAlg))
        return TPM_RCS_SIZE + RC_Create_inSensitive;

// Command Output
    // Create the object using the default TPM random-number generator
    result = CryptCreateObject(newObject, &in->inSensitive.sensitive, NULL);
    if(result != TPM_RC_SUCCESS)
        return result;
    // Fill in creation data
    FillInCreationData(in->parentHandle, publicArea->nameAlg,
                       &in->creationPCR, &in->outsideInfo,
                       &out->creationData, &out->creationHash);

    // Compute creation ticket
    TicketComputeCreation(EntityGetHierarchy(in->parentHandle), &newObject->name,
                          &out->creationHash, &out->creationTicket);

    // Prepare output private data from sensitive
    SensitiveToPrivate(&newObject->sensitive, &newObject->name, parentObject,
                       publicArea->nameAlg,
                       &out->outPrivate);

    // Finish by copying the remaining return values
    out->outPublic.publicArea = newObject->publicArea;

    return TPM_RC_SUCCESS;
}
#endif // CC_Create
```

12.2 TPM2_Load

12.2.1 General Description

This command is used to load objects into the TPM. This command is used when both a TPM2B_PUBLIC and TPM2B_PRIVATE are to be loaded. If only a TPM2B_PUBLIC is to be loaded, the TPM2_LoadExternal command is used.

NOTE 1 Loading an object is not the same as restoring a saved object context.

The object's TPMA_OBJECT attributes will be checked according to the rules defined in "TPMA_OBJECT" in TPM 2.0 Part 2 of this specification. If the Object is a not a *keyedHash* object, and the *sign* and *encrypt* attributes are CLEAR, the TPM shall return TPM_RC_ATTRIBUTES.

Objects loaded using this command will have a Name. The Name is the concatenation of *nameAlg* and the digest of the public area using the *nameAlg.*

NOTE 2 *nameAlg* is a parameter in the public area of the inPublic structure.

If *inPrivate.size* is zero, the load will fail.

After *inPrivate.buffer* is decrypted using the symmetric key of the parent, the integrity value shall be checked before the sensitive area is used, or unmarshaled.

NOTE 3 Checking the integrity before the data is used prevents attacks on the sensitive area by fuzzing the data and looking at the differences in the response codes.

The command returns a handle for the loaded object and the Name that the TPM computed for *inPublic.public* (that is, the digest of the TPMT_PUBLIC structure in *inPublic*).

NOTE 4 The TPM-computed Name is provided as a convenience to the caller for those cases where the caller does not implement the hash algorithms specified in the *nameAlg* of the object.

NOTE 5 The returned handle is associated with the object until the object is flushed (TPM2_FlushContext) or until the next TPM2_Startup.

For all objects, the size of the key in the sensitive area shall be consistent with the key size indicated in the public area or the TPM shall return TPM_RC_KEY_SIZE.

Before use, a loaded object shall be checked to validate that the public and sensitive portions are properly linked, cryptographically. Use of an object includes use in any policy command. If the parts of the object are not properly linked, the TPM shall return TPM_RC_BINDING. If a weak symmetric key is in the sensitive portion, the TPM shall return TPM_RC_KEY.

EXAMPLE 1 For a symmetric object, the unique value in the public area shall be the digest of the sensitive key and the obfuscation value.

EXAMPLE 2 For a two-prime RSA key, the remainder when dividing the public modulus by the private key shall be zero and it shall be possible to form a private exponent from the two prime factors of the public modulus.

EXAMPLE 3 For an ECC key, the public point shall be f(x) where x is the private key.

12.2.2 Command and Response

Table 21 — TPM2_Load Command

Type	**Name**	**Description**
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_Load
TPMI_DH_OBJECT	@parentHandle	TPM handle of parent key; shall not be a reserved handle Auth Index: 1 Auth Role: USER
TPM2B_PRIVATE	inPrivate	the private portion of the object
TPM2B_PUBLIC	inPublic	the public portion of the object

Table 22 — TPM2_Load Response

Type	**Name**	**Description**
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM_HANDLE	objectHandle	handle of type TPM_HT_TRANSIENT for the loaded object
TPM2B_NAME	name	Name of the loaded object

12.2.3 Detailed Actions

```
#include "Tpm.h"
#include "Load_fp.h"
#if CC_Load  // Conditional expansion of this file
#include "Object_spt_fp.h"
```

Error Returns	Meaning
TPM_RC_ATTRIBUTES	*inPulblic* attributes are not allowed with selected parent
TPM_RC_BINDING	*inPrivate* and *inPublic* are not cryptographically bound
TPM_RC_HASH	incorrect hash selection for signing key or the *nameAlg* for '*inPubic* is not valid
TPM_RC_INTEGRITY	HMAC on *inPrivate* was not valid
TPM_RC_KDF	KDF selection not allowed
TPM_RC_KEY	the size of the object's *unique* field is not consistent with the indicated size in the object's parameters
TPM_RC_OBJECT_MEMORY	no available object slot
TPM_RC_SCHEME	the signing scheme is not valid for the key
TPM_RC_SENSITIVE	the *inPrivate* did not unmarshal correctly
TPM_RC_SIZE	*inPrivate* missing, or *authPolicy* size for *inPublic* or is not valid
TPM_RC_SYMMETRIC	symmetric algorithm not provided when required
TPM_RC_TYPE	*parentHandle* is not a storage key, or the object to load is a storage key but its parameters do not match the parameters of the parent.
TPM_RC_VALUE	decryption failure

```
TPM_RC
TPM2_Load(
    Load_In         *in,            // IN: input parameter list
    Load_Out        *out            // OUT: output parameter list
    )
{
    TPM_RC                      result = TPM_RC_SUCCESS;
    TPMT_SENSITIVE              sensitive;
    OBJECT                     *parentObject;
    OBJECT                     *newObject;

// Input Validation
    // Don't get invested in loading if there is no place to put it.
    newObject = FindEmptyObjectSlot(&out->objectHandle);
    if(newObject == NULL)
        return TPM_RC_OBJECT_MEMORY;

    if(in->inPrivate.t.size == 0)
        return TPM_RCS_SIZE + RC_Load_inPrivate;

    parentObject = HandleToObject(in->parentHandle);
    pAssert(parentObject != NULL);
    // Is the object that is being used as the parent actually a parent.
    if(!ObjectIsParent(parentObject))
        return TPM_RCS_TYPE + RC_Load_parentHandle;

    // Compute the name of object. If there isn't one, it is because the nameAlg is
    // not valid.
```



```
    PublicMarshalAndComputeName(&in->inPublic.publicArea, &out->name);
    if(out->name.t.size == 0)
        return TPM_RCS_HASH + RC_Load_inPublic;

    // Retrieve sensitive data.
    result = PrivateToSensitive(&in->inPrivate.b, &out->name.b, parentObject,
                                in->inPublic.publicArea.nameAlg,
                                &sensitive);
    if(result != TPM_RC_SUCCESS)
        return RcSafeAddToResult(result, RC_Load_inPrivate);

// Internal Data Update
    // Load and validate object
    result = ObjectLoad(newObject, parentObject,
                        &in->inPublic.publicArea, &sensitive,
                        RC_Load_inPublic, RC_Load_inPrivate,
                        &out->name);
    if(result == TPM_RC_SUCCESS)
    {
        // Set the common OBJECT attributes for a loaded object.
        ObjectSetLoadedAttributes(newObject, in->parentHandle);
    }
    return result;

}
#endif // CC_Load
```


12.3 TPM2_LoadExternal

12.3.1 General Description

This command is used to load an object that is not a Protected Object into the TPM. The command allows loading of a public area or both a public and sensitive area.

NOTE 1 Typical use for loading a public area is to allow the TPM to validate an asymmetric signature. Typical use for loading both a public and sensitive area is to allow the TPM to be used as a crypto accelerator.

Load of a public external object area allows the object to be associated with a hierarchy so that the correct algorithms may be used when creating tickets. The *hierarchy* parameter provides this association. If the public and sensitive portions of the object are loaded, *hierarchy* is required to be TPM_RH_NULL.

NOTE 2 If both the public and private portions of an object are loaded, the object is not allowed to appear to be part of a hierarchy.

The object's TPMA_OBJECT attributes will be checked according to the rules defined in "TPMA_OBJECT" in TPM 2.0 Part 2. In particular, *fixedTPM*, *fixedParent*, and *restricted* shall be CLEAR if *inPrivate* is not the Empty Buffer.

NOTE 3 The duplication status of a public key needs to be able to be the same as the full key which may be resident on a different TPM. If both the public and private parts of the key are loaded, then it is not possible for the key to be either *fixedTPM* or *fixedParent*, since, its private area would not be available in the clear to load.

Objects loaded using this command will have a Name. The Name is the *nameAlg* of the object concatenated with the digest of the public area using the *nameAlg.* The Qualified Name for the object will be the same as its Name. The TPM will validate that the *authPolicy* is either the size of the digest produced by *nameAlg* or the Empty Buffer.

NOTE 4 If *nameAlg* is TPM_ALG_NULL, then the Name is the Empty Buffer. When the authorization value for an object with no Name is computed, no Name value is included in the HMAC. To ensure that these unnamed entities are not substituted, they should have an *authValue* that is statistically unique.

NOTE 5 The digest size for TPM_ALG_NULL is zero.

If the *nameAlg* is TPM_ALG_NULL, the TPM shall not verify the cryptographic binding between the public and sensitive areas, but the TPM will validate that the size of the key in the sensitive area is consistent with the size indicated in the public area. If it is not, the TPM shall return TPM_RC_KEY_SIZE.

NOTE 6 For an ECC object, the TPM will verify that the public key is on the curve of the key before the public area is used.

If *nameAlg* is not TPM_ALG_NULL, then the same consistency checks between *inPublic* and *inPrivate* are made as for TPM2_Load().

NOTE 7 Consistency checks are necessary because an object with a Name needs to have the public and sensitive portions cryptographically bound so that an attacker cannot mix pubic and sensitive areas.

The command returns a handle for the loaded object and the Name that the TPM computed for *inPublic.public* (that is, the TPMT_PUBLIC structure in *inPublic*).

NOTE 8 The TPM-computed Name is provided as a convenience to the caller for those cases where the caller does not implement the hash algorithm specified in the *nameAlg* of the object.

The *hierarchy* parameter associates the external object with a hierarchy. External objects are flushed when their associated hierarchy is disabled. If *hierarchy* is TPM_RH_NULL, the object is part of no hierarchy, and there is no implicit flush.

If *hierarchy* is TPM_RH_NULL or *nameAlg* is TPM_ALG_NULL, a ticket produced using the object shall be a NULL Ticket.

EXAMPLE If a key is loaded with hierarchy set to TPM_RH_NULL, then TPM2_VerifySignature() will produce a NULL Ticket of the required type.

External objects are Temporary Objects. The saved external object contexts shall be invalidated at the next TPM Reset.

If a weak symmetric key is in the sensitive area, the TPM shall return TPM_RC_KEY.

For an RSA key, the private exponent is computed using the two prime factors of the public modulus. One of the primes is P, and the second prime (Q) is found by dividing the public modulus by P. A TPM may return an error (TPM_RC_BINDING) if the bit size of P and Q are not the same.”

12.3.2 Command and Response

Table 23 — TPM2_LoadExternal Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit, encrypt, or decrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_LoadExternal
TPM2B_SENSITIVE	inPrivate	the sensitive portion of the object (optional)
TPM2B_PUBLIC+	inPublic	the public portion of the object
TPMI_RH_HIERARCHY+	hierarchy	hierarchy with which the object area is associated

Table 24 — TPM2_LoadExternal Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM_HANDLE	objectHandle	handle of type TPM_HT_TRANSIENT for the loaded object
TPM2B_NAME	name	name of the loaded object

12.3.3 Detailed Actions

```
#include "Tpm.h"
#include "LoadExternal_fp.h"
#if CC_LoadExternal  // Conditional expansion of this file
#include "Object_spt_fp.h"
```

Error Returns	Meaning
TPM_RC_ATTRIBUTES	'*fixedParent*", *fixedTPM*, and *restricted* must be CLEAR if sensitive portion of an object is loaded
TPM_RC_BINDING	the *inPublic* and *inPrivate* structures are not cryptographically bound
TPM_RC_HASH	incorrect hash selection for signing key
TPM_RC_HIERARCHY	*hierarchy* is turned off, or only NULL hierarchy is allowed when loading public and private parts of an object
TPM_RC_KDF	incorrect KDF selection for decrypting *keyedHash* object
TPM_RC_KEY	the size of the object's *unique* field is not consistent with the indicated size in the object's parameters
TPM_RC_OBJECT_MEMORY	if there is no free slot for an object
TPM_RC_ECC_POINT	for a public-only ECC key, the ECC point is not on the curve
TPM_RC_SCHEME	the signing scheme is not valid for the key
TPM_RC_SIZE	*authPolicy* is not zero and is not the size of a digest produced by the object's *nameAlg* TPM_RH_NULL hierarchy
TPM_RC_SYMMETRIC	symmetric algorithm not provided when required
TPM_RC_TYPE	*inPublic* and *inPrivate* are not the same type

```
TPM_RC
TPM2_LoadExternal(
    LoadExternal_In     *in,            // IN: input parameter list
    LoadExternal_Out    *out            // OUT: output parameter list
    )
{
    TPM_RC               result;
    OBJECT              *object;
    TPMT_SENSITIVE      *sensitive = NULL;

// Input Validation
    // Don't get invested in loading if there is no place to put it.
    object = FindEmptyObjectSlot(&out->objectHandle);
    if(object == NULL)
        return TPM_RC_OBJECT_MEMORY;

    // If the hierarchy to be associated with this object is turned off, the object
    // cannot be loaded.
    if(!HierarchyIsEnabled(in->hierarchy))
        return TPM_RCS_HIERARCHY + RC_LoadExternal_hierarchy;

    // For loading an object with both public and sensitive
    if(in->inPrivate.size != 0)
    {
        // An external object with a sensitive area can only be loaded in the
        // NULL hierarchy
        if(in->hierarchy != TPM_RH_NULL)
            return TPM_RCS_HIERARCHY + RC_LoadExternal_hierarchy;
        // An external object with a sensitive area must have fixedTPM == CLEAR
```



```
        // fixedParent == CLEAR so that it does not appear to be a key created by
        // this TPM.
        if(IS_ATTRIBUTE(in->inPublic.publicArea.objectAttributes, TPMA_OBJECT,
                        fixedTPM)
           || IS_ATTRIBUTE(in->inPublic.publicArea.objectAttributes, TPMA_OBJECT,
                           fixedParent)
           || IS_ATTRIBUTE(in->inPublic.publicArea.objectAttributes, TPMA_OBJECT,
                           restricted))
            return TPM_RCS_ATTRIBUTES + RC_LoadExternal_inPublic;

        // Have sensitive point to something other than NULL so that object
        // initialization will load the sensitive part too
        sensitive = &in->inPrivate.sensitiveArea;
    }

    // Need the name to initialize the object structure
    PublicMarshalAndComputeName(&in->inPublic.publicArea, &out->name);

    // Load and validate key
    result = ObjectLoad(object, NULL,
                        &in->inPublic.publicArea, sensitive,
                        RC_LoadExternal_inPublic, RC_LoadExternal_inPrivate,
                        &out->name);
    if(result == TPM_RC_SUCCESS)
    {
        object->attributes.external = SET;
        // Set the common OBJECT attributes for a loaded object.
        ObjectSetLoadedAttributes(object, in->hierarchy);
    }
    return result;
}
#endif // CC_LoadExternal
```

12.4 TPM2_ReadPublic

12.4.1 General Description

This command allows access to the public area of a loaded object.

Use of the *objectHandle* does not require authorization.

NOTE Since the caller is not likely to know the public area of the object associated with *objectHandle*, it would not be possible to include the Name associated with *objectHandle* in the *cpHash* computation.

If *objectHandle* references a sequence object, the TPM shall return TPM_RC_SEQUENCE.

12.4.2 Command and Response

Table 25 — TPM2_ReadPublic Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit or encrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_ReadPublic
TPMI_DH_OBJECT	objectHandle	TPM handle of an object Auth Index: None

Table 26 — TPM2_ReadPublic Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_PUBLIC	outPublic	structure containing the public area of an object
TPM2B_NAME	name	name of the object
TPM2B_NAME	qualifiedName	the Qualified Name of the object

12.4.3 Detailed Actions

```
#include "Tpm.h"
#include "ReadPublic_fp.h"
#if CC_ReadPublic  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_SEQUENCE	can not read the public area of a sequence object

```
TPM_RC
TPM2_ReadPublic(
    ReadPublic_In   *in,            // IN: input parameter list
    ReadPublic_Out  *out            // OUT: output parameter list
    )
{
    OBJECT                  *object = HandleToObject(in->objectHandle);

// Input Validation
    // Can not read public area of a sequence object
    if(ObjectIsSequence(object))
        return TPM_RC_SEQUENCE;

// Command Output
    out->outPublic.publicArea = object->publicArea;
    out->name = object->name;
    out->qualifiedName = object->qualifiedName;

    return TPM_RC_SUCCESS;
}
#endif // CC_ReadPublic
```

12.5 TPM2_ActivateCredential

12.5.1 General Description

This command enables the association of a credential with an object in a way that ensures that the TPM has validated the parameters of the credentialed object.

If both the public and private portions of *activateHandle* and *keyHandle* are not loaded, then the TPM shall return TPM_RC_AUTH_UNAVAILABLE.

If *keyHandle* is not a Storage Key, then the TPM shall return TPM_RC_TYPE.

Authorization for *activateHandle* requires the ADMIN role.

The key associated with *keyHandle* is used to recover a seed from secret, which is the encrypted seed. The Name of the object associated with activateHandle and the recovered seed are used in a KDF to recover the symmetric key. The recovered seed (but not the Name) is used in a KDF to recover the HMAC key.

The HMAC is used to validate that the *credentialBlob* is associated with *activateHandle* and that the data in *credentialBlob* has not been modified. The linkage to the object associated with activateHandle is achieved by including the Name in the HMAC calculation.

If the integrity checks succeed, *credentialBlob* is decrypted and returned as *certInfo*.

NOTE The output *certInfo* parameter is an application defined value. It is typically a symmetric key or seed that is used to decrypt a certificate. See the TPM2_MakeCredential *credential* input parameter.

12.5.2 Command and Response

Table 27 — TPM2_ActivateCredential Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_ActivateCredential
TPMI_DH_OBJECT	@activateHandle	handle of the object associated with certificate in *credentialBlob* Auth Index: 1 Auth Role: ADMIN
TPMI_DH_OBJECT	@keyHandle	loaded key used to decrypt the TPMS_SENSITIVE in *credentialBlob* Auth Index: 2 Auth Role: USER
TPM2B_ID_OBJECT	credentialBlob	the credential
TPM2B_ENCRYPTED_SECRET	secret	*keyHandle* algorithm-dependent encrypted seed that protects *credentialBlob*

Table 28 — TPM2_ActivateCredential Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_DIGEST	certInfo	the decrypted certificate information the data should be no larger than the size of the digest of the *nameAlg* associated with *keyHandle*

12.5.3 Detailed Actions

```
#include "Tpm.h"
#include "ActivateCredential_fp.h"
#if CC_ActivateCredential  // Conditional expansion of this file
#include "Object_spt_fp.h"
```

Error Returns	Meaning
TPM_RC_ATTRIBUTES	*keyHandle* does not reference a decryption key
TPM_RC_ECC_POINT	*secret* is invalid (when *keyHandle* is an ECC key)
TPM_RC_INSUFFICIENT	*secret* is invalid (when *keyHandle* is an ECC key)
TPM_RC_INTEGRITY	*credentialBlob* fails integrity test
TPM_RC_NO_RESULT	*secret* is invalid (when *keyHandle* is an ECC key)
TPM_RC_SIZE	*secret* size is invalid or the *credentialBlob* does not unmarshal correctly
TPM_RC_TYPE	*keyHandle* does not reference an asymmetric key.
TPM_RC_VALUE	*secret* is invalid (when *keyHandle* is an RSA key)

```
TPM_RC
TPM2_ActivateCredential(
    ActivateCredential_In   *in,            // IN: input parameter list
    ActivateCredential_Out  *out            // OUT: output parameter list
    )
{
    TPM_RC                   result = TPM_RC_SUCCESS;
    OBJECT                  *object;            // decrypt key
    OBJECT                  *activateObject;    // key associated with credential
    TPM2B_DATA               data;          // credential data

// Input Validation

    // Get decrypt key pointer
    object = HandleToObject(in->keyHandle);

    // Get certificated object pointer
    activateObject = HandleToObject(in->activateHandle);

    // input decrypt key must be an asymmetric, restricted decryption key
    if(!CryptIsAsymAlgorithm(object->publicArea.type)
       || !IS_ATTRIBUTE(object->publicArea.objectAttributes, TPMA_OBJECT, decrypt)
       || !IS_ATTRIBUTE(object->publicArea.objectAttributes,
                        TPMA_OBJECT, restricted))
        return TPM_RCS_TYPE + RC_ActivateCredential_keyHandle;

// Command output

    // Decrypt input credential data via asymmetric decryption.  A
    // TPM_RC_VALUE, TPM_RC_KEY or unmarshal errors may be returned at this
    // point
    result = CryptSecretDecrypt(object, NULL, IDENTITY_STRING, &in->secret, &data);
    if(result != TPM_RC_SUCCESS)
    {
        if(result == TPM_RC_KEY)
            return TPM_RC_FAILURE;
        return RcSafeAddToResult(result, RC_ActivateCredential_secret);
    }

```

```
    // Retrieve secret data.  A TPM_RC_INTEGRITY error or unmarshal
    // errors may be returned at this point
    result = CredentialToSecret(&in->credentialBlob.b,
                                &activateObject->name.b,
                                &data.b,
                                object,
                                &out->certInfo);
    if(result != TPM_RC_SUCCESS)
        return RcSafeAddToResult(result, RC_ActivateCredential_credentialBlob);

    return TPM_RC_SUCCESS;
}
#endif // CC_ActivateCredential
```

12.6 TPM2_MakeCredential

12.6.1 General Description

This command allows the TPM to perform the actions required of a Certificate Authority (CA) in creating a TPM2B_ID_OBJECT containing an activation credential.

NOTE The input *credential* parameter is an application defined value. It is typically a symmetric key or seed that is used to encrypt a certificate. See the TPM2_ActivateCredential *certInfo* output parameter.

The TPM will produce a TPM2B_ID_OBJECT according to the methods in “Credential Protection” in TPM 2.0 Part 1.

The loaded public area referenced by *handle* is required to be the public area of a Storage key, otherwise, the credential cannot be properly sealed.

This command does not use any TPM secrets nor does it require authorization. It is a convenience function, using the TPM to perform cryptographic calculations that could be done externally.

12.6.2 Command and Response

Table 29 — TPM2_MakeCredential Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit, encrypt, or decrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_MakeCredential
TPMI_DH_OBJECT	handle	loaded public area, used to encrypt the sensitive area containing the credential key Auth Index: None
TPM2B_DIGEST	credential	the credential information
TPM2B_NAME	objectName	Name of the object to which the credential applies

Table 30 — TPM2_MakeCredential Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_ID_OBJECT	credentialBlob	the credential
TPM2B_ENCRYPTED_SECRET	secret	*handle* algorithm-dependent data that wraps the key that encrypts *credentialBlob*

12.6.3 Detailed Actions

```
#include "Tpm.h"
#include "MakeCredential_fp.h"
#if CC_MakeCredential  // Conditional expansion of this file
#include "Object_spt_fp.h"
```

Error Returns	Meaning
TPM_RC_KEY	*handle* referenced an ECC key that has a unique field that is not a point on the curve of the key
TPM_RC_SIZE	*credential* is larger than the digest size of Name algorithm of *handle*
TPM_RC_TYPE	*handle* does not reference an asymmetric decryption key

```
TPM_RC
TPM2_MakeCredential(
    MakeCredential_In   *in,            // IN: input parameter list
    MakeCredential_Out  *out            // OUT: output parameter list
    )
{
    TPM_RC               result = TPM_RC_SUCCESS;

    OBJECT              *object;
    TPM2B_DATA           data;

// Input Validation

    // Get object pointer
    object = HandleToObject(in->handle);

    // input key must be an asymmetric, restricted decryption key
    // NOTE: Needs to be restricted to have a symmetric value.
    if(!CryptIsAsymAlgorithm(object->publicArea.type)
       || !IS_ATTRIBUTE(object->publicArea.objectAttributes, TPMA_OBJECT, decrypt)
       || !IS_ATTRIBUTE(object->publicArea.objectAttributes,
                        TPMA_OBJECT, restricted))
        return TPM_RCS_TYPE + RC_MakeCredential_handle;

    // The credential information may not be larger than the digest size used for
    // the Name of the key associated with handle.
    if(in->credential.t.size > CryptHashGetDigestSize(object->publicArea.nameAlg))
        return TPM_RCS_SIZE + RC_MakeCredential_credential;

// Command Output

    // Make encrypt key and its associated secret structure.
    out->secret.t.size = sizeof(out->secret.t.secret);
    result = CryptSecretEncrypt(object, IDENTITY_STRING, &data, &out->secret);
    if(result != TPM_RC_SUCCESS)
        return result;

    // Prepare output credential data from secret
    SecretToCredential(&in->credential, &in->objectName.b, &data.b,
                       object, &out->credentialBlob);

    return TPM_RC_SUCCESS;
}
#endif // CC_MakeCredential
```

12.7 TPM2_Unseal

12.7.1 General Description

This command returns the data in a loaded Sealed Data Object.

NOTE 1 A random, TPM-generated, Sealed Data Object may be created by the TPM with TPM2_Create() or TPM2_CreatePrimary() using the template for a Sealed Data Object.

NOTE 2 TPM 1.2 hard coded PCR authorization. TPM 2.0 PCR authorization requires a policy.

The returned value may be encrypted using authorization session encryption.

If either *restricted*, *decrypt,* or *sign* is SET in the attributes of *itemHandle,* then the TPM shall return TPM_RC_ATTRIBUTES. If the *type* of *itemHandle* is not TPM_ALG_KEYEDHASH, then the TPM shall return TPM_RC_TYPE.

12.7.2 Command and Response

Table 31 — TPM2_Unseal Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_Unseal
TPMI_DH_OBJECT	@itemHandle	handle of a loaded data object Auth Index: 1 Auth Role: USER

Table 32 — TPM2_Unseal Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_SENSITIVE_DATA	outData	unsealed data Size of *outData* is limited to be no more than 128 octets.

12.7.3 Detailed Actions

```
#include "Tpm.h"
#include "Unseal_fp.h"
#if CC_Unseal // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_ATTRIBUTES	*itemHandle* has wrong attributes
TPM_RC_TYPE	*itemHandle* is not a KEYEDHASH data object

```
TPM_RC
TPM2_Unseal(
    Unseal_In           *in,
    Unseal_Out          *out
    )
{
    OBJECT                  *object;
// Input Validation
    // Get pointer to loaded object
    object = HandleToObject(in->itemHandle);

    // Input handle must be a data object
    if(object->publicArea.type != TPM_ALG_KEYEDHASH)
        return TPM_RCS_TYPE + RC_Unseal_itemHandle;
    if(IS_ATTRIBUTE(object->publicArea.objectAttributes, TPMA_OBJECT, decrypt)
       || IS_ATTRIBUTE(object->publicArea.objectAttributes, TPMA_OBJECT, sign)
       || IS_ATTRIBUTE(object->publicArea.objectAttributes, TPMA_OBJECT, restricted))
        return TPM_RCS_ATTRIBUTES + RC_Unseal_itemHandle;
// Command Output
    // Copy data
    out->outData = object->sensitive.sensitive.bits;
    return TPM_RC_SUCCESS;
}
#endif // CC_Unseal
```

12.8 TPM2_ObjectChangeAuth

12.8.1 General Description

This command is used to change the authorization secret for a TPM-resident object.

If successful, a new private area for the TPM-resident object associated with *objectHandle* is returned, which includes the new authorization value.

This command does not change the authorization of the TPM-resident object on which it operates. Therefore, the old authValue (of the TPM-resident object) is used when generating the response HMAC key if required.

NOTE 1 The returned *outPrivate* will need to be loaded before the new authorization will apply.

NOTE 2 The TPM-resident object may be persistent and changing the authorization value of the persistent object could prevent other users from accessing the object. This is why this command does not change the TPM-resident object.

EXAMPLE If a persistent key is being used as a Storage Root Key and the authorization of the key is a well-known value so that the key can be used generally, then changing the authorization value in the persistent key would deny access to other users.

This command may not be used to change the authorization value for an NV Index or a Primary Object.

NOTE 3 If an NV Index is to have a new authorization, it is done with TPM2_NV_ChangeAuth().

NOTE 4 If a Primary Object is to have a new authorization, it needs to be recreated (TPM2_CreatePrimary()).

12.8.2 Command and Response

Table 33 — TPM2_ObjectChangeAuth Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_ObjectChangeAuth
TPMI_DH_OBJECT	@objectHandle	handle of the object Auth Index: 1 Auth Role: ADMIN
TPMI_DH_OBJECT	parentHandle	handle of the parent Auth Index: None
TPM2B_AUTH	newAuth	new authorization value

Table 34 — TPM2_ObjectChangeAuth Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_PRIVATE	outPrivate	private area containing the new authorization value

12.8.3 Detailed Actions

```
#include "Tpm.h"
#include "ObjectChangeAuth_fp.h"
#if CC_ObjectChangeAuth  // Conditional expansion of this file
#include "Object_spt_fp.h"
```

Error Returns	Meaning
TPM_RC_SIZE	*newAuth* is larger than the size of the digest of the Name algorithm of *objectHandle*
TPM_RC_TYPE	the key referenced by *parentHandle* is not the parent of the object referenced by *objectHandle*; or *objectHandle* is a sequence object.

```
TPM_RC
TPM2_ObjectChangeAuth(
    ObjectChangeAuth_In     *in,            // IN: input parameter list
    ObjectChangeAuth_Out    *out            // OUT: output parameter list
    )
{
    TPMT_SENSITIVE           sensitive;

    OBJECT                  *object = HandleToObject(in->objectHandle);
    TPM2B_NAME               QNCompare;

// Input Validation

    // Can not change authorization on sequence object
    if(ObjectIsSequence(object))
        return TPM_RCS_TYPE + RC_ObjectChangeAuth_objectHandle;

    // Make sure that the authorization value is consistent with the nameAlg
    if(!AdjustAuthSize(&in->newAuth, object->publicArea.nameAlg))
        return TPM_RCS_SIZE + RC_ObjectChangeAuth_newAuth;

    // Parent handle should be the parent of object handle.  In this
    // implementation we verify this by checking the QN of object.  Other
    // implementation may choose different method to verify this attribute.
    ComputeQualifiedName(in->parentHandle,
                         object->publicArea.nameAlg,
                         &object->name, &QNCompare);
    if(!MemoryEqual2B(&object->qualifiedName.b, &QNCompare.b))
        return TPM_RCS_TYPE + RC_ObjectChangeAuth_parentHandle;

// Command Output
    // Prepare the sensitive area with the new authorization value
    sensitive = object->sensitive;
    sensitive.authValue = in->newAuth;

    // Protect the sensitive area
    SensitiveToPrivate(&sensitive, &object->name, HandleToObject(in->parentHandle),
                       object->publicArea.nameAlg,
                       &out->outPrivate);
    return TPM_RC_SUCCESS;
}
#endif // CC_ObjectChangeAuth
```


12.9 TPM2_CreateLoaded

12.9.1 General Description

This command creates an object and loads it in the TPM. This command allows creation of any type of object (Primary, Ordinary, or Derived) depending on the type of *parentHandle.* If *parentHandle* references a Primary Seed, then a Primary Object is created; if *parentHandle* references a Storage Parent, then an Ordinary Object is created; and if *parentHandle* references a Derivation Parent, then a Derived Object is generated.

The input validation is the same as for TPM2_Create() and TPM2_CreatePrimary() with one exception: when *parentHandle* references a Derivation Parent, then *sensitiveDataOrigin* in *inPublic* is required to be CLEAR.

Note 1 In the general descriptions of TPM2_Create() and TPM2_CreatePrimary() the validations refer to a TPMT_PUBLIC structure that is in *inPublic.* For TPM2_CreateLoaded(), *inPublic* is a TPM2B_TEMPLATE that may contain a TPMT_PUBLIC that is used for object creation. For object derivation, the *unique* field can contain a *label* and *context* that are used in the derivation process. To allow both the TPMT_PUBLIC and the derivation variation, a TPM2B_TEMPLATE is used. When referring to the checks in TPM2_Create() and TPM2_CreatePrimary(), TPM2B_TEMPLATE should be assumed to contain a TPMT_PUBLIC.

If *parentHandle* references a Derivation Parent, then the TPM may return TPM_RC_TYPE if the key type to be generated is an RSA key.

If *parentHandle* references a Derivation Parent or a Primary Seed, then *outPrivate* will be an Empty Buffer.

NOTE 2 Returning outPrivate would imply that the returned primary or derived object can be loaded and it cannot. It can only be re-derived.

A primary key cannot be loaded is because loading a key is a way to attack the protections of a key (e.g. using DPA). A saved context for a primary object is protected. The TPM will go into failure mode if the integrity of a saved context is good but the fingerprint doesn't decrypt. It is not possible to have these protections on loaded objects because this would be a simple way for an attacker to put the TPM into failure mode Saved contexts are assumed to be under control of the driver but loaded objects are not.

If all objects were derived from their parents then, load could not be used as an attack. However, that would preclude importation of objects and key hierarchies.

NOTE 3 Unlike TPM2_Create() and TPM2_CreatePrimary(), this command does not return creation data. If creation data is needed, then TPM2_Create() or TPM2_CreatePrimary() should be used.

12.9.2 Command and Response

Table 35 — TPM2_CreateLoaded Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_CreateLoade
TPMI_DH_PARENT+	@parentHandle	Handle of a transient storage key, a persistent storage key, TPM_RH_ENDORSEMENT, TPM_RH_OWNER, TPM_RH_PLATFORM+{PP}, or TPM_RH_NULL Auth Index: 1 Auth Role: USER
TPM2B_SENSITIVE_CREATE	inSensitive	the sensitive data, see TPM 2.0 Part 1 Sensitive Values
TPM2B_TEMPLATE	inPublic	the public template

Table 36 — TPM2_CreateLoaded Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM_HANDLE	objectHandle	handle of type TPM_HT_TRANSIENT for created object
TPM2B_PRIVATE	outPrivate	the sensitive area of the object (optional)
TPM2B_PUBLIC	outPublic	the public portion of the created object
TPM2B_NAME	name	the name of the created object

12.9.3 Detailed Actions

```
#include "Tpm.h"
#include "CreateLoaded_fp.h"
#if CC_CreateLoaded  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_ATTRIBUTES	*sensitiveDataOrigin* is CLEAR when *sensitive.data* is an Empty Buffer; *fixedTPM*, *fixedParent*, or *encryptedDuplication* attributes are inconsistent between themselves or with those of the parent object; inconsistent *restricted*, *decrypt* and *sign* attributes; attempt to inject sensitive data for an asymmetric key; attempt to create a symmetric cipher key that is not a decryption key
TPM_RC_KDF	incorrect KDF specified for decrypting keyed hash object
TPM_RC_KEY	the value of a provided symmetric key is not allowed
TPM_RC_OBJECT_MEMORY	there is no free slot for the object
TPM_RC_SCHEME	inconsistent attributes *decrypt*, *sign*, *restricted* and key's scheme ID; or hash algorithm is inconsistent with the scheme ID for keyed hash object
TPM_RC_SIZE	size of public authorization policy or sensitive authorization value does not match digest size of the name algorithm sensitive data size for the keyed hash object is larger than is allowed for the scheme
TPM_RC_SYMMETRIC	a storage key with no symmetric algorithm specified; or non-storage key with symmetric algorithm different from TPM_ALG_NULL
TPM_RC_TYPE	cannot create the object of the indicated type (usually only occurs if trying to derive an RSA key).

```
TPM_RC
TPM2_CreateLoaded(
    CreateLoaded_In     *in,            // IN: input parameter list
    CreateLoaded_Out    *out            // OUT: output parameter list
    )
{
    TPM_RC                     result = TPM_RC_SUCCESS;
    OBJECT                    *parent = HandleToObject(in->parentHandle);
    OBJECT                    *newObject;
    BOOL                       derivation;
    TPMT_PUBLIC               *publicArea;
    RAND_STATE                 randState;
    RAND_STATE                *rand = &randState;
    TPMS_DERIVE                labelContext;

// Input Validation

    // How the public area is unmarshaled is determined by the parent, so
    // see if parent is a derivation parent
    derivation = (parent != NULL && parent->attributes.derivation);

    // If the parent is an object, then make sure that it is either a parent or
    // derivation parent
    if(parent != NULL && !parent->attributes.isParent && !derivation)
        return TPM_RCS_TYPE + RC_CreateLoaded_parentHandle;

    // Get a spot in which to create the newObject
    newObject = FindEmptyObjectSlot(&out->objectHandle);
    if(newObject == NULL)
        return TPM_RC_OBJECT_MEMORY;
```



```

    // Do this to save typing
    publicArea = &newObject->publicArea;

    // Unmarshal the template into the object space. TPM2_Create() and
    // TPM2_CreatePrimary() have the publicArea unmarshaled by CommandDispatcher.
    // This command is different because of an unfortunate property of the
    // unique field of an ECC key. It is a structure rather than a single TPM2B. If
    // if had been a TPM2B, then the label and context could be within a TPM2B and
    // unmarshaled like other public areas. Since it is not, this command needs its
    // on template that is a TPM2B that is unmarshaled as a BYTE array with a
    // its own unmarshal function.
    result = UnmarshalToPublic(publicArea, &in->inPublic, derivation,
                               &labelContext);
    if(result != TPM_RC_SUCCESS)
        return result + RC_CreateLoaded_inPublic;

    // Validate that the authorization size is appropriate
    if(!AdjustAuthSize(&in->inSensitive.sensitive.userAuth, publicArea->nameAlg))
        return TPM_RCS_SIZE + RC_CreateLoaded_inSensitive;

    // Command output
    if(derivation)
    {
        TPMT_KEYEDHASH_SCHEME       *scheme;
        scheme = &parent->publicArea.parameters.keyedHashDetail.scheme;

        // SP800-108 is the only KDF supported by this implementation and there is
        // no default hash algorithm.
        pAssert(scheme->details.xor.hashAlg != TPM_ALG_NULL
                && scheme->details.xor.kdf == TPM_ALG_KDF1_SP800_108);
        // Don't derive RSA keys
        if(publicArea->type == ALG_RSA_VALUE)
            return TPM_RCS_TYPE + RC_CreateLoaded_inPublic;
        // sensitiveDataOrigin has to be CLEAR in a derived object. Since this
        // is specific to a derived object, it is checked here.
        if(IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT,
                        sensitiveDataOrigin))
            return TPM_RCS_ATTRIBUTES;
        // Check the reset of the attributes
        result = PublicAttributesValidation(parent, publicArea);
        if(result != TPM_RC_SUCCESS)
            return RcSafeAddToResult(result, RC_CreateLoaded_inPublic);
        // Process the template and sensitive areas to get the actual 'label' and
        // 'context' values to be used for this derivation.
        result = SetLabelAndContext(&labelContext, &in->inSensitive.sensitive.data);
        if(result != TPM_RC_SUCCESS)
            return result;
        // Set up the KDF for object generation
        DRBG_InstantiateSeededKdf((KDF_STATE *)rand,
                                  scheme->details.xor.hashAlg,
                                  scheme->details.xor.kdf,
                                  &parent->sensitive.sensitive.bits.b,
                                  &labelContext.label.b,
                                  &labelContext.context.b,
                                  MAX_DERIVATION_BITS);
        // Clear the sensitive size so that the creation functions will not try
        // to use this value.
        in->inSensitive.sensitive.data.t.size = 0;
    }
    else
    {
        // Check attributes in input public area. CreateChecks() checks the things
        // that are unique to creation and then validates the attributes and values
        // that are common to create and load.
        result = CreateChecks(parent, publicArea,
```

```
                              in->inSensitive.sensitive.data.t.size);
        if(result != TPM_RC_SUCCESS)
            return RcSafeAddToResult(result, RC_CreateLoaded_inPublic);
        // Creating a primary object
        if(parent == NULL)
        {
            TPM2B_NAME              name;
            newObject->attributes.primary = SET;
            if(in->parentHandle == TPM_RH_ENDORSEMENT)
                newObject->attributes.epsHierarchy = SET;
            // If so, use the primary seed and the digest of the template
            // to seed the DRBG
            result = DRBG_InstantiateSeeded((DRBG_STATE *)rand,
                                   &HierarchyGetPrimarySeed(in->parentHandle)->b,
                                   PRIMARY_OBJECT_CREATION,
                                   (TPM2B *)PublicMarshalAndComputeName(publicArea,
                                                                       &name),
                                   &in->inSensitive.sensitive.data.b);
            if(result != TPM_RC_SUCCESS)
                return result;
        }
        else
        {
           // This is an ordinary object so use the normal random number generator
            rand = NULL;
        }
    }
// Internal data update
    // Create the object
    result = CryptCreateObject(newObject, &in->inSensitive.sensitive, rand);
    if(result != TPM_RC_SUCCESS)
        return result;
    // if this is not a Primary key and not a derived key, then return the sensitive
    // area
    if(parent != NULL && !derivation)
        // Prepare output private data from sensitive
        SensitiveToPrivate(&newObject->sensitive, &newObject->name,
                           parent, newObject->publicArea.nameAlg,
                           &out->outPrivate);
    else
        out->outPrivate.t.size = 0;
    // Set the remaining return values
    out->outPublic.publicArea = newObject->publicArea;
    out->name = newObject->name;
    // Set the remaining attributes for a loaded object
    ObjectSetLoadedAttributes(newObject, in->parentHandle);

    return result;
}
#endif // CC_CreateLoaded
```

13 Duplication Commands

13.1 TPM2_Duplicate

13.1.1 General Description

This command duplicates a loaded object so that it may be used in a different hierarchy. The new parent key for the duplicate may be on the same or different TPM or TPM_RH_NULL. Only the public area of *newParentHandle* is required to be loaded.

NOTE 1 Since the new parent may only be extant on a different TPM, it is likely that the new parent's sensitive area could not be loaded in the TPM from which *objectHandle* is being duplicated.

If *encryptedDuplication* is SET in the object being duplicated, then the TPM shall return TPM_RC_SYMMETRIC if *symmetricAlg.algorithm* is TPM_ALG_NULL or TPM_RC_HIERARCHY if *newParentHandle* is TPM_RH_NULL.

The authorization for this command shall be with a policy session.

If *fixedParent* of *objectHandle→attributes* is SET, the TPM shall return TPM_RC_ATTRIBUTES. If *objectHandle→nameAlg* is TPM_ALG_NULL, the TPM shall return TPM_RC_TYPE.

The *policySession→commandCode* parameter in the policy session is required to be TPM_CC_Duplicate to indicate that authorization for duplication has been provided. This indicates that the policy that is being used is a policy that is for duplication, and not a policy that would approve another use. That is, authority to use an object does not grant authority to duplicate the object.

The policy is likely to include cpHash in order to restrict where duplication can occur. If TPM2_PolicyCpHash() has been executed as part of the policy, the *policySession→cpHash* is compared to the cpHash of the command.

If TPM2_PolicyDuplicationSelect() has been executed as part of the policy, the *policySession→nameHash* is compared to

$$\mathbf{H}_{policyAlg}(objectHandle{\rightarrow}Name \mid\mid newParentHandle{\rightarrow}Name) \tag{2}$$

If the compared hashes are not the same, then the TPM shall return TPM_RC_POLICY_FAIL.

NOTE 2 It is allowed that policySesion→nameHash and policySession→cpHash share the same memory space.

NOTE 3 A duplication policy is not required to have either TPM2_PolicyDuplicationSelect() or TPM2_PolicyCpHash() as part of the policy. If neither is present, then the duplication policy may be satisfied with a policy that only contains TPM2_PolicyCommandCode(*code* = TPM_CC_Duplicate).

The TPM shall follow the process of encryption defined in the "Duplication" subclause of "Protected Storage Hierarchy" in TPM 2.0 Part 1.

13.1.2 Command and Response

Table 37 — TPM2_Duplicate Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_Duplicate
TPMI_DH_OBJECT	@objectHandle	loaded object to duplicate Auth Index: 1 Auth Role: DUP
TPMI_DH_OBJECT+	newParentHandle	shall reference the public area of an asymmetric key Auth Index: None
TPM2B_DATA	encryptionKeyIn	optional symmetric encryption key The size for this key is set to zero when the TPM is to generate the key. This parameter may be encrypted.
TPMT_SYM_DEF_OBJECT+	symmetricAlg	definition for the symmetric algorithm to be used for the inner wrapper may be TPM_ALG_NULL if no inner wrapper is applied

Table 38 — TPM2_Duplicate Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_DATA	encryptionKeyOut	If the caller provided an encryption key or if *symmetricAlg* was TPM_ALG_NULL, then this will be the Empty Buffer; otherwise, it shall contain the TPM-generated, symmetric encryption key for the inner wrapper.
TPM2B_PRIVATE	duplicate	private area that may be encrypted by *encryptionKeyIn*; and may be doubly encrypted
TPM2B_ENCRYPTED_SECRET	outSymSeed	seed protected by the asymmetric algorithms of new parent (NP)

13.1.3 Detailed Actions

```
#include "Tpm.h"
#include "Duplicate_fp.h"
#if CC_Duplicate  // Conditional expansion of this file
#include "Object_spt_fp.h"
```

Error Returns	Meaning
TPM_RC_ATTRIBUTES	key to duplicate has *fixedParent* SET
TPM_RC_HASH	for an RSA key, the *nameAlg* digest size for the *newParent* is not compatible with the key size
TPM_RC_HIERARCHY	*encryptedDuplication* is SET and *newParentHandle* specifies Null Hierarchy
TPM_RC_KEY	*newParentHandle* references invalid ECC key (public point not on the curve)
TPM_RC_SIZE	input encryption key size does not match the size specified in symmetric algorithm
TPM_RC_SYMMETRIC	*encryptedDuplication* is SET but no symmetric algorithm is provided
TPM_RC_TYPE	*newParentHandle* is neither a storage key nor TPM_RH_NULL; or the object has a NULL *nameAlg*
TPM_RC_VALUE	for an RSA *newParent*, the sizes of the digest and the encryption key are too large to be OAEP encoded

```
TPM_RC
TPM2_Duplicate(
    Duplicate_In    *in,            // IN: input parameter list
    Duplicate_Out   *out            // OUT: output parameter list
    )
{
    TPM_RC                  result = TPM_RC_SUCCESS;
    TPMT_SENSITIVE          sensitive;

    UINT16                  innerKeySize = 0; // encrypt key size for inner wrap

    OBJECT                  *object;
    OBJECT                  *newParent;
    TPM2B_DATA              data;

// Input Validation

    // Get duplicate object pointer
    object = HandleToObject(in->objectHandle);
    // Get new parent
    newParent = HandleToObject(in->newParentHandle);

    // duplicate key must have fixParent bit CLEAR.
    if(IS_ATTRIBUTE(object->publicArea.objectAttributes, TPMA_OBJECT, fixedParent))
        return TPM_RCS_ATTRIBUTES + RC_Duplicate_objectHandle;

    // Do not duplicate object with NULL nameAlg
    if(object->publicArea.nameAlg == TPM_ALG_NULL)
        return TPM_RCS_TYPE + RC_Duplicate_objectHandle;

    // new parent key must be a storage object or TPM_RH_NULL
    if(in->newParentHandle != TPM_RH_NULL
       && !ObjectIsStorage(in->newParentHandle))
        return TPM_RCS_TYPE + RC_Duplicate_newParentHandle;
```

```

    // If the duplicated object has encryptedDuplication SET, then there must be
    // an inner wrapper and the new parent may not be TPM_RH_NULL
    if(IS_ATTRIBUTE(object->publicArea.objectAttributes, TPMA_OBJECT,
                    encryptedDuplication))
    {
        if(in->symmetricAlg.algorithm == TPM_ALG_NULL)
            return TPM_RCS_SYMMETRIC + RC_Duplicate_symmetricAlg;
        if(in->newParentHandle == TPM_RH_NULL)
            return TPM_RCS_HIERARCHY + RC_Duplicate_newParentHandle;
    }

    if(in->symmetricAlg.algorithm == TPM_ALG_NULL)
    {
        // if algorithm is TPM_ALG_NULL, input key size must be 0
        if(in->encryptionKeyIn.t.size != 0)
            return TPM_RCS_SIZE + RC_Duplicate_encryptionKeyIn;
    }
    else
    {
        // Get inner wrap key size
        innerKeySize = in->symmetricAlg.keyBits.sym;

        // If provided the input symmetric key must match the size of the algorithm
        if(in->encryptionKeyIn.t.size != 0
           && in->encryptionKeyIn.t.size != (innerKeySize + 7) / 8)
            return TPM_RCS_SIZE + RC_Duplicate_encryptionKeyIn;
    }

// Command Output

    if(in->newParentHandle != TPM_RH_NULL)
    {
        // Make encrypt key and its associated secret structure.  A TPM_RC_KEY
        // error may be returned at this point
        out->outSymSeed.t.size = sizeof(out->outSymSeed.t.secret);
        result = CryptSecretEncrypt(newParent, DUPLICATE_STRING, &data,
                                    &out->outSymSeed);
        if(result != TPM_RC_SUCCESS)
            return result;
    }
    else
    {
        // Do not apply outer wrapper
        data.t.size = 0;
        out->outSymSeed.t.size = 0;
    }

    // Copy sensitive area
    sensitive = object->sensitive;

    // Prepare output private data from sensitive.
    // Note: If there is no encryption key, one will be provided by
    // SensitiveToDuplicate(). This is why the assignment of encryptionKeyIn to
    // encryptionKeyOut will work properly and is not conditional.
    SensitiveToDuplicate(&sensitive, &object->name.b, newParent,
                         object->publicArea.nameAlg, &data.b,
                         &in->symmetricAlg, &in->encryptionKeyIn,
                         &out->duplicate);

    out->encryptionKeyOut = in->encryptionKeyIn;

    return TPM_RC_SUCCESS;
}
#endif // CC_Duplicate
```


13.2 TPM2_Rewrap

13.2.1 General Description

This command allows the TPM to serve in the role as a Duplication Authority. If proper authorization for use of the *oldParent* is provided, then an HMAC key and a symmetric key are recovered from *inSymSeed* and used to integrity check and decrypt *inDuplicate.* A new protection seed value is generated according to the methods appropriate for *newParent* and the blob is re-encrypted and a new integrity value is computed. The re-encrypted blob is returned in *outDuplicate* and the symmetric key returned in *outSymKey*.

In the rewrap process, L is "DUPLICATE" (see TPM 2.0 Part 1, *Terms and Definitions*).

If *inSymSeed* has a zero length, then *oldParent* is required to be TPM_RH_NULL and no decryption of *inDuplicate* takes place.

If *newParent* is TPM_RH_NULL, then no encryption is performed on *outDuplicate. outSymSeed* will have a zero length. See TPM 2.0 Part 2 *encryptedDuplication*.

13.2.2 Command and Response

Table 39 — TPM2_Rewrap Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_Rewrap
TPMI_DH_OBJECT+	@oldParent	parent of object Auth Index: 1 Auth Role: User
TPMI_DH_OBJECT+	newParent	new parent of the object Auth Index: None
TPM2B_PRIVATE	inDuplicate	an object encrypted using symmetric key derived from *inSymSeed*
TPM2B_NAME	name	the Name of the object being rewrapped
TPM2B_ENCRYPTED_SECRET	inSymSeed	the seed for the symmetric key and HMAC key needs *oldParent* private key to recover the seed and generate the symmetric key

Table 40 — TPM2_Rewrap Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_PRIVATE	outDuplicate	an object encrypted using symmetric key derived from *outSymSeed*
TPM2B_ENCRYPTED_SECRET	outSymSeed	seed for a symmetric key protected by *newParent* asymmetric key

13.2.3 Detailed Actions

```
#include "Tpm.h"
#include "Rewrap_fp.h"
#if CC_Rewrap  // Conditional expansion of this file
#include "Object_spt_fp.h"
```

Error Returns	Meaning
TPM_RC_ATTRIBUTES	*newParent* is not a decryption key
TPM_RC_HANDLE	*oldParent* does not consistent with *inSymSeed*
TPM_RC_INTEGRITY	the integrity check of *inDuplicate* failed
TPM_RC_KEY	for an ECC key, the public key is not on the curve of the curve ID
TPM_RC_KEY_SIZE	the decrypted input symmetric key size does not matches the symmetric algorithm key size of *oldParent*
TPM_RC_TYPE	*oldParent* is not a storage key, or '*newParent* is not a storage key
TPM_RC_VALUE	for an '*oldParent*; RSA key, the data to be decrypted is greater than the public exponent
errors	errors during unmarshaling the input encrypted buffer to a ECC public key, or unmarshal the private buffer to sensitive

```
TPM_RC
TPM2_Rewrap(
    Rewrap_In       *in,            // IN: input parameter list
    Rewrap_Out      *out            // OUT: output parameter list
    )
{
    TPM_RC                  result = TPM_RC_SUCCESS;
    TPM2B_DATA              data;           // symmetric key
    UINT16                  hashSize = 0;
    TPM2B_PRIVATE           privateBlob;    // A temporary private blob
                                            // to transit between old
                                            // and new wrappers
// Input Validation
    if((in->inSymSeed.t.size == 0 && in->oldParent != TPM_RH_NULL)
       || (in->inSymSeed.t.size != 0 && in->oldParent == TPM_RH_NULL))
        return TPM_RCS_HANDLE + RC_Rewrap_oldParent;
    if(in->oldParent != TPM_RH_NULL)
    {
        OBJECT              *oldParent = HandleToObject(in->oldParent);

        // old parent key must be a storage object
        if(!ObjectIsStorage(in->oldParent))
            return TPM_RCS_TYPE + RC_Rewrap_oldParent;
        // Decrypt input secret data via asymmetric decryption.  A
        // TPM_RC_VALUE, TPM_RC_KEY or unmarshal errors may be returned at this
        // point
        result = CryptSecretDecrypt(oldParent, NULL, DUPLICATE_STRING,
                                    &in->inSymSeed, &data);
        if(result != TPM_RC_SUCCESS)
            return TPM_RCS_VALUE + RC_Rewrap_inSymSeed;
        // Unwrap Outer
        result = UnwrapOuter(oldParent, &in->name.b,
                             oldParent->publicArea.nameAlg, &data.b,
                             FALSE,
                             in->inDuplicate.t.size, in->inDuplicate.t.buffer);
        if(result != TPM_RC_SUCCESS)
            return RcSafeAddToResult(result, RC_Rewrap_inDuplicate);
```

```
        // Copy unwrapped data to temporary variable, remove the integrity field
        hashSize = sizeof(UINT16) +
            CryptHashGetDigestSize(oldParent->publicArea.nameAlg);
        privateBlob.t.size = in->inDuplicate.t.size - hashSize;
        pAssert(privateBlob.t.size <= sizeof(privateBlob.t.buffer));
        MemoryCopy(privateBlob.t.buffer, in->inDuplicate.t.buffer + hashSize,
                   privateBlob.t.size);
    }
    else
    {
        // No outer wrap from input blob.  Direct copy.
        privateBlob = in->inDuplicate;
    }
    if(in->newParent != TPM_RH_NULL)
    {
        OBJECT          *newParent;
        newParent = HandleToObject(in->newParent);

        // New parent must be a storage object
        if(!ObjectIsStorage(in->newParent))
            return TPM_RCS_TYPE + RC_Rewrap_newParent;
        // Make new encrypt key and its associated secret structure.  A
        // TPM_RC_VALUE error may be returned at this point if RSA algorithm is
        // enabled in TPM
        out->outSymSeed.t.size = sizeof(out->outSymSeed.t.secret);
        result = CryptSecretEncrypt(newParent, DUPLICATE_STRING, &data,
                                    &out->outSymSeed);
        if(result != TPM_RC_SUCCESS)
            return result;
        // Copy temporary variable to output, reserve the space for integrity
        hashSize = sizeof(UINT16) +
            CryptHashGetDigestSize(newParent->publicArea.nameAlg);
 // Make sure that everything fits into the output buffer
 // Note: this is mostly only an issue if there was no outer wrapper on
 // 'inDuplicate'. It could be as large as a TPM2B_PRIVATE buffer. If we add
 // a digest for an outer wrapper, it won't fit anymore.
        if((privateBlob.t.size + hashSize) > sizeof(out->outDuplicate.t.buffer))
            return TPM_RCS_VALUE + RC_Rewrap_inDuplicate;
// Command output
        out->outDuplicate.t.size = privateBlob.t.size;
        pAssert(privateBlob.t.size
                <= sizeof(out->outDuplicate.t.buffer) - hashSize);
        MemoryCopy(out->outDuplicate.t.buffer + hashSize, privateBlob.t.buffer,
                   privateBlob.t.size);
        // Produce outer wrapper for output
        out->outDuplicate.t.size = ProduceOuterWrap(newParent, &in->name.b,
                                                    newParent->publicArea.nameAlg,
                                                    &data.b,
                                                    FALSE,
                                                    out->outDuplicate.t.size,
                                                    out->outDuplicate.t.buffer);
    }
    else  // New parent is a null key so there is no seed
    {
        out->outSymSeed.t.size = 0;

        // Copy privateBlob directly
        out->outDuplicate = privateBlob;
    }
    return TPM_RC_SUCCESS;
}
#endif // CC_Rewrap
```

13.3 TPM2_Import

13.3.1 General Description

This command allows an object to be encrypted using the symmetric encryption values of a Storage Key. After encryption, the object may be loaded and used in the new hierarchy. The imported object (*duplicate*) may be singly encrypted, multiply encrypted, or unencrypted.

If *fixedTPM* or *fixedParent* is SET in *objectPublic*, the TPM shall return TPM_RC_ATTRIBUTES.

If *encryptedDuplication* is SET in the object referenced by *parentHandle* and *encryptedDuplication* is CLEAR in *objectPublic,* the TPM may return TPM_RC_ATTRIBUTES.

If encryptedDuplication is SET in objectPublic, then *inSymSeed* and *encryptionKey* shall not be Empty buffers (TPM_RC_ATTRIBUTES). Recovery of the sensitive data of the object occurs in the TPM in a multi--step process in the following order:

a) If *inSymSeed* has a non-zero size:

   1) The asymmetric parameters and private key of *parentHandle* are used to recover the seed used in the creation of the HMAC key and encryption keys used to protect the duplication blob.

      NOTE 1 When recovering the seed from *inSymSeed*, *L* is "DUPLICATE".

   2) The integrity value in *duplicate.buffer.integrityOuter* is used to verify the integrity of the data blob, which is the remainder of *duplicate.buffer* (TPM_RC_INTEGRITY).

      NOTE 2 The data blob will contain a TPMT_SENSITIVE and may contain a TPM2B_DIGEST for the *innerIntegrity.*

   3) The symmetric key recovered in 1) is used to decrypt the data blob.

      NOTE 3 Checking the integrity before the data is used prevents attacks on the sensitive area by fuzzing the data and looking at the differences in the response codes.

b) If *encryptionKey* is not an Empty Buffer:

   1) Use *encryptionKey* to decrypt the inner blob.

   2) Use the TPM2B_DIGEST at the start of the inner blob to verify the integrity of the inner blob (TPM_RC_INTEGRITY).

c) Unmarshal the sensitive area

NOTE 4 It is not necessary to validate that the sensitive area data is cryptographically bound to the public area other than that the Name of the public area is included in the HMAC. However, if the binding is not validated by this command, the binding must be checked each time the object is loaded. For an object that is imported under a parent with *fixedTPM* SET, binding need only be checked at import. If the parent has *fixedTPM* CLEAR, then the binding needs to be checked each time the object is loaded, or before the TPM performs an operation for which the binding affects the outcome of the operation (for example, TPM2_PolicySigned() or TPM2_Certify()).

Similarly, if the new parent's *fixedTPM* is set, the *encryptedDuplication* state need only be checked at import.

If the new parent is not *fixedTPM*, then that object will be loadable on any TPM (including SW versions) on which the new parent exists. This means that, each time an object is loaded under a parent that is not *fixedTPM*, it is necessary to validate all of the properties of that object. If the parent is *fixedTPM*, then the new private blob is integrity protected by the TPM that "owns" the parent. So, it is sufficient to validate the object's properties (attribute and public-private binding) on import and not again.

If a weak symmetric key is being imported, the TPM shall return TPM_RC_KEY.

After integrity checks and decryption, the TPM will create a new symmetrically encrypted private area using the encryption key of the parent.

NOTE 5 The symmetric re-encryption is the normal integrity generation and symmetric encryption applied to a child object.

NOTE 6 Revision 01.16 of this specification required the ECC private key in *duplicate* to be padded.

13.3.2 Command and Response

Table 41 — TPM2_Import Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_Import
TPMI_DH_OBJECT	@parentHandle	the handle of the new parent for the object Auth Index: 1 Auth Role: USER
TPM2B_DATA	encryptionKey	the optional symmetric encryption key used as the inner wrapper for *duplicate* If *symmetricAlg* is TPM_ALG_NULL, then this parameter shall be the Empty Buffer.
TPM2B_PUBLIC	objectPublic	the public area of the object to be imported This is provided so that the integrity value for *duplicate* and the object attributes can be checked. NOTE Even if the integrity value of the object is not checked on input, the object Name is required to create the integrity value for the imported object.
TPM2B_PRIVATE	duplicate	the symmetrically encrypted duplicate object that may contain an inner symmetric wrapper
TPM2B_ENCRYPTED_SECRET	inSymSeed	the seed for the symmetric key and HMAC key *inSymSeed* is encrypted/encoded using the algorithms of *newParent*.
TPMT_SYM_DEF_OBJECT+	symmetricAlg	definition for the symmetric algorithm to use for the inner wrapper If this algorithm is TPM_ALG_NULL, no inner wrapper is present and *encryptionKey* shall be the Empty Buffer.

Table 42 — TPM2_Import Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_PRIVATE	outPrivate	the sensitive area encrypted with the symmetric key of *parentHandle*

13.3.3 Detailed Actions

```
#include "Tpm.h"
#include "Import_fp.h"
#if CC_Import  // Conditional expansion of this file
#include "Object_spt_fp.h"
```

Error Returns	Meaning
TPM_RC_ATTRIBUTES	*FixedTPM* and *fixedParent* of *objectPublic* are not both CLEAR; or *inSymSeed* is nonempty and *parentHandle* does not reference a decryption key; or *objectPublic* and *parentHandle* have incompatible or inconsistent attributes; or *encrytpedDuplication* is SET in *objectPublic* but the inner or outer wrapper is missing. Note that if the TPM provides parameter values, the parameter number will indicate *symmetricKey* (missing inner wrapper) or *inSymSeed* (missing outer wrapper)
TPM_RC_BINDING	*duplicate* and *objectPublic* are not cryptographically bound
TPM_RC_ECC_POINT	*inSymSeed* is nonempty and ECC point in *inSymSeed* is not on the curve
TPM_RC_HASH	*objectPublic* does not have a valid *nameAlg*
TPM_RC_INSUFFICIENT	*inSymSeed* is nonempty and failed to retrieve ECC point from the secret; or unmarshaling sensitive value from *duplicate* failed the result of *inSymSeed* decryption
TPM_RC_INTEGRITY	*duplicate* integrity is broken
TPM_RC_KDF	*objectPublic* representing decrypting keyed hash object specifies invalid KDF
TPM_RC_KEY	inconsistent parameters of *objectPublic*; or *inSymSeed* is nonempty and *parentHandle* does not reference a key of supported type; or invalid key size in *objectPublic* representing an asymmetric key
TPM_RC_NO_RESULT	*inSymSeed* is nonempty and multiplication resulted in ECC point at infinity
TPM_RC_OBJECT_MEMORY	no available object slot
TPM_RC_SCHEME	inconsistent attributes *decrypt*, *sign*, *restricted* and key's scheme ID in *objectPublic*; or hash algorithm is inconsistent with the scheme ID for keyed hash object
TPM_RC_SIZE	*authPolicy* size does not match digest size of the name algorithm in *objectPublic*; or *symmetricAlg* and *encryptionKey* have different sizes; or *inSymSeed* is nonempty and it size is not consistent with the type of *parentHandle*; or unmarshaling sensitive value from *duplicate* failed
TPM_RC_SYMMETRIC	*objectPublic* is either a storage key with no symmetric algorithm or a non-storage key with symmetric algorithm different from TPM_ALG_NULL
TPM_RC_TYPE	unsupported type of *objectPublic*; or *parentHandle* is not a storage key; or only the public portion of *parentHandle* is loaded; or *objectPublic* and *duplicate* are of different types
TPM_RC_VALUE	nonempty *inSymSeed* and its numeric value is greater than the modulus of the key referenced by *parentHandle* or *inSymSeed* is larger than the size of the digest produced by the name algorithm of the symmetric key referenced by *parentHandle*

```
TPM_RC
```



```
TPM2_Import(
    Import_In       *in,            // IN: input parameter list
    Import_Out      *out            // OUT: output parameter list
    )
{
    TPM_RC                  result = TPM_RC_SUCCESS;
    OBJECT                 *parentObject;
    TPM2B_DATA              data;                   // symmetric key
    TPMT_SENSITIVE          sensitive;
    TPM2B_NAME              name;
    TPMA_OBJECT             attributes;
    UINT16                  innerKeySize = 0;       // encrypt key size for inner
                                                    // wrapper

// Input Validation
    // to save typing
    attributes = in->objectPublic.publicArea.objectAttributes;
    // FixedTPM and fixedParent must be CLEAR
    if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, fixedTPM)
       || IS_ATTRIBUTE(attributes, TPMA_OBJECT, fixedParent))
        return TPM_RCS_ATTRIBUTES + RC_Import_objectPublic;

    // Get parent pointer
    parentObject = HandleToObject(in->parentHandle);

    if(!ObjectIsParent(parentObject))
        return TPM_RCS_TYPE + RC_Import_parentHandle;

    if(in->symmetricAlg.algorithm != TPM_ALG_NULL)
    {
        // Get inner wrap key size
        innerKeySize = in->symmetricAlg.keyBits.sym;
        // Input symmetric key must match the size of algorithm.
        if(in->encryptionKey.t.size != (innerKeySize + 7) / 8)
            return TPM_RCS_SIZE + RC_Import_encryptionKey;
    }
    else
    {
        // If input symmetric algorithm is NULL, input symmetric key size must
        // be 0 as well
        if(in->encryptionKey.t.size != 0)
            return TPM_RCS_SIZE + RC_Import_encryptionKey;
        // If encryptedDuplication is SET, then the object must have an inner
        // wrapper
        if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, encryptedDuplication))
            return TPM_RCS_ATTRIBUTES + RC_Import_encryptionKey;
    }
    // See if there is an outer wrapper
    if(in->inSymSeed.t.size != 0)
    {
        // in->inParentHandle is a parent, but in order to decrypt an outer wrapper,
        // it must be able to do key exchange and a symmetric key can't do that.
        if(parentObject->publicArea.type == TPM_ALG_SYMCIPHER)
            return TPM_RCS_TYPE + RC_Import_parentHandle;

        // Decrypt input secret data via asymmetric decryption. TPM_RC_ATTRIBUTES,
        // TPM_RC_ECC_POINT, TPM_RC_INSUFFICIENT, TPM_RC_KEY, TPM_RC_NO_RESULT,
        // TPM_RC_SIZE, TPM_RC_VALUE may be returned at this point
        result = CryptSecretDecrypt(parentObject, NULL, DUPLICATE_STRING,
                                    &in->inSymSeed, &data);
        pAssert(result != TPM_RC_BINDING);
        if(result != TPM_RC_SUCCESS)
            return RcSafeAddToResult(result, RC_Import_inSymSeed);
    }
    else
    {
```

```
        // If encrytpedDuplication is set, then the object must have an outer
        // wrapper
        if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, encryptedDuplication))
            return TPM_RCS_ATTRIBUTES + RC_Import_inSymSeed;
        data.t.size = 0;
    }
    // Compute name of object
    PublicMarshalAndComputeName(&(in->objectPublic.publicArea), &name);
    if(name.t.size == 0)
        return TPM_RCS_HASH + RC_Import_objectPublic;

    // Retrieve sensitive from private.
    // TPM_RC_INSUFFICIENT, TPM_RC_INTEGRITY, TPM_RC_SIZE may be returned here.
    result = DuplicateToSensitive(&in->duplicate.b, &name.b, parentObject,
                                  in->objectPublic.publicArea.nameAlg,
                                  &data.b, &in->symmetricAlg,
                                  &in->encryptionKey.b, &sensitive);
    if(result != TPM_RC_SUCCESS)
        return RcSafeAddToResult(result, RC_Import_duplicate);

    // If the parent of this object has fixedTPM SET, then validate this
    // object as if it were being loaded so that validation can be skipped
    // when it is actually loaded.
    if(IS_ATTRIBUTE(parentObject->publicArea.objectAttributes, TPMA_OBJECT, fixedTPM))
    {
        result = ObjectLoad(NULL, NULL, &in->objectPublic.publicArea,
                            &sensitive, RC_Import_objectPublic, RC_Import_duplicate,
                            NULL);
    }
// Command output
    if(result == TPM_RC_SUCCESS)
    {
        // Prepare output private data from sensitive
        SensitiveToPrivate(&sensitive, &name, parentObject,
                           in->objectPublic.publicArea.nameAlg,
                           &out->outPrivate);
    }
    return result;
}
#endif // CC_Import
```

14 Asymmetric Primitives

14.1 Introduction

The commands in this clause provide low-level primitives for access to the asymmetric algorithms implemented in the TPM. Many of these commands are only allowed if the asymmetric key is an unrestricted key.

14.2 TPM2_RSA_Encrypt

14.2.1 General Description

This command performs RSA encryption using the indicated padding scheme according to IETF RFC 8017. If the *scheme* of *keyHandle* is TPM_ALG_NULL, then the caller may use *inScheme* to specify the padding scheme. If *scheme* of keyHandle is not TPM_ALG_NULL, then *inScheme* shall either be TPM_ALG_NULL or be the same as *scheme* (TPM_RC_SCHEME).

The key referenced by *keyHandle* is required to be an RSA key (TPM_RC_KEY).

The three types of allowed padding are:

1) TPM_ALG_OAEP – Data is OAEP padded as described in 7.1 of IETF RFC 8017 (PKCS#1). The only supported mask generation is MGF1.

2) TPM_ALG_RSAES – Data is padded as described in 7.2 of IETF RFC 8017 (PKCS#1).

3) TPM_ALG_NULL – Data is not padded by the TPM and the TPM will treat *message* as an unsigned integer and perform a modular exponentiation of *message* using the public exponent of the key referenced by *keyHandle*. This scheme is only used if both the *scheme* in the key referenced by *keyHandle* is TPM_ALG_NULL, and the *inScheme* parameter of the command is TPM_ALG_NULL. The input value cannot be larger than the public modulus of the key referenced by *keyHandle*.

Table 43 — Padding Scheme Selection

keyHandle→scheme	***inScheme***	**padding scheme used**
TPM_ALG_NULL	TPM_ALG_NULL	none
	TPM_ALG_RSAES	RSAES
	TPM_ALG_OAEP	OAEP
TPM_ALG_RSAES	TPM_ALG_NULL	RSAES
	TPM_ALG_RSAES	RSAES
	TPM_ALG_OAEP	error (TPM_RC_SCHEME)
TPM_ALG_OAEP	TPM_ALG_NULL	OAEP
	TPM_ALG_RSAES	error (TPM_RC_SCHEME)
	TPM_ALG_OAEP	OAEP

After padding, the data is RSAEP encrypted according to 5.1.1 of IETF RFC 8017 (PKCS#1).

If *inScheme* is used, and the scheme requires a hash algorithm it may not be TPM_ALG_NULL.

NOTE 1 Because only the public portion of the key needs to be loaded for this command, the caller can manipulate the attributes of the key in any way desired. As a result, the TPM shall not check the consistency of the attributes. The only property checking is that the key is an RSA key and that the padding scheme is supported.

The *message* parameter is limited in size by the padding scheme according to the following table:

Table 44 — Message Size Limits Based on Padding

Scheme	Maximum Message Length (*mLen*) in Octets	Comments
TPM_ALG_OAEP	$mLen \leq k - 2hLen - 2$	
TPM_ALG_RSAES	$mLen \leq k - 11$	
TPM_ALG_NULL	$mLen \leq k$	The numeric value of the message must be less than the numeric value of the public modulus (n).
NOTES 1) k ≔ the number of byes in the public modulus 2) $hLen$ ≔ the number of octets in the digest produced by the hash algorithm used in the process		

The *label* parameter is optional. If provided (*label.size* != 0) then the TPM shall return TPM_RC_VALUE if the last octet in *label* is not zero. The terminating octet of zero is included in the *label* used in the padding scheme.

NOTE 2 If the scheme does not use a label, the TPM will still verify that label is properly formatted if label is present.

NOTE 3 Specifications before version 1.54 stated that *label* is truncated after the first zero octet. Applications should not include embedded zero bytes for compatibility.

The function returns padded and encrypted value *outData*.

The *message* parameter in the command may be encrypted using parameter encryption.

NOTE 4 Only the public area of *keyHandle* is required to be loaded. A public key may be loaded with any desired scheme. If the scheme is to be changed, a different public area must be loaded.

14.2.2 Command and Response

Table 45 — TPM2_RSA_Encrypt Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit, encrypt, or decrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_RSA_Encrypt
TPMI_DH_OBJECT	keyHandle	reference to public portion of RSA key to use for encryption Auth Index: None
TPM2B_PUBLIC_KEY_RSA	message	message to be encrypted NOTE 1 The data type was chosen because it limits the overall size of the input to no greater than the size of the largest RSA public key. This may be larger than allowed for *keyHandle*.
TPMT_RSA_DECRYPT+	inScheme	the padding scheme to use if *scheme* associated with *keyHandle* is TPM_ALG_NULL
TPM2B_DATA	label	optional label *L* to be associated with the message Size of the buffer is zero if no label is present NOTE 2 See description of label above.

Table 46 — TPM2_RSA_Encrypt Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_PUBLIC_KEY_RSA	outData	encrypted output

14.2.3 Detailed Actions

```
#include "Tpm.h"
#include "RSA_Encrypt_fp.h"
#if CC_RSA_Encrypt  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_ATTRIBUTES	*decrypt* attribute is not SET in key referenced by *keyHandle*
TPM_RC_KEY	*keyHandle* does not reference an RSA key
TPM_RC_SCHEME	incorrect input scheme, or the chosen scheme is not a valid RSA decrypt scheme
TPM_RC_VALUE	the numeric value of *message* is greater than the public modulus of the key referenced by *keyHandle*, or *label* is not a null-terminated string

```
TPM_RC
TPM2_RSA_Encrypt(
    RSA_Encrypt_In      *in,            // IN: input parameter list
    RSA_Encrypt_Out     *out            // OUT: output parameter list
    )
{
    TPM_RC                  result;
    OBJECT                  *rsaKey;
    TPMT_RSA_DECRYPT        *scheme;
// Input Validation
    rsaKey = HandleToObject(in->keyHandle);

    // selected key must be an RSA key
    if(rsaKey->publicArea.type != TPM_ALG_RSA)
        return TPM_RCS_KEY + RC_RSA_Encrypt_keyHandle;
    // selected key must have the decryption attribute
    if(!IS_ATTRIBUTE(rsaKey->publicArea.objectAttributes, TPMA_OBJECT, decrypt))
        return TPM_RCS_ATTRIBUTES + RC_RSA_Encrypt_keyHandle;

    // Is there a label?
    if(!IsLabelProperlyFormatted(&in->label.b))
        return TPM_RCS_VALUE + RC_RSA_Encrypt_label;
// Command Output
    // Select a scheme for encryption
    scheme = CryptRsaSelectScheme(in->keyHandle, &in->inScheme);
    if(scheme == NULL)
        return TPM_RCS_SCHEME + RC_RSA_Encrypt_inScheme;

    // Encryption.  TPM_RC_VALUE, or TPM_RC_SCHEME errors my be returned buy
    // CryptEncyptRSA.
    out->outData.t.size = sizeof(out->outData.t.buffer);

    result = CryptRsaEncrypt(&out->outData, &in->message.b, rsaKey, scheme,
                             &in->label.b, NULL);
    return result;
}
#endif // CC_RSA_Encrypt
```

14.3 TPM2_RSA_Decrypt

14.3.1 General Description

This command performs RSA decryption using the indicated padding scheme according to IETF RFC 8017 ((PKCS#1).

The scheme selection for this command is the same as for TPM2_RSA_Encrypt() and is shown in Table 43.

The key referenced by *keyHandle* shall be an RSA key (TPM_RC_KEY) with *restricted* CLEAR and *decrypt* SET (TPM_RC_ATTRIBUTES).

This command uses the private key of *keyHandle* for this operation and authorization is required.

The TPM will perform a modular exponentiation of ciphertext using the private exponent associated with *keyHandle* (this is described in IETF RFC 8017 (PKCS#1), clause 5.1.2). It will then validate the padding according to the selected scheme. If the padding checks fail, TPM_RC_VALUE is returned. Otherwise, the data is returned with the padding removed. If no padding is used, the returned value is an unsigned integer value that is the result of the modular exponentiation of *cipherText* using the private exponent of *keyHandle*. The returned value may include leading octets zeros so that it is the same size as the public modulus. For the other padding schemes, the returned value will be smaller than the public modulus but will contain all the data remaining after padding is removed and this may include leading zeros if the original encrypted value contained leading zeros.

If a label is used in the padding process of the scheme during encryption, the *label* parameter is required to be present in the decryption process and *label* is required to be the same in both cases. If label is not the same, the decrypt operation is very likely to fail ((TPM_RC_VALUE). If *label* is present (*label.size* != 0), it shall be a byte stream whose last byte is zero or the TPM will return TPM_RC_VALUE.

NOTE 1 The size of *label* includes the terminating null.

The *message* parameter in the response may be encrypted using parameter encryption.

If *inScheme* is used, and the scheme requires a hash algorithm it may not be TPM_ALG_NULL.

If the scheme does not require a label, the value in *label* is not used but the size of the label field is checked for consistency with the indicated data type (TPM2B_DATA). That is, the field may not be larger than allowed for a TPM2B_DATA.

14.3.2 Command and Response

Table 47 — TPM2_RSA_Decrypt Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_RSA_Decrypt
TPMI_DH_OBJECT	@keyHandle	RSA key to use for decryption Auth Index: 1 Auth Role: USER
TPM2B_PUBLIC_KEY_RSA	cipherText	cipher text to be decrypted NOTE An encrypted RSA data block is the size of the public modulus.
TPMT_RSA_DECRYPT+	inScheme	the padding scheme to use if *scheme* associated with *keyHandle* is TPM_ALG_NULL
TPM2B_DATA	label	label whose association with the message is to be verified

Table 48 — TPM2_RSA_Decrypt Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_PUBLIC_KEY_RSA	message	decrypted output

14.3.3 Detailed Actions

```
#include "Tpm.h"
#include "RSA_Decrypt_fp.h"
#if CC_RSA_Decrypt  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_ATTRIBUTES	*decrypt* is not SET or if *restricted* is SET in the key referenced by *keyHandle*
TPM_RC_BINDING	The public and private parts of the key are not properly bound
TPM_RC_KEY	*keyHandle* does not reference an unrestricted decrypt key
TPM_RC_SCHEME	incorrect input scheme, or the chosen *scheme* is not a valid RSA decrypt scheme
TPM_RC_SIZE	*cipherText* is not the size of the modulus of key referenced by *keyHandle*
TPM_RC_VALUE	*label* is not a null terminated string or the value of *cipherText* is greater that the modulus of *keyHandle* or the encoding of the data is not valid

```
TPM_RC
TPM2_RSA_Decrypt(
    RSA_Decrypt_In      *in,            // IN: input parameter list
    RSA_Decrypt_Out     *out            // OUT: output parameter list
    )
{
    TPM_RC                       result;
    OBJECT                      *rsaKey;
    TPMT_RSA_DECRYPT            *scheme;

// Input Validation

    rsaKey = HandleToObject(in->keyHandle);

    // The selected key must be an RSA key
    if(rsaKey->publicArea.type != TPM_ALG_RSA)
        return TPM_RCS_KEY + RC_RSA_Decrypt_keyHandle;

    // The selected key must be an unrestricted decryption key
    if(IS_ATTRIBUTE(rsaKey->publicArea.objectAttributes, TPMA_OBJECT, restricted)
       || !IS_ATTRIBUTE(rsaKey->publicArea.objectAttributes, TPMA_OBJECT, decrypt))
        return TPM_RCS_ATTRIBUTES + RC_RSA_Decrypt_keyHandle;

    // NOTE: Proper operation of this command requires that the sensitive area
    // of the key is loaded. This is assured because authorization is required
    // to use the sensitive area of the key. In order to check the authorization,
    // the sensitive area has to be loaded, even if authorization is with policy.

    // If label is present, make sure that it is a NULL-terminated string
    if(!IsLabelProperlyFormatted(&in->label.b))
        return TPM_RCS_VALUE + RC_RSA_Decrypt_label;
// Command Output
    // Select a scheme for decrypt.
    scheme = CryptRsaSelectScheme(in->keyHandle, &in->inScheme);
    if(scheme == NULL)
        return TPM_RCS_SCHEME + RC_RSA_Decrypt_inScheme;

    // Decryption.  TPM_RC_VALUE, TPM_RC_SIZE, and TPM_RC_KEY error may be
    // returned by CryptRsaDecrypt.
```



```
    // NOTE: CryptRsaDecrypt can also return TPM_RC_ATTRIBUTES or TPM_RC_BINDING
    // when the key is not a decryption key but that was checked above.
    out->message.t.size = sizeof(out->message.t.buffer);
    result = CryptRsaDecrypt(&out->message.b, &in->cipherText.b, rsaKey,
                             scheme, &in->label.b);
    return result;
}
#endif // CC_RSA_Decrypt
```

14.4 TPM2_ECDH_KeyGen

14.4.1 General Description

This command uses the TPM to generate an ephemeral key pair (d_e, Q_e where $Q_e \coloneqq [d_e]G$). It uses the private ephemeral key and a loaded public key (Q_S) to compute the shared secret value ($P \coloneqq [hd_e]Q_S$).

keyHandle shall refer to a loaded, ECC key (TPM_RC_KEY). The sensitive portion of this key need not be loaded.

The curve parameters of the loaded ECC key are used to generate the ephemeral key.

NOTE This function is the equivalent of encrypting data to another object's public key. The *seed* value is used in a KDF to generate a symmetric key and that key is used to encrypt the data. Once the data is encrypted and the symmetric key discarded, only the object with the private portion of the *keyHandle* will be able to decrypt it.

The *zPoint* in the response may be encrypted using parameter encryption.

14.4.2 Command and Response

Table 49 — TPM2_ECDH_KeyGen Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit or encrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_ECDH_KeyGen
TPMI_DH_OBJECT	keyHandle	Handle of a loaded ECC key public area. Auth Index: None

Table 50 — TPM2_ECDH_KeyGen Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_ECC_POINT	zPoint	results of $P \coloneqq h[d_e]Q_s$
TPM2B_ECC_POINT	pubPoint	generated ephemeral public point (Q_e)

14.4.3 Detailed Actions

```
#include "Tpm.h"
#include "ECDH_KeyGen_fp.h"
#if CC_ECDH_KeyGen  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_KEY	*keyHandle* does not reference an ECC key

```
TPM_RC
TPM2_ECDH_KeyGen(
    ECDH_KeyGen_In      *in,            // IN: input parameter list
    ECDH_KeyGen_Out     *out            // OUT: output parameter list
    )
{
    OBJECT                  *eccKey;
    TPM2B_ECC_PARAMETER      sensitive;
    TPM_RC                   result;

// Input Validation

    eccKey = HandleToObject(in->keyHandle);

    // Referenced key must be an ECC key
    if(eccKey->publicArea.type != TPM_ALG_ECC)
        return TPM_RCS_KEY + RC_ECDH_KeyGen_keyHandle;

// Command Output
    do
    {
        TPMT_PUBLIC         *keyPublic = &eccKey->publicArea;
        // Create ephemeral ECC key
        result = CryptEccNewKeyPair(&out->pubPoint.point, &sensitive,
                                    keyPublic->parameters.eccDetail.curveID);
        if(result == TPM_RC_SUCCESS)
        {
            // Compute Z
            result = CryptEccPointMultiply(&out->zPoint.point,
                                           keyPublic->parameters.eccDetail.curveID,
                                           &keyPublic->unique.ecc,
                                           &sensitive,
                                           NULL, NULL);
                    // The point in the key is not on the curve. Indicate
                    // that the key is bad.
            if(result == TPM_RC_ECC_POINT)
                return TPM_RCS_KEY + RC_ECDH_KeyGen_keyHandle;
             // The other possible error from CryptEccPointMultiply is
             // TPM_RC_NO_RESULT indicating that the multiplication resulted in
             // the point at infinity, so get a new random key and start over
             // BTW, this never happens.
        }
    } while(result == TPM_RC_NO_RESULT);
    return result;
}
#endif // CC_ECDH_KeyGen
```

14.5 TPM2_ECDH_ZGen

14.5.1 General Description

This command uses the TPM to recover the *Z* value from a public point (Q_B) and a private key (d_s). It will perform the multiplication of the provided *inPoint* (Q_B) with the private key (d_s) and return the coordinates of the resultant point ($Z = (x_Z, y_Z) \coloneqq [hd_s]Q_B$; where h is the cofactor of the curve).

keyHandle shall refer to a loaded, ECC key (TPM_RC_KEY) with the *restricted* attribute CLEAR and the *decrypt* attribute SET (TPM_RC_ATTRIBUTES).

NOTE While TPM_RC_ATTRIBUTES is preferred, TPM_RC_KEY is acceptable.

The *scheme* of the key referenced by *keyHandle* is required to be either TPM_ALG_ECDH or TPM_ALG_NULL (TPM_RC_SCHEME).

inPoint is required to be on the curve of the key referenced by *keyHandle* (TPM_RC_ECC_POINT).

The parameters of the key referenced by *keyHandle* are used to perform the point multiplication.

14.5.2 Command and Response

Table 51 — TPM2_ECDH_ZGen Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_ECDH_ZGen
TPMI_DH_OBJECT	@keyHandle	handle of a loaded ECC key Auth Index: 1 Auth Role: USER
TPM2B_ECC_POINT	inPoint	a public key

Table 52 — TPM2_ECDH_ZGen Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_ECC_POINT	outPoint	X and Y coordinates of the product of the multiplication $Z = (x_Z, y_Z) \coloneqq [hd_S]Q_B$

14.5.3 Detailed Actions

```
#include "Tpm.h"
#include "ECDH_ZGen_fp.h"
#if CC_ECDH_ZGen  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_ATTRIBUTES	key referenced by *keyA* is restricted or not a decrypt key
TPM_RC_KEY	key referenced by *keyA* is not an ECC key
TPM_RC_NO_RESULT	multiplying *inPoint* resulted in a point at infinity
TPM_RC_SCHEME	the scheme of the key referenced by *keyA* is not TPM_ALG_NULL, TPM_ALG_ECDH,

```
TPM_RC
TPM2_ECDH_ZGen(
    ECDH_ZGen_In    *in,            // IN: input parameter list
    ECDH_ZGen_Out   *out            // OUT: output parameter list
    )
{
    TPM_RC                   result;
    OBJECT                  *eccKey;

// Input Validation
    eccKey = HandleToObject(in->keyHandle);

    // Selected key must be a non-restricted, decrypt ECC key
    if(eccKey->publicArea.type != TPM_ALG_ECC)
        return TPM_RCS_KEY + RC_ECDH_ZGen_keyHandle;
    // Selected key needs to be unrestricted with the 'decrypt' attribute
    if(IS_ATTRIBUTE(eccKey->publicArea.objectAttributes, TPMA_OBJECT, restricted)
       || !IS_ATTRIBUTE(eccKey->publicArea.objectAttributes, TPMA_OBJECT, decrypt))
        return TPM_RCS_ATTRIBUTES + RC_ECDH_ZGen_keyHandle;
    // Make sure the scheme allows this use
    if(eccKey->publicArea.parameters.eccDetail.scheme.scheme != TPM_ALG_ECDH
       &&  eccKey->publicArea.parameters.eccDetail.scheme.scheme != TPM_ALG_NULL)
        return TPM_RCS_SCHEME + RC_ECDH_ZGen_keyHandle;
// Command Output
    // Compute Z. TPM_RC_ECC_POINT or TPM_RC_NO_RESULT may be returned here.
    result = CryptEccPointMultiply(&out->outPoint.point,
                                   eccKey->publicArea.parameters.eccDetail.curveID,
                                   &in->inPoint.point,
                                   &eccKey->sensitive.sensitive.ecc,
                                   NULL, NULL);
    if(result != TPM_RC_SUCCESS)
        return RcSafeAddToResult(result, RC_ECDH_ZGen_inPoint);
    return result;
}
#endif // CC_ECDH_ZGen
```


14.6 TPM2_ECC_Parameters

14.6.1 General Description

This command returns the parameters of an ECC curve identified by its TCG-assigned *curveID*.

The value returned is the same as that from the TCG Algorithm Registry, but may not be the same size.

EXAMPLE The value 01 may be returned as 00000001.

14.6.2 Command and Response

Table 53 — TPM2_ECC_Parameters Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_ECC_Parameters
TPMI_ECC_CURVE	curveID	parameter set selector

Table 54 — TPM2_ECC_Parameters Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPMS_ALGORITHM_DETAIL_ECC	parameters	ECC parameters for the selected curve

14.6.3 Detailed Actions

```
#include "Tpm.h"
#include "ECC_Parameters_fp.h"
#if CC_ECC_Parameters  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_VALUE	Unsupported ECC curve ID

```
TPM_RC
TPM2_ECC_Parameters(
    ECC_Parameters_In    *in,            // IN: input parameter list
    ECC_Parameters_Out   *out            // OUT: output parameter list
    )
{
// Command Output

    // Get ECC curve parameters
    if(CryptEccGetParameters(in->curveID, &out->parameters))
        return TPM_RC_SUCCESS;
    else
        return TPM_RCS_VALUE + RC_ECC_Parameters_curveID;
}
#endif // CC_ECC_Parameters
```

14.7 TPM2_ZGen_2Phase

14.7.1 General Description

This command supports two-phase key exchange protocols. The command is used in combination with TPM2_EC_Ephemeral(). TPM2_EC_Ephemeral() generates an ephemeral key and returns the public point of that ephemeral key along with a numeric value that allows the TPM to regenerate the associated private key.

The input parameters for this command are a static public key (*inQsU*), an ephemeral key (*inQeU*) from party B, and the *commitCounter* returned by TPM2_EC_Ephemeral(). The TPM uses the counter value to regenerate the ephemeral private key ($d_{e,V}$) and the associated public key ($Q_{e,V}$). *keyA* provides the static ephemeral elements $d_{s,V}$ and $Q_{s,V}$. This provides the two pairs of ephemeral and static keys that are required for the schemes supported by this command.

The TPM will compute $Z\ or\ Z_s$ and Z_e according to the selected scheme. If the scheme is not a two-phase key exchange scheme or if the scheme is not supported, the TPM will return TPM_RC_SCHEME.

It is an error if *inQsB* or *inQeB* are not on the curve of *keyA* (TPM_RC_ECC_POINT).

The two-phase key schemes that were assigned an algorithm ID as of the time of the publication of this specification are TPM_ALG_ECDH, TPM_ALG_ECMQV, and TPM_ALG_SM2.

If this command is supported, then support for TPM_ALG_ECDH is required. Support for TPM_ALG_ECMQV or TPM_ALG_SM2 is optional.

NOTE 1 If SM2 is supported and this command is supported, then the implementation is required to support the key exchange protocol of SM2, part 3.

For TPM_ALG_ECDH *outZ1* will be Zs and *outZ2* will Ze as defined in 6.1.1.2 of SP800-56A.

NOTE 2 An unrestricted decryption key using ECDH may be used in either TPM2_ECDH_ZGen() or TPM2_ZGen_2Phase as the computation done with the private part of *keyA* is the same in both cases.

For TPM_ALG_ECMQV or TPM_ALG_SM2 *outZ1* will be Z and *outZ2* will be an Empty Point.

NOTE 3 An Empty Point has two Empty Buffers as coordinates meaning the minimum *size* value for *outZ2* will be four.

If the input scheme is TPM_ALG_ECDH, then *outZ1* will be Z_s and outZ2 will be Z_e. For schemes like MQV (including SM2), *outZ1* will contain the computed value and *outZ2* will be an Empty Point.

NOTE 4 The Z values returned by the TPM are a full point and not just an x-coordinate.

If a computation of either Z produces the point at infinity, then the corresponding Z value will be an Empty Point.

14.7.2 Command and Response

Table 55 — TPM2_ZGen_2Phase Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_ZGen_2Phase
TPMI_DH_OBJECT	@keyA	handle of an unrestricted decryption key ECC The private key referenced by this handle is used as $d_{S,A}$ Auth Index: 1 Auth Role: USER
TPM2B_ECC_POINT	inQsB	other party's static public key ($Q_{s,B} = (X_{s,B}, Y_{s,B})$)
TPM2B_ECC_POINT	inQeB	other party's ephemeral public key ($Q_{e,B} = (X_{e,B}, Y_{e,B})$)
TPMI_ECC_KEY_EXCHANGE	inScheme	the key exchange scheme
UINT16	counter	value returned by TPM2_EC_Ephemeral()

Table 56 — TPM2_ZGen_2Phase Response

Type	Name	Description
TPM_ST	tag	
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_ECC_POINT	outZ1	X and Y coordinates of the computed value (scheme dependent)
TPM2B_ECC_POINT	outZ2	X and Y coordinates of the second computed value (scheme dependent)

14.7.3 Detailed Actions

```
#include "Tpm.h"
#include "ZGen_2Phase_fp.h"
#if CC_ZGen_2Phase  // Conditional expansion of this file
```

This command uses the TPM to recover one or two Z values in a two phase key exchange protocol

Error Returns	Meaning
TPM_RC_ATTRIBUTES	key referenced by *keyA* is restricted or not a decrypt key
TPM_RC_ECC_POINT	*inQsB* or *inQeB* is not on the curve of the key reference by *keyA*
TPM_RC_KEY	key referenced by *keyA* is not an ECC key
TPM_RC_SCHEME	the scheme of the key referenced by *keyA* is not TPM_ALG_NULL, TPM_ALG_ECDH, ALG_ECMQV or TPM_ALG_SM2

```
TPM_RC
TPM2_ZGen_2Phase(
    ZGen_2Phase_In      *in,            // IN: input parameter list
    ZGen_2Phase_Out     *out            // OUT: output parameter list
    )
{
    TPM_RC                   result;
    OBJECT                  *eccKey;
    TPM2B_ECC_PARAMETER      r;
    TPM_ALG_ID               scheme;

// Input Validation

    eccKey = HandleToObject(in->keyA);

    // keyA must be an ECC key
    if(eccKey->publicArea.type != TPM_ALG_ECC)
        return TPM_RCS_KEY + RC_ZGen_2Phase_keyA;

    // keyA must not be restricted and must be a decrypt key
    if(IS_ATTRIBUTE(eccKey->publicArea.objectAttributes, TPMA_OBJECT, restricted)
       || !IS_ATTRIBUTE(eccKey->publicArea.objectAttributes, TPMA_OBJECT, decrypt))
        return TPM_RCS_ATTRIBUTES + RC_ZGen_2Phase_keyA;

    // if the scheme of keyA is TPM_ALG_NULL, then use the input scheme; otherwise
    // the input scheme must be the same as the scheme of keyA
    scheme = eccKey->publicArea.parameters.asymDetail.scheme.scheme;
    if(scheme != TPM_ALG_NULL)
    {
        if(scheme != in->inScheme)
            return TPM_RCS_SCHEME + RC_ZGen_2Phase_inScheme;
    }
    else
        scheme = in->inScheme;
    if(scheme == TPM_ALG_NULL)
        return TPM_RCS_SCHEME + RC_ZGen_2Phase_inScheme;

    // Input points must be on the curve of keyA
    if(!CryptEccIsPointOnCurve(eccKey->publicArea.parameters.eccDetail.curveID,
                               &in->inQsB.point))
        return TPM_RCS_ECC_POINT + RC_ZGen_2Phase_inQsB;

    if(!CryptEccIsPointOnCurve(eccKey->publicArea.parameters.eccDetail.curveID,
                               &in->inQeB.point))
        return TPM_RCS_ECC_POINT + RC_ZGen_2Phase_inQeB;
```



```

    if(!CryptGenerateR(&r, &in->counter,
                       eccKey->publicArea.parameters.eccDetail.curveID,
                       NULL))
        return TPM_RCS_VALUE + RC_ZGen_2Phase_counter;

// Command Output

    result =
        CryptEcc2PhaseKeyExchange(&out->outZ1.point,
                                  &out->outZ2.point,
                                  eccKey->publicArea.parameters.eccDetail.curveID,
                                  scheme,
                                  &eccKey->sensitive.sensitive.ecc,
                                  &r,
                                  &in->inQsB.point,
                                  &in->inQeB.point);
    if(result == TPM_RC_SCHEME)
        return TPM_RCS_SCHEME + RC_ZGen_2Phase_inScheme;

    if(result == TPM_RC_SUCCESS)
        CryptEndCommit(in->counter);

    return result;
}
#endif // CC_ZGen_2Phase
```

15 Symmetric Primitives

15.1 Introduction

The commands in this clause provide low-level primitives for access to the symmetric algorithms implemented in the TPM that operate on blocks of data. These include symmetric encryption and decryption as well as hash and HMAC. All of the commands in this group are stateless. That is, they have no persistent state that is retained in the TPM when the command is complete.

For hashing, HMAC, and Events that require large blocks of data with retained state, the sequence commands are provided (see clause 1).

Some of the symmetric encryption/decryption modes use an IV. When an IV is used, it may be an initiation value or a chained value from a previous stage. The chaining for each mode is:

Table 57 — Symmetric Chaining Process

Mode	**Chaining process**
TPM_ALG_CTR	The TPM will increment the entire IV provided by the caller. The next count value will be returned to the caller as *ivOut*. This can be the input value to the next encrypt or decrypt operation. *ivIn* is required to be the size of a block encrypted by the selected algorithm and key combination. If the size of *ivIn* is not correct, the TPM shall return TPM_RC_SIZE. EXAMPLE 1 AES requires that *ivIn* be 128 bits (16 octets). *ivOut* will be the size of a cipher block and not the size of the last encrypted block. NOTE *ivOut* will be the value of the counter after the last block is encrypted. EXAMPLE 2 If *ivIn* were 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00_{16} and four data blocks were encrypted, *ivOut* will have a value of 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 04_{16}. All the bits of the IV are incremented as if it were an unsigned integer.
TPM_ALG_OFB	In Output Feedback (OFB), the output of the pseudo-random function (the block encryption algorithm) is XORed with a plaintext block to produce a ciphertext block. *ivOut* will be the value that was XORed with the last plaintext block. That value can be used as the *ivIn* for a next buffer. *ivIn* is required to be the size of a block encrypted by the selected algorithm and key combination. If the size of *ivIn* is not correct, the TPM shall return TPM_RC_SIZE. *ivOut* will be the size of a cipher block and not the size of the last encrypted block.
TPM_ALG_CBC	For Cipher Block Chaining (CBC), a block of ciphertext is XORed with the next plaintext block and that block is encrypted. The encrypted block is then input to the encryption of the next block. The last ciphertext block then is used as an IV for the next buffer. Even though the last ciphertext block is evident in the encrypted data, it is also returned in *ivOut*. *ivIn* is required to be the size of a block encrypted by the selected algorithm and key combination. If the size of *ivIn* is not correct, the TPM shall return TPM_RC_SIZE. *inData* is required to be an even multiple of the block encrypted by the selected algorithm and key combination. If the size of *inData* is not correct, the TPM shall return TPM_RC_SIZE.
TPM_ALG_CFB	Similar to CBC in that the last ciphertext block is an input to the encryption of the next block. *ivOut* will be the value that was XORed with the last plaintext block. That value can be used as the *ivIn* for a next buffer. *ivIn* is required to be the size of a block encrypted by the selected algorithm and key combination. If the size of *ivIn* is not correct, the TPM shall return TPM_RC_SIZE. *ivOut* will be the size of a cipher block and not the size of the last encrypted block.
TPM_ALG_ECB	Electronic Codebook (ECB) has no chaining. Each block of plaintext is encrypted using the key. ECB does not support chaining and *ivIn* shall be the Empty Buffer. *ivOut* will be the Empty Buffer. *inData* is required to be an even multiple of the block encrypted by the selected algorithm and key combination. If the size of *inData* is not correct, the TPM shall return TPM_RC_SIZE.

15.2 TPM2_EncryptDecrypt

15.2.1 General Description

NOTE 1 This command is deprecated, and TPM2_EncryptDecrypt2() is preferred. This should be reflected in platform-specific specifications.

This command performs symmetric encryption or decryption using the symmetric key referenced by keyHandle and the selected mode.

keyHandle shall reference a symmetric cipher object (TPM_RC_KEY) with the *restricted* attribute CLEAR (TPM_RC_ATTRIBUTES).

If the *decrypt* parameter of the command is TRUE, then the *decrypt* attribute of the key is required to be SET (TPM_RC_ATTRIBUTES). If the *decrypt* parameter of the command is FALSE, then the *sign* attribute of the key is required to be SET (TPM_RC_ATTRIBUTES).

NOTE 2 A key may have both *decrypt* and *sign* SET.

If the mode of the key is not TPM_ALG_NULL, then that is the only mode that can be used with the key and the caller is required to set *mode* either to TPM_ALG_NULL or to the same mode as the key (TPM_RC_MODE). If the mode of the key is TPM_ALG_NULL, then the caller may set *mode* to any valid symmetric encryption/decryption mode but may not select TPM_ALG_NULL (TPM_RC_MODE).

If the TPM allows this command to be canceled before completion, then the TPM may produce incremental results and return TPM_RC_SUCCESS rather than TPM_RC_CANCELED. In such case, *outData* may be less than *inData*.

NOTE 3 If all the data is encrypted/decrypted, the size of outData will be the same as inData.

15.2.2 Command and Response

Table 58 — TPM2_EncryptDecrypt Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_EncryptDecrypt
TPMI_DH_OBJECT	@keyHandle	the symmetric key used for the operation Auth Index: 1 Auth Role: USER
TPMI_YES_NO	decrypt	if YES, then the operation is decryption; if NO, the operation is encryption
TPMI_ALG_CIPHER_MODE+	mode	symmetric encryption/decryption mode this field shall match the default mode of the key or be TPM_ALG_NULL.
TPM2B_IV	ivIn	an initial value as required by the algorithm
TPM2B_MAX_BUFFER	inData	the data to be encrypted/decrypted

Table 59 — TPM2_EncryptDecrypt Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_MAX_BUFFER	outData	encrypted or decrypted output
TPM2B_IV	ivOut	chaining value to use for IV in next round

15.2.3 Detailed Actions

```
#include "Tpm.h"
#include "EncryptDecrypt_fp.h"
#if CC_EncryptDecrypt2
#include  "EncryptDecrypt_spt_fp.h"
#endif
#if CC_EncryptDecrypt  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_KEY	is not a symmetric decryption key with both public and private portions loaded
TPM_RC_SIZE	*IvIn* size is incompatible with the block cipher mode; or *inData* size is not an even multiple of the block size for CBC or ECB mode
TPM_RC_VALUE	*keyHandle* is restricted and the argument *mode* does not match the key's mode

```
TPM_RC
TPM2_EncryptDecrypt(
    EncryptDecrypt_In   *in,            // IN: input parameter list
    EncryptDecrypt_Out  *out            // OUT: output parameter list
    )
{
#if CC_EncryptDecrypt2
    return EncryptDecryptShared(in->keyHandle, in->decrypt, in->mode,
                                &in->ivIn, &in->inData, out);
#else
    OBJECT              *symKey;
    UINT16               keySize;
    UINT16               blockSize;
    BYTE                *key;
    TPM_ALG_ID           alg;
    TPM_ALG_ID           mode;
    TPM_RC               result;
    BOOL                 OK;
    TPMA_OBJECT          attributes;

// Input Validation
    symKey = HandleToObject(in->keyHandle);
    mode = symKey->publicArea.parameters.symDetail.sym.mode.sym;
    attributes = symKey->publicArea.objectAttributes;

    // The input key should be a symmetric key
    if(symKey->publicArea.type != TPM_ALG_SYMCIPHER)
        return TPM_RCS_KEY + RC_EncryptDecrypt_keyHandle;
    // The key must be unrestricted and allow the selected operation
    OK = IS_ATTRIBUTE(attributes, TPMA_OBJECT, restricted)
    if(YES == in->decrypt)
        OK = OK && IS_ATTRIBUTE(attributes, TPMA_OBJECT, decrypt);
    else
        OK = OK && IS_ATTRIBUTE(attributes, TPMA_OBJECT, sign);
    if(!OK)
        return TPM_RCS_ATTRIBUTES + RC_EncryptDecrypt_keyHandle;

    // If the key mode is not TPM_ALG_NULL...
    // or TPM_ALG_NULL
    if(mode != TPM_ALG_NULL)
    {
        // then the input mode has to be TPM_ALG_NULL or the same as the key
        if((in->mode != TPM_ALG_NULL) && (in->mode != mode))
```

```
            return TPM_RCS_MODE + RC_EncryptDecrypt_mode;
    }
    else
    {
        // if the key mode is null, then the input can't be null
        if(in->mode == TPM_ALG_NULL)
            return TPM_RCS_MODE + RC_EncryptDecrypt_mode;
        mode = in->mode;
    }
    // The input iv for ECB mode should be an Empty Buffer.  All the other modes
    // should have an iv size same as encryption block size
    keySize = symKey->publicArea.parameters.symDetail.sym.keyBits.sym;
    alg = symKey->publicArea.parameters.symDetail.sym.algorithm;
    blockSize = CryptGetSymmetricBlockSize(alg, keySize);

    // reverify the algorithm. This is mainly to keep static analysis tools happy
    if(blockSize == 0)
        return TPM_RCS_KEY + RC_EncryptDecrypt_keyHandle;

    // Note: When an algorithm is not supported by a TPM, the TPM_ALG_xxx for that
    // algorithm is not defined. However, it is assumed that the ALG_xxx_VALUE for
    // the algorithm is always defined. Both have the same numeric value.
    // ALG_xxx_VALUE is used here so that the code does not get cluttered with
    // #ifdef's. Having this check does not mean that the algorithm is supported.
    // If it was not supported the unmarshaling code would have rejected it before
    // this function were called. This means that, depending on the implementation,
    // the check could be redundant but it doesn't hurt.
    if(((mode == ALG_ECB_VALUE) && (in->ivIn.t.size != 0))
       || ((mode != ALG_ECB_VALUE) && (in->ivIn.t.size != blockSize)))
        return TPM_RCS_SIZE + RC_EncryptDecrypt_ivIn;

    // The input data size of CBC mode or ECB mode must be an even multiple of
    // the symmetric algorithm's block size
    if(((mode == ALG_CBC_VALUE) || (mode == ALG_ECB_VALUE))
       && ((in->inData.t.size % blockSize) != 0))
        return TPM_RCS_SIZE + RC_EncryptDecrypt_inData;

    // Copy IV
    // Note: This is copied here so that the calls to the encrypt/decrypt functions
    // will modify the output buffer, not the input buffer
    out->ivOut = in->ivIn;

// Command Output
    key = symKey->sensitive.sensitive.sym.t.buffer;
    // For symmetric encryption, the cipher data size is the same as plain data
    // size.
    out->outData.t.size = in->inData.t.size;
    if(in->decrypt == YES)
    {
        // Decrypt data to output
        result = CryptSymmetricDecrypt(out->outData.t.buffer, alg, keySize, key,
                                       &(out->ivOut), mode, in->inData.t.size,
                                       in->inData.t.buffer);
    }
    else
    {
        // Encrypt data to output
        result = CryptSymmetricEncrypt(out->outData.t.buffer, alg, keySize, key,
                                       &(out->ivOut), mode, in->inData.t.size,
                                       in->inData.t.buffer);
    }
    return result;
#endif // CC_EncryptDecrypt2

}
#endif // CC_EncryptDecrypt
```


15.3 TPM2_EncryptDecrypt2

15.3.1 General Description

This command is identical to TPM2_EncryptDecrypt(), except that the *inData* parameter is the first parameter. This permits *inData* to be parameter encrypted.

NOTE In platform specification updates, this command is preferred and TPM2_EncryptDecrypt() should be deprecated.

15.3.2 Comand and Response

Table 60 — TPM2_EncryptDecrypt2 Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_EncryptDecrypt2
TPMI_DH_OBJECT	@keyHandle	the symmetric key used for the operation Auth Index: 1 Auth Role: USER
TPM2B_MAX_BUFFER	inData	the data to be encrypted/decrypted
TPMI_YES_NO	decrypt	if YES, then the operation is decryption; if NO, the operation is encryption
TPMI_ALG_CIPHER_MODE+	mode	symmetric mode this field shall match the default mode of the key or be TPM_ALG_NULL.
TPM2B_IV	ivIn	an initial value as required by the algorithm

Table 61 — TPM2_EncryptDecrypt2 Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_MAX_BUFFER	outData	encrypted or decrypted output
TPM2B_IV	ivOut	chaining value to use for IV in next round

15.3.3 Detailed Actions

```
#include "Tpm.h"
#include "EncryptDecrypt2_fp.h"
#include "EncryptDecrypt_fp.h"
#include "EncryptDecrypt_spt_fp.h"
#if CC_EncryptDecrypt2  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_KEY	is not a symmetric decryption key with both public and private portions loaded
TPM_RC_SIZE	*IvIn* size is incompatible with the block cipher mode; or *inData* size is not an even multiple of the block size for CBC or ECB mode
TPM_RC_VALUE	*keyHandle* is restricted and the argument *mode* does not match the key's mode

```
TPM_RC
TPM2_EncryptDecrypt2(
    EncryptDecrypt2_In   *in,            // IN: input parameter list
    EncryptDecrypt2_Out  *out            // OUT: output parameter list
    )
{
    TPM_RC                result;
    // EncryptDecyrptShared() performs the operations as shown in
    // TPM2_EncrypDecrypt
    result = EncryptDecryptShared(in->keyHandle, in->decrypt, in->mode,
                                  &in->ivIn, &in->inData,
                                  (EncryptDecrypt_Out *)out);
    // Handle response code swizzle.
    switch(result)
    {
        case TPM_RCS_MODE + RC_EncryptDecrypt_mode:
            result = TPM_RCS_MODE + RC_EncryptDecrypt2_mode;
            break;
        case TPM_RCS_SIZE + RC_EncryptDecrypt_ivIn:
            result = TPM_RCS_SIZE + RC_EncryptDecrypt2_ivIn;
            break;
        case TPM_RCS_SIZE + RC_EncryptDecrypt_inData:
            result = TPM_RCS_SIZE + RC_EncryptDecrypt2_inData;
            break;
        default:
            break;
    }
    return result;
}
#endif // CC_EncryptDecrypt2
```

15.4 TPM2_Hash

15.4.1 General Description

This command performs a hash operation on a data buffer and returns the results.

NOTE If the data buffer to be hashed is larger than will fit into the TPM's input buffer, then the sequence hash commands will need to be used.

If the results of the hash will be used in a signing operation that uses a restricted signing key, then the ticket returned by this command can indicate that the hash is safe to sign.

If the digest is not safe to sign, then the TPM will return a TPMT_TK_HASHCHECK with the hierarchy set to TPM_RH_NULL and *digest* set to the Empty Buffer.

If *hierarchy* is TPM_RH_NULL, then *digest* in the ticket will be the Empty Buffer.

15.4.2 Command and Response

Table 62 — TPM2_Hash Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit, decrypt, or encrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_Hash
TPM2B_MAX_BUFFER	data	data to be hashed
TPMI_ALG_HASH	hashAlg	algorithm for the hash being computed – shall not be TPM_ALG_NULL
TPMI_RH_HIERARCHY+	hierarchy	hierarchy to use for the ticket (TPM_RH_NULL allowed)

Table 63 — TPM2_Hash Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_DIGEST	outHash	results
TPMT_TK_HASHCHECK	validation	ticket indicating that the sequence of octets used to compute *outDigest* did not start with TPM_GENERATED_VALUE will be a NULL ticket if the digest may not be signed with a restricted key

15.4.3 Detailed Actions

```
#include "Tpm.h"
#include "Hash_fp.h"
#if CC_Hash  // Conditional expansion of this file
TPM_RC
TPM2_Hash(
    Hash_In         *in,            // IN: input parameter list
    Hash_Out        *out            // OUT: output parameter list
    )
{
    HASH_STATE          hashState;

// Command Output

    // Output hash
        // Start hash stack
    out->outHash.t.size = CryptHashStart(&hashState, in->hashAlg);
        // Adding hash data
    CryptDigestUpdate2B(&hashState, &in->data.b);
        // Complete hash
    CryptHashEnd2B(&hashState, &out->outHash.b);

    // Output ticket
    out->validation.tag = TPM_ST_HASHCHECK;
    out->validation.hierarchy = in->hierarchy;

    if(in->hierarchy == TPM_RH_NULL)
    {
        // Ticket is not required
        out->validation.hierarchy = TPM_RH_NULL;
        out->validation.digest.t.size = 0;
    }
    else if(in->data.t.size >= sizeof(TPM_GENERATED)
            && !TicketIsSafe(&in->data.b))
    {
        // Ticket is not safe
        out->validation.hierarchy = TPM_RH_NULL;
        out->validation.digest.t.size = 0;
    }
    else
    {
        // Compute ticket
        TicketComputeHashCheck(in->hierarchy, in->hashAlg,
                               &out->outHash, &out->validation);
    }

    return TPM_RC_SUCCESS;
}
#endif // CC_Hash
```

15.5 TPM2_HMAC

15.5.1 General Description

This command performs an HMAC on the supplied data using the indicated hash algorithm.

NOTE 1 A TPM may implement either TPM2_HMAC() or TPM2_MAC() but not both, as they have the same command code and there is no way to distinguish them. A TPM that supports TPM2_MAC() will support any code that was written to use TPM2_HMAC(), but a TPM that supports TPM2_HMAC() will not support a MAC based on symmetric block ciphers.

The caller shall provide proper authorization for use of *handle*.

If the *sign* attribute is not SET in the key referenced by *handle* then the TPM shall return TPM_RC_KEY. If the key type is not TPM_ALG_KEYEDHASH then the TPM shall return TPM_RC_TYPE. If the key referenced by *handle* has the *restricted* attribute SET, the TPM shall return TPM_RC_ATTRIBUTES.

NOTE 2 For symmetric signing with a restricted key, see TPM2_Sign.

If the default scheme of the key referenced by *handle* is not TPM_ALG_NULL, then the *hashAlg* parameter is required to be either the same as the key's default or TPM_ALG_NULL (TPM_RC_VALUE). If the default scheme of the key is TPM_ALG_NULL, then hashAlg is required to be a valid hash and not TPM_ALG_NULL (TPM_RC_VALUE) (see hash selection matrix in

Table 72).

NOTE 3 A key may only have both sign and decrypt SET if the key is unrestricted. When both sign and decrypt are set, there is no default scheme for the key and the hash algorithm must be specified.

15.5.2 Command and Response

Table 64 — TPM2_HMAC Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_HMAC
TPMI_DH_OBJECT	@handle	handle for the symmetric signing key providing the HMAC key Auth Index: 1 Auth Role: USER
TPM2B_MAX_BUFFER	buffer	HMAC data
TPMI_ALG_HASH+	hashAlg	algorithm to use for HMAC

Table 65 — TPM2_HMAC Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_DIGEST	outHMAC	the returned HMAC in a sized buffer

15.5.3 Detailed Actions

```
#include "Tpm.h"
#include "HMAC_fp.h"
#if CC_HMAC  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_ATTRIBUTES	key referenced by *handle* is a restricted key
TPM_RC_KEY	*handle* does not reference a signing key
TPM_RC_TYPE	key referenced by *handle* is not an HMAC key
TPM_RC_VALUE	*hashAlg* is not compatible with the hash algorithm of the scheme of the object referenced by *handle*

```
TPM_RC
TPM2_HMAC(
    HMAC_In         *in,            // IN: input parameter list
    HMAC_Out        *out            // OUT: output parameter list
    )
{
    HMAC_STATE                   hmacState;
    OBJECT                      *hmacObject;
    TPMI_ALG_HASH                hashAlg;
    TPMT_PUBLIC                 *publicArea;

// Input Validation

    // Get HMAC key object and public area pointers
    hmacObject = HandleToObject(in->handle);
    publicArea = &hmacObject->publicArea;
    // Make sure that the key is an HMAC key
    if(publicArea->type != TPM_ALG_KEYEDHASH)
        return TPM_RCS_TYPE + RC_HMAC_handle;

    // and that it is unrestricted
    if(IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, restricted))
        return TPM_RCS_ATTRIBUTES + RC_HMAC_handle;

    // and that it is a signing key
    if(!IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, sign))
        return TPM_RCS_KEY + RC_HMAC_handle;

    // See if the key has a default
    if(publicArea->parameters.keyedHashDetail.scheme.scheme == TPM_ALG_NULL)
        // it doesn't so use the input value
        hashAlg = in->hashAlg;
    else
    {
        // key has a default so use it
        hashAlg
            = publicArea->parameters.keyedHashDetail.scheme.details.hmac.hashAlg;
        // and verify that the input was either the  TPM_ALG_NULL or the default
        if(in->hashAlg != TPM_ALG_NULL && in->hashAlg != hashAlg)
            hashAlg = TPM_ALG_NULL;
    }
    // if we ended up without a hash algorithm then return an error
    if(hashAlg == TPM_ALG_NULL)
        return TPM_RCS_VALUE + RC_HMAC_hashAlg;

// Command Output

```

```
    // Start HMAC stack
    out->outHMAC.t.size = CryptHmacStart2B(&hmacState, hashAlg,
                                           &hmacObject->sensitive.sensitive.bits.b);
    // Adding HMAC data
    CryptDigestUpdate2B(&hmacState.hashState, &in->buffer.b);

    // Complete HMAC
    CryptHmacEnd2B(&hmacState, &out->outHMAC.b);

    return TPM_RC_SUCCESS;
}
#endif // CC_HMAC
```

15.6 TPM2_MAC

15.6.1 General Description

This command performs an HMAC or a block cipher MAC on the supplied data using the indicated algorithm.

NOTE 1 A TPM may implement either TPM2_HMAC() or TPM2_MAC() but not both as they have the same command code and there is no way to distinguish them. A TPM that supports TPM2_MAC() will support any code that was written to use TPM2_HMAC() but a TPM that supports TPM2_HMAC () will not support a MAC based on symmetric block ciphers.

The caller shall provide proper authorization for use of *handle*.

If the *sign* attribute is not SET in the key referenced by *handle* then the TPM shall return TPM_RC_KEY. If the key type is neither TPM_ALG_KEYEDHASH nor TPM_ALG_SYMCIPHER then the TPM shall return TPM_RC_TYPE. If the key referenced by *handle* has the *restricted* attribute SET, the TPM shall return TPM_RC_ATTRIBUTES.

NOTE 2 For symmetric signing with a restricted key, see TPM2_Sign.

If the default scheme or mode of the key referenced by *handle* is not TPM_ALG_NULL, then the *inScheme* parameter is required to be either the same as the key's default or TPM_ALG_NULL (TPM_RC_VALUE).

If the default scheme of an HMAC key is TPM_ALG_NULL, then *inScheme* is required to be a valid hash and not TPM_ALG_NULL (TPM_RC_VALUE) (see algorithm selection matrix in

Table 75).

If the default mode of a symmetric cipher key is TPM_ALG_NULL, then *inScheme* is required to be a valid block cipher mode for authentication and not TPM_ALG_NULL (TPM_RC_VALUE)

NOTE 3 A key may only have both sign and decrypt SET if the key is unrestricted. When both sign and decrypt are set, there is no default scheme for the key and *inScheme* may not be TPM_ALG_NULL.

NOTE 4 TPM2_MAC() was added in revision 01.43.

15.6.2 Command and Response

Table 66 — TPM2_MAC Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_MAC
TPMI_DH_OBJECT	@handle	handle for the symmetric signing key providing the MAC key Auth Index: 1 Auth Role: USER
TPM2B_MAX_BUFFER	buffer	MAC data
TPMI_ALG_MAC_SCHEME+	inScheme	algorithm to use for MAC

Table 67 — TPM2_MAC Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_DIGEST	outMAC	the returned MAC in a sized buffer

15.6.3 Detailed Actions

```
#include "Tpm.h"
#include "MAC_fp.h"
#if CC_MAC  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_ATTRIBUTES	key referenced by *handle* is a restricted key
TPM_RC_KEY	*handle* does not reference a signing key
TPM_RC_TYPE	key referenced by *handle* is not an HMAC key
TPM_RC_VALUE	*hashAlg* is not compatible with the hash algorithm of the scheme of the object referenced by *handle*

```
TPM_RC
TPM2_MAC(
    MAC_In          *in,            // IN: input parameter list
    MAC_Out         *out            // OUT: output parameter list
    )
{
    OBJECT                  *keyObject;
    HMAC_STATE               state;
    TPMT_PUBLIC             *publicArea;
    TPM_RC                   result;

// Input Validation
    // Get MAC key object and public area pointers
    keyObject = HandleToObject(in->handle);
    publicArea = &keyObject->publicArea;

    // If the key is not able to do a MAC, indicate that the handle selects an
    // object that can't do a MAC
    result = CryptSelectMac(publicArea, &in->inScheme);
    if(result == TPM_RCS_TYPE)
        return TPM_RCS_TYPE + RC_MAC_handle;
    // If there is another error type, indicate that the scheme and key are not
    // compatible
    if(result != TPM_RC_SUCCESS)
        return RcSafeAddToResult(result, RC_MAC_inScheme);
    // Make sure that the key is not restricted
    if(IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, restricted))
        return TPM_RCS_ATTRIBUTES + RC_MAC_handle;
    // and that it is a signing key
    if(!IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, sign))
        return TPM_RCS_KEY + RC_MAC_handle;
// Command Output
    out->outMAC.t.size = CryptMacStart(&state, &publicArea->parameters,
                                       in->inScheme,
                                       &keyObject->sensitive.sensitive.any.b);
    // If the mac can't start, treat it as a fatal error
    if(out->outMAC.t.size == 0)
        return TPM_RC_FAILURE;
    CryptDigestUpdate2B(&state.hashState, &in->buffer.b);
    // If the MAC result is not what was expected, it is a fatal error
    if(CryptHmacEnd2B(&state, &out->outMAC.b) != out->outMAC.t.size)
        return TPM_RC_FAILURE;
    return TPM_RC_SUCCESS;
}
#endif // CC_MAC
```

16 Random Number Generator

16.1 TPM2_GetRandom

16.1.1 General Description

This command returns the next *bytesRequested* octets from the random number generator (RNG).

NOTE 1 It is recommended that a TPM implement the RNG in a manner that would allow it to return RNG octets such that, as long as the value of *bytesRequested* is not greater than the maximum digest size, the frequency of *bytesRequested* being more than the number of octets available is an infrequent occurrence.

If *bytesRequested* is more than will fit into a TPM2B_DIGEST on the TPM, no error is returned but the TPM will only return as much data as will fit into a TPM2B_DIGEST buffer for the TPM.

NOTE 2 TPM2B_DIGEST is large enough to hold the largest digest that may be produced by the TPM. Because that digest size changes according to the implemented hashes, the maximum amount of data returned by this command is TPM implementation-dependent.

16.1.2 Command and Response

Table 68 — TPM2_GetRandom Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit or encrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_GetRandom
UINT16	bytesRequested	number of octets to return

Table 69 — TPM2_GetRandom Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_DIGEST	randomBytes	the random octets

16.1.3 Detailed Actions

```
#include "Tpm.h"
#include "GetRandom_fp.h"
#if CC_GetRandom  // Conditional expansion of this file
TPM_RC
TPM2_GetRandom(
    GetRandom_In    *in,            // IN: input parameter list
    GetRandom_Out   *out            // OUT: output parameter list
    )
{
// Command Output

    // if the requested bytes exceed the output buffer size, generates the
    // maximum bytes that the output buffer allows
    if(in->bytesRequested > sizeof(TPMU_HA))
        out->randomBytes.t.size = sizeof(TPMU_HA);
    else
        out->randomBytes.t.size = in->bytesRequested;

    CryptRandomGenerate(out->randomBytes.t.size, out->randomBytes.t.buffer);

    return TPM_RC_SUCCESS;
}
#endif // CC_GetRandom
```

16.2 TPM2_StirRandom

16.2.1 General Description

This command is used to add "additional information" to the RNG state.

NOTE The "additional information" is as defined in SP800-90A.

The *inData* parameter may not be larger than 128 octets.

16.2.2 Command and Response

Table 70 — TPM2_StirRandom Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit or decrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_StirRandom {NV}
TPM2B_SENSITIVE_DATA	inData	additional information

Table 71 — TPM2_StirRandom Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

16.2.3 Detailed Actions

```
#include "Tpm.h"
#include "StirRandom_fp.h"
#if CC_StirRandom  // Conditional expansion of this file
TPM_RC
TPM2_StirRandom(
    StirRandom_In   *in             // IN: input parameter list
    )
{
// Internal Data Update
    CryptRandomStir(in->inData.t.size, in->inData.t.buffer);

    return TPM_RC_SUCCESS;
}
#endif // CC_StirRandom
```


17 Hash/HMAC/Event Sequences

17.1 Introduction

All of the commands in this group are to support sequences for which an intermediate state must be maintained. For a description of sequences, see “Hash, MAC, and Event Sequences” in TPM 2.0 Part 1.

A TPM may implement either TPM2_HMAC_Start() or TPM2_MAC_Start() but not both as they have the same command code and there is no way to distinguish them. A TPM that supports TPM2_MAC_Start() will support any code that was written to use TPM2_HMAC_Start() but a TPM that supports TPM2_HMAC_Start() will not support a MAC based on symmetric block ciphers.

17.2 TPM2_HMAC_Start

17.2.1 General Description

This command starts an HMAC sequence. The TPM will create and initialize an HMAC sequence structure, assign a handle to the sequence, and set the *authValue* of the sequence object to the value in *auth*.

NOTE 1 The structure of a sequence object is vendor-dependent.

The caller shall provide proper authorization for use of *handle*.

If the *sign* attribute is not SET in the key referenced by *handle* then the TPM shall return TPM_RC_KEY. If the key type is not TPM_ALG_KEYEDHASH then the TPM shall return TPM_RC_TYPE. If the key referenced by *handle* has the *restricted* attribute SET, the TPM shall return TPM_RC_ATTRIBUTES.

NOTE 2 For symmetric signing with a restricted key, see TPM2_Sign.

If the default scheme of the key referenced by *handle* is not TPM_ALG_NULL, then the *hashAlg* parameter is required to be either the same as the key’s default or TPM_ALG_NULL (TPM_RC_VALUE). If the default scheme of the key is TPM_ALG_NULL, then hashAlg is required to be a valid hash and not TPM_ALG_NULL (TPM_RC_VALUE).

Table 72 — Hash Selection Matrix

handle→restricted **(key's restricted attribute)**	***handle→scheme*** **(hash algorithm from key's *scheme*)**	***hashAlg***	**hash used**
CLEAR (unrestricted)	TPM_ALG_NULL(1)	TPM_ALG_NULL	error(1) (TPM_RC_VALUE)
CLEAR	TPM_ALG_NULL	valid hash	*hashAlg*
CLEAR	valid hash	TPM_ALG_NULL or same as *handle→scheme*	*handle→scheme*
CLEAR	valid hash	valid hash	error (TPM_RC_VALUE) if *hashAlg* != *handle->scheme*
SET (restricted)	don't care	don't care	TPM_RC_ATTRIBUTES
NOTES: 1) A hash algorithm is required for the HMAC.			

NOTE 1 A TPM may implement either TPM2_HMAC_Start() or TPM2_MAC_Start() but not both, as they have the same command code and there is no way to distinguish them. A TPM that supports TPM2_MAC_Start() will support any code that was written to use TPM2_HMAC_Start(), but a TPM that supports TPM2_HMAC_Start() will not support a MAC based on symmetric block ciphers.

17.2.2 Command and Response

Table 73 — TPM2_HMAC_Start Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_HMAC_Start
TPMI_DH_OBJECT	@handle	handle of an HMAC key Auth Index: 1 Auth Role: USER
TPM2B_AUTH	auth	authorization value for subsequent use of the sequence
TPMI_ALG_HASH+	hashAlg	the hash algorithm to use for the HMAC

Table 74 — TPM2_HMAC_Start Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPMI_DH_OBJECT	sequenceHandle	a handle to reference the sequence

17.2.3 Detailed Actions

```
#include "Tpm.h"
#include "HMAC_Start_fp.h"
#if CC_HMAC_Start  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_ATTRIBUTES	key referenced by *handle* is not a signing key or is restricted
TPM_RC_OBJECT_MEMORY	no space to create an internal object
TPM_RC_KEY	key referenced by *handle* is not an HMAC key
TPM_RC_VALUE	*hashAlg* is not compatible with the hash algorithm of the scheme of the object referenced by *handle*

```
TPM_RC
TPM2_HMAC_Start(
    HMAC_Start_In   *in,            // IN: input parameter list
    HMAC_Start_Out  *out            // OUT: output parameter list
    )
{
    OBJECT                   *keyObject;
    TPMT_PUBLIC              *publicArea;
    TPM_ALG_ID                hashAlg;

// Input Validation

    // Get HMAC key object and public area pointers
    keyObject = HandleToObject(in->handle);
    publicArea = &keyObject->publicArea;

    // Make sure that the key is an HMAC key
    if(publicArea->type != TPM_ALG_KEYEDHASH)
        return TPM_RCS_TYPE + RC_HMAC_Start_handle;

    // and that it is unrestricted
    if(IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, restricted))
        return TPM_RCS_ATTRIBUTES + RC_HMAC_Start_handle;

    // and that it is a signing key
    if(!IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, sign))
        return TPM_RCS_KEY + RC_HMAC_Start_handle;

    // See if the key has a default
    if(publicArea->parameters.keyedHashDetail.scheme.scheme == TPM_ALG_NULL)
        // it doesn't so use the input value
        hashAlg = in->hashAlg;
    else
    {
        // key has a default so use it
        hashAlg
            = publicArea->parameters.keyedHashDetail.scheme.details.hmac.hashAlg;
        // and verify that the input was either the  TPM_ALG_NULL or the default
        if(in->hashAlg != TPM_ALG_NULL && in->hashAlg != hashAlg)
            hashAlg = TPM_ALG_NULL;
    }
    // if we ended up without a hash algorithm then return an error
    if(hashAlg == TPM_ALG_NULL)
        return TPM_RCS_VALUE + RC_HMAC_Start_hashAlg;

// Internal Data Update

```

```
    // Create a HMAC sequence object. A TPM_RC_OBJECT_MEMORY error may be
    // returned at this point
    return ObjectCreateHMACSequence(hashAlg,
                                    keyObject,
                                    &in->auth,
                                    &out->sequenceHandle);
}
#endif // CC_HMAC_Start
```


17.3 TPM2_MAC_Start

17.3.1 General Description

This command starts a MAC sequence. The TPM will create and initialize a MAC sequence structure, assign a handle to the sequence, and set the *authValue* of the sequence object to the value in *auth*.

NOTE 1 The structure of a sequence object is vendor-dependent.

The caller shall provide proper authorization for use of *handle*.

If the *sign* attribute is not SET in the key referenced by *handle* then the TPM shall return TPM_RC_KEY. If the key type is not TPM_ALG_KEYEDHASH or TPM_ALG_SYMCIPHER then the TPM shall return TPM_RC_TYPE. If the key referenced by *handle* has the *restricted* attribute SET, the TPM shall return TPM_RC_ATTRIBUTES.

NOTE 2 For symmetric signing with a restricted key, see TPM2_Sign.

If the default scheme of the key referenced by *handle* is not TPM_ALG_NULL, then the *inScheme* parameter is required to be either the same as the key's default or TPM_ALG_NULL (TPM_RC_VALUE). If the default scheme of the key is TPM_ALG_NULL, then *inScheme*is required to be a valid hash or symmetric MAC scheme and not TPM_ALG_NULL (TPM_RC_VALUE).

Table 75 — Algorithm Selection Matrix

handle→restricted **(key's restricted attribute)**	***handle→scheme*** **(algorithm from key's *scheme*)**	***inScheme***	**algorithm used**
CLEAR (unrestricted)	TPM_ALG_NULL(1)	TPM_ALG_NULL	error(1) (TPM_RC_VALUE)
CLEAR	TPM_ALG_NULL	valid hash or symmetric MAC	*inScheme*
CLEAR	not TPM_ALG_NULL	TPM_ALG_NULL or same as *handle→scheme*	*handle→scheme*
CLEAR	not TPM_ALG_NULL	not TPM_AGL_NULL	error (TPM_RC_VALUE) if*inScheme*!= *handle->scheme*
SET (restricted)	don't care	don't care	TPM_RC_ATTRIBUTES

NOTES:
1) A hash algorithm is required for the HMAC.
2) hashAlg shall be TPM_ALG_NULL for handle referencing a CMAC key.

NOTE 3 For a TPM_ALG_SYMCIPHER key, the symmetric block cipher algorithm is part of the key definition.

NOTE 4 TPM2_MAC_Start() was added in revision 01.43.

17.3.2 Command and Response

Table 76 — TPM2_MAC_Start Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_MAC_Start
TPMI_DH_OBJECT	@handle	handle of a MAC key Auth Index: 1 Auth Role: USER
TPM2B_AUTH	auth	authorization value for subsequent use of the sequence
TPMI_ALG_MAC_SCHEME+	inScheme	the algorithm to use for the MAC

Table 77 — TPM2_MAC_Start Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPMI_DH_OBJECT	sequenceHandle	a handle to reference the sequence

17.3.3 Detailed Actions

```
#include "Tpm.h"
#include "MAC_Start_fp.h"
#if CC_MAC_Start  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_ATTRIBUTES	key referenced by *handle* is not a signing key or is restricted
TPM_RC_OBJECT_MEMORY	no space to create an internal object
TPM_RC_KEY	key referenced by *handle* is not an HMAC key
TPM_RC_VALUE	*hashAlg* is not compatible with the hash algorithm of the scheme of the object referenced by *handle*

```
TPM_RC
TPM2_MAC_Start(
    MAC_Start_In    *in,            // IN: input parameter list
    MAC_Start_Out   *out            // OUT: output parameter list
    )
{
    OBJECT                  *keyObject;
    TPMT_PUBLIC             *publicArea;
    TPM_RC                   result;

// Input Validation

    // Get HMAC key object and public area pointers
    keyObject = HandleToObject(in->handle);
    publicArea = &keyObject->publicArea;

    // Make sure that the key can do what is required
    result = CryptSelectMac(publicArea, &in->inScheme);
    // If the key is not able to do a MAC, indicate that the handle selects an
    // object that can't do a MAC
    if(result == TPM_RCS_TYPE)
        return TPM_RCS_TYPE + RC_MAC_Start_handle;
    // If there is another error type, indicate that the scheme and key are not
    // compatible
    if(result != TPM_RC_SUCCESS)
        return RcSafeAddToResult(result, RC_MAC_Start_inScheme);
    // Make sure that the key is not restricted
    if(IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, restricted))
        return TPM_RCS_ATTRIBUTES + RC_MAC_Start_handle;
    // and that it is a signing key
    if(!IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, sign))
        return TPM_RCS_KEY + RC_MAC_Start_handle;

// Internal Data Update
    // Create a HMAC sequence object. A TPM_RC_OBJECT_MEMORY error may be
    // returned at this point
    return ObjectCreateHMACSequence(in->inScheme,
                                    keyObject,
                                    &in->auth,
                                    &out->sequenceHandle);
}
#endif // CC_MAC_Start
```

17.4 TPM2_HashSequenceStart

17.4.1 General Description

This command starts a hash or an Event Sequence. If *hashAlg* is an implemented hash, then a hash sequence is started. If *hashAlg* is TPM_ALG_NULL, then an Event Sequence is started. If *hashAlg* is neither an implemented algorithm nor TPM_ALG_NULL, then the TPM shall return TPM_RC_HASH.

Depending on *hashAlg*, the TPM will create and initialize a Hash Sequence context or an Event Sequence context. Additionally, it will assign a handle to the context and set the *authValue* of the context to the value in *auth*. A sequence context for an Event (*hashAlg* = TPM_ALG_NULL) contains a hash context for each of the PCR banks implemented on the TPM.

17.4.2 Command and Response

Table 78 — TPM2_HashSequenceStart Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit or decrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_HashSequenceStart
TPM2B_AUTH	auth	authorization value for subsequent use of the sequence
TPMI_ALG_HASH+	hashAlg	the hash algorithm to use for the hash sequence An Event Sequence starts if this is TPM_ALG_NULL.

Table 79 — TPM2_HashSequenceStart Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPMI_DH_OBJECT	sequenceHandle	a handle to reference the sequence

17.4.3 Detailed Actions

```
#include "Tpm.h"
#include "HashSequenceStart_fp.h"
#if CC_HashSequenceStart  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_OBJECT_MEMORY	no space to create an internal object

```
TPM_RC
TPM2_HashSequenceStart(
    HashSequenceStart_In     *in,            // IN: input parameter list
    HashSequenceStart_Out    *out            // OUT: output parameter list
    )
{
// Internal Data Update

    if(in->hashAlg == TPM_ALG_NULL)
        // Start a event sequence.  A TPM_RC_OBJECT_MEMORY error may be
        // returned at this point
        return ObjectCreateEventSequence(&in->auth, &out->sequenceHandle);

    // Start a hash sequence.  A TPM_RC_OBJECT_MEMORY error may be
    // returned at this point
    return ObjectCreateHashSequence(in->hashAlg, &in->auth, &out->sequenceHandle);
}
#endif // CC_HashSequenceStart
```

17.5 TPM2_SequenceUpdate

17.5.1 General Description

This command is used to add data to a hash or HMAC sequence. The amount of data in buffer may be any size up to the limits of the TPM.

NOTE 1 In all TPM, a *buffer* size of 1,024 octets is allowed.

Proper authorization for the sequence object associated with *sequenceHandle* is required. If an authorization or audit of this command requires computation of a *cpHash* and an *rpHash*, the Name associated with *sequenceHandle* will be the Empty Buffer.

If the command does not return TPM_RC_SUCCESS, the state of the sequence is unmodified.

If the sequence is intended to produce a digest that will be signed by a restricted signing key, then the first block of data shall contain sizeof(TPM_GENERATED) octets and the first octets shall not be TPM_GENERATED_VALUE.

NOTE 2 This requirement allows the TPM to validate that the first block is safe to sign without having to accumulate octets over multiple calls.

17.5.2 Command and Response

Table 80 — TPM2_SequenceUpdate Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_SequenceUpdate
TPMI_DH_OBJECT	@sequenceHandle	handle for the sequence object Auth Index: 1 Auth Role: USER
TPM2B_MAX_BUFFER	buffer	data to be added to hash

Table 81 — TPM2_SequenceUpdate Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

17.5.3 Detailed Actions

```
#include "Tpm.h"
#include "SequenceUpdate_fp.h"
#if CC_SequenceUpdate  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_MODE	*sequenceHandle* does not reference a hash or HMAC sequence object

```
TPM_RC
TPM2_SequenceUpdate(
    SequenceUpdate_In   *in             // IN: input parameter list
    )
{
    OBJECT                  *object;
    HASH_OBJECT             *hashObject;

// Input Validation

    // Get sequence object pointer
    object = HandleToObject(in->sequenceHandle);
    hashObject = (HASH_OBJECT *)object;

    // Check that referenced object is a sequence object.
    if(!ObjectIsSequence(object))
        return TPM_RCS_MODE + RC_SequenceUpdate_sequenceHandle;

// Internal Data Update

    if(object->attributes.eventSeq == SET)
    {
        // Update event sequence object
        UINT32          i;
        for(i = 0; i < HASH_COUNT; i++)
        {
            // Update sequence object
            CryptDigestUpdate2B(&hashObject->state.hashState[i], &in->buffer.b);
        }
    }
    else
    {
        // Update hash/HMAC sequence object
        if(hashObject->attributes.hashSeq == SET)
        {
            // Is this the first block of the sequence
            if(hashObject->attributes.firstBlock == CLEAR)
            {
                // If so, indicate that first block was received
                hashObject->attributes.firstBlock = SET;

                // Check the first block to see if the first block can contain
                // the TPM_GENERATED_VALUE.  If it does, it is not safe for
                // a ticket.
                if(TicketIsSafe(&in->buffer.b))
                    hashObject->attributes.ticketSafe = SET;
            }
            // Update sequence object hash/HMAC stack
            CryptDigestUpdate2B(&hashObject->state.hashState[0], &in->buffer.b);
        }
        else if(object->attributes.hmacSeq == SET)
        {
```

```
            // Update sequence object HMAC stack
            CryptDigestUpdate2B(&hashObject->state.hmacState.hashState,
                                &in->buffer.b);
        }
    }
    return TPM_RC_SUCCESS;
}
#endif // CC_SequenceUpdate
```


17.6 TPM2_SequenceComplete

17.6.1 General Description

This command adds the last part of data, if any, to a hash/HMAC sequence and returns the result.

NOTE 1 This command is not used to complete an Event Sequence. TPM2_EventSequenceComplete() is used for that purpose.

For a hash sequence, if the results of the hash will be used in a signing operation that uses a restricted signing key, then the ticket returned by this command can indicate that the hash is safe to sign.

If the *digest* is not safe to sign, then *validation* will be a TPMT_TK_HASHCHECK with the hierarchy set to TPM_RH_NULL and *digest* set to the Empty Buffer.

If *hierarchy* is TPM_RH_NULL, then *digest* in the ticket will be the Empty Buffer.

NOTE 2 Regardless of the contents of the first octets of the hashed message, if the first buffer sent to the TPM had fewer than sizeof(TPM_GENERATED) octets, then the TPM will operate as if *digest* is not safe to sign.

NOTE 3 The ticket is only required for a signing operation that uses a restricted signing key. It is always returned, but can be ignored if not needed.

If *sequenceHandle* references an Event Sequence, then the TPM shall return TPM_RC_MODE.

Proper authorization for the sequence object associated with *sequenceHandle* is required. If an authorization or audit of this command requires computation of a *cpHash* and an *rpHash*, the Name associated with *sequenceHandle* will be the Empty Buffer.

If this command completes successfully, the *sequenceHandle* object will be flushed.

17.6.2 Command and Response

Table 82 — TPM2_SequenceComplete Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_SequenceComplete {F}
TPMI_DH_OBJECT	@sequenceHandle	authorization for the sequence Auth Index: 1 Auth Role: USER
TPM2B_MAX_BUFFER	buffer	data to be added to the hash/HMAC
TPMI_RH_HIERARCHY+	hierarchy	hierarchy of the ticket for a hash

Table 83 — TPM2_SequenceComplete Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_DIGEST	result	the returned HMAC or digest in a sized buffer
TPMT_TK_HASHCHECK	validation	ticket indicating that the sequence of octets used to compute *outDigest* did not start with TPM_GENERATED_VALUE This is a NULL Ticket when the sequence is HMAC.

17.6.3 Detailed Actions

```
#include "Tpm.h"
#include "SequenceComplete_fp.h"
#if CC_SequenceComplete  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_MODE	*sequenceHandle* does not reference a hash or HMAC sequence object

```
TPM_RC
TPM2_SequenceComplete(
    SequenceComplete_In      *in,            // IN: input parameter list
    SequenceComplete_Out     *out            // OUT: output parameter list
    )
{
    HASH_OBJECT                      *hashObject;
// Input validation
    // Get hash object pointer
    hashObject = (HASH_OBJECT *)HandleToObject(in->sequenceHandle);

    // input handle must be a hash or HMAC sequence object.
    if(hashObject->attributes.hashSeq == CLEAR
       && hashObject->attributes.hmacSeq == CLEAR)
        return TPM_RCS_MODE + RC_SequenceComplete_sequenceHandle;
// Command Output
    if(hashObject->attributes.hashSeq == SET)           // sequence object for hash
    {
       // Get the hash algorithm before the algorithm is lost in CryptHashEnd
        TPM_ALG_ID       hashAlg = hashObject->state.hashState[0].hashAlg;

        // Update last piece of the data
        CryptDigestUpdate2B(&hashObject->state.hashState[0], &in->buffer.b);

        // Complete hash
        out->result.t.size = CryptHashEnd(&hashObject->state.hashState[0],
                                          sizeof(out->result.t.buffer),
                                          out->result.t.buffer);
        // Check if the first block of the sequence has been received
        if(hashObject->attributes.firstBlock == CLEAR)
        {
            // If not, then this is the first block so see if it is 'safe'
            // to sign.
            if(TicketIsSafe(&in->buffer.b))
                hashObject->attributes.ticketSafe = SET;
        }
        // Output ticket
        out->validation.tag = TPM_ST_HASHCHECK;
        out->validation.hierarchy = in->hierarchy;

        if(in->hierarchy == TPM_RH_NULL)
        {
            // Ticket is not required
            out->validation.digest.t.size = 0;
        }
        else if(hashObject->attributes.ticketSafe == CLEAR)
        {
            // Ticket is not safe to generate
            out->validation.hierarchy = TPM_RH_NULL;
            out->validation.digest.t.size = 0;
        }
        else
```



```
        {
            // Compute ticket
            TicketComputeHashCheck(out->validation.hierarchy, hashAlg,
                                   &out->result, &out->validation);
        }
    }
    else
    {
        //   Update last piece of data
        CryptDigestUpdate2B(&hashObject->state.hmacState.hashState, &in->buffer.b);
#if !SMAC_IMPLEMENTED
        // Complete HMAC
        out->result.t.size = CryptHmacEnd(&(hashObject->state.hmacState),
                                          sizeof(out->result.t.buffer),
                                          out->result.t.buffer);
#else
        // Complete the MAC
        out->result.t.size = CryptMacEnd(&hashObject->state.hmacState,
                                         sizeof(out->result.t.buffer),
                                         out->result.t.buffer);
#endif
        // No ticket is generated for HMAC sequence
        out->validation.tag = TPM_ST_HASHCHECK;
        out->validation.hierarchy = TPM_RH_NULL;
        out->validation.digest.t.size = 0;
    }
// Internal Data Update
    // mark sequence object as evict so it will be flushed on the way out
    hashObject->attributes.evict = SET;

    return TPM_RC_SUCCESS;
}
#endif // CC_SequenceComplete
```


17.7 TPM2_EventSequenceComplete

17.7.1 General Description

This command adds the last part of data, if any, to an Event Sequence and returns the result in a digest list. If *pcrHandle* references a PCR and not TPM_RH_NULL, then the returned digest list is processed in the same manner as the digest list input parameter to TPM2_PCR_Extend(). That is, if a bank contains a PCR associated with *pcrHandle,* it is extended with the associated digest value from the list.

If *sequenceHandle* references a hash or HMAC sequence, the TPM shall return TPM_RC_MODE.

Proper authorization for the sequence object associated with *sequenceHandle* is required. If an authorization or audit of this command requires computation of a *cpHash* and an *rpHash*, the Name associated with *sequenceHandle* will be the Empty Buffer.

If this command completes successfully, the *sequenceHandle* object will be flushed.

NOTE: Unlike TPM2_PCR_Event(), a digest is always returned for each implemented hash algorithm. There is no option to only return digests for which *pcrHandle* is allocated.

17.7.2 Command and Response

Table 84 — TPM2_EventSequenceComplete Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_EventSequenceComplete {NV F}
TPMI_DH_PCR+	@pcrHandle	PCR to be extended with the Event data Auth Index: 1 Auth Role: USER
TPMI_DH_OBJECT	@sequenceHandle	authorization for the sequence Auth Index: 2 Auth Role: USER
TPM2B_MAX_BUFFER	buffer	data to be added to the Event

Table 85 — TPM2_EventSequenceComplete Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPML_DIGEST_VALUES	results	list of digests computed for the PCR

17.7.3 Detailed Actions

```
#include "Tpm.h"
#include "EventSequenceComplete_fp.h"
#if CC_EventSequenceComplete  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_LOCALITY	PCR extension is not allowed at the current locality
TPM_RC_MODE	input handle is not a valid event sequence object

```
TPM_RC
TPM2_EventSequenceComplete(
    EventSequenceComplete_In     *in,            // IN: input parameter list
    EventSequenceComplete_Out    *out            // OUT: output parameter list
    )
{
    HASH_OBJECT          *hashObject;
    UINT32                i;
    TPM_ALG_ID            hashAlg;
// Input validation
    // get the event sequence object pointer
    hashObject = (HASH_OBJECT *)HandleToObject(in->sequenceHandle);

    // input handle must reference an event sequence object
    if(hashObject->attributes.eventSeq != SET)
        return TPM_RCS_MODE + RC_EventSequenceComplete_sequenceHandle;

    // see if a PCR extend is requested in call
    if(in->pcrHandle != TPM_RH_NULL)
    {
        // see if extend of the PCR is allowed at the locality of the command,
        if(!PCRIsExtendAllowed(in->pcrHandle))
            return TPM_RC_LOCALITY;
        // if an extend is going to take place, then check to see if there has
        // been an orderly shutdown. If so, and the selected PCR is one of the
        // state saved PCR, then the orderly state has to change. The orderly state
        // does not change for PCR that are not preserved.
        // NOTE: This doesn't just check for Shutdown(STATE) because the orderly
        // state will have to change if this is a state-saved PCR regardless
        // of the current state. This is because a subsequent Shutdown(STATE) will
        // check to see if there was an orderly shutdown and not do anything if
        // there was. So, this must indicate that a future Shutdown(STATE) has
        // something to do.
        if(PCRIsStateSaved(in->pcrHandle))
            RETURN_IF_ORDERLY;
    }
// Command Output
    out->results.count = 0;

    for(i = 0; i < HASH_COUNT; i++)
    {
        hashAlg = CryptHashGetAlgByIndex(i);
        // Update last piece of data
        CryptDigestUpdate2B(&hashObject->state.hashState[i], &in->buffer.b);
        // Complete hash
        out->results.digests[out->results.count].hashAlg = hashAlg;
        CryptHashEnd(&hashObject->state.hashState[i],
                     CryptHashGetDigestSize(hashAlg),
                     (BYTE *)&out->results.digests[out->results.count].digest);
     // Extend PCR
        if(in->pcrHandle != TPM_RH_NULL)
```



```
            PCRExtend(in->pcrHandle, hashAlg,
                      CryptHashGetDigestSize(hashAlg),
                      (BYTE *)&out->results.digests[out->results.count].digest);
        out->results.count++;
    }
// Internal Data Update
    // mark sequence object as evict so it will be flushed on the way out
    hashObject->attributes.evict = SET;

    return TPM_RC_SUCCESS;
}
#endif // CC_EventSequenceComplete
```

18 Attestation Commands

18.1 Introduction

The attestation commands cause the TPM to sign an internally generated data structure. The contents of the data structure vary according to the command.

If the *sign* attribute is not SET in the key referenced by *signHandle* then the TPM shall return TPM_RC_KEY.

All signing commands include a parameter (typically *inScheme*) for the caller to specify a scheme to be used for the signing operation. This scheme will be applied only if the scheme of the key is TPM_ALG_NULL or the key handle is TPM_RH_NULL. If the scheme for *signHandle* is not TPM_ALG_NULL, then *inScheme.scheme* shall be TPM_ALG_NULL or the same as *scheme* in the public area of the key. If the scheme for *signHandle* is TPM_ALG_NULL or the key handle is TPM_RH_NULL, then *inScheme* will be used for the signing operation and may not be TPM_ALG_NULL. The TPM shall return TPM_RC_SCHEME to indicate that the scheme is not appropriate.

For a signing key that is not restricted, the caller may specify the scheme to be used as long as the scheme is compatible with the family of the key (for example, TPM_ALG_RSAPSS cannot be selected for an ECC key). If the caller sets *scheme* to TPM_ALG_NULL, then the default scheme of the key is used. For a restricted signing key, the key's scheme cannot be TPM_ALG_NULL and cannot be overridden.

If the handle for the signing key (*signHandle*) is TPM_RH_NULL, then all of the actions of the command are performed and the attestation block is “signed” with the NULL Signature.

NOTE 1 This mechanism is provided so that additional commands are not required to access the data that might be in an attestation structure.

NOTE 2 When *signHandle* is TPM_RH_NULL, *scheme* is still required to be a valid signing scheme (may be TPM_ALG_NULL), but the scheme will have no effect on the format of the signature. It will always be the NULL Signature.

TPM2_NV_Certify() is an attestation command that is documented in 1. The remaining attestation commands are collected in the remainder of this clause.

Each of the attestation structures contains a TPMS_CLOCK_INFO structure and a firmware version number. These values may be considered privacy-sensitive, because they would aid in the correlation of attestations by different keys. To provide improved privacy, the *resetCount*, *restartCount*, and *firmwareV*ersion numbers are obfuscated when the signing key is not in the Endorsement or Platform hierarchies.

The obfuscation value is computed by:

$$obfuscation \coloneqq \mathbf{KDFa}(signHandle{\rightarrow}nameAlg, shProof, \text{“OBFUSCATE”}, signHandle{\rightarrow}QN, 0, 128) \quad (3)$$

Of the returned 128 bits, 64 bits are added to the *versionNumber* field of the attestation structure; 32 bits are added to the *clockInfo.resetCount* and 32 bits are added to the *clockInfo.restartCount*. The order in which the bits are added is implementation-dependent.

NOTE 3 The obfuscation value for each signing key will be unique to that key in a specific location. That is, each version of a duplicated signing key will have a different obfuscation value.

When the signing key is TPM_RH_NULL, the data structure is produced but not signed; and the values in the signed data structure are obfuscated. When computing the obfuscation value for TPM_RH_NULL, the hash used for context integrity is used.

NOTE 4 The QN for TPM_RH_NULL is TPM_RH_NULL.

If the signing scheme of *signHandle* is an anonymous scheme, then the attestation blocks will not contain the Qualified Name of the *signHandle*.

Each of the attestation structures allows the caller to provide some qualifying data (*qualifyingData*). For most signing schemes, this value will be placed in the TPMS_ATTEST.*extraData* parameter that is then hashed and signed. However, for some schemes such as ECDAA, the *qualifyingData* is used in a different manner (for details, see "ECDAA" in TPM 2.0 Part 1).

18.2 TPM2_Certify

18.2.1 General Description

The purpose of this command is to prove that an object with a specific Name is loaded in the TPM. By certifying that the object is loaded, the TPM warrants that a public area with a given Name is self-consistent and associated with a valid sensitive area. If a relying party has a public area that has the same Name as a Name certified with this command, then the values in that public area are correct.

NOTE 1 See 18.1 for description of how the signing scheme is selected.

Authorization for *objectHandle* requires ADMIN role authorization. If performed with a policy session, the session shall have a policySession→*commandCode* set to TPM_CC_Certify. This indicates that the policy that is being used is a policy that is for certification, and not a policy that would approve another use. That is, authority to use an object does not grant authority to certify the object.

The object may be any object that is loaded with TPM2_Load() or TPM2_CreatePrimary(). An object that only has its public area loaded cannot be certified.

NOTE 2 The restriction occurs because the Name is used to identify the object being certified. If the TPM has not validated that the public area is associated with a matched sensitive area, then the public area may not represent a valid object and cannot be certified.

The certification includes the Name and Qualified Name of the certified object as well as the Name and the Qualified Name of the certifying object.

NOTE 3 If *signHandle* is TPM_RH_NULL, the TPMS_ATTEST structure is returned and *signature* is a NULL Signature.

18.2.2 Command and Response

Table 86 — TPM2_Certify Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_Certify
TPMI_DH_OBJECT	@objectHandle	handle of the object to be certified Auth Index: 1 Auth Role: ADMIN
TPMI_DH_OBJECT+	@signHandle	handle of the key used to sign the attestation structure Auth Index: 2 Auth Role: USER
TPM2B_DATA	qualifyingData	user provided qualifying data
TPMT_SIG_SCHEME+	inScheme	signing scheme to use if the *scheme* for *signHandle* is TPM_ALG_NULL

Table 87 — TPM2_Certify Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	.
TPM2B_ATTEST	certifyInfo	the structure that was signed
TPMT_SIGNATURE	signature	the asymmetric signature over *certifyInfo* using the key referenced by *signHandle*

18.2.3 Detailed Actions

```
#include "Tpm.h"
#include "Attest_spt_fp.h"
#include "Certify_fp.h"
#if CC_Certify  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_KEY	key referenced by *signHandle* is not a signing key
TPM_RC_SCHEME	*inScheme* is not compatible with *signHandle*
TPM_RC_VALUE	digest generated for *inScheme* is greater or has larger size than the modulus of *signHandle*, or the buffer for the result in *signature* is too small (for an RSA key); invalid commit status (for an ECC key with a split scheme)

```
TPM_RC
TPM2_Certify(
    Certify_In      *in,            // IN: input parameter list
    Certify_Out     *out            // OUT: output parameter list
    )
{
    TPMS_ATTEST             certifyInfo;
    OBJECT                  *signObject = HandleToObject(in->signHandle);
    OBJECT                  *certifiedObject = HandleToObject(in->objectHandle);
// Input validation
    if(!IsSigningObject(signObject))
        return TPM_RCS_KEY + RC_Certify_signHandle;
    if(!CryptSelectSignScheme(signObject, &in->inScheme))
        return TPM_RCS_SCHEME + RC_Certify_inScheme;

// Command Output
    // Filling in attest information
    // Common fields
    FillInAttestInfo(in->signHandle, &in->inScheme, &in->qualifyingData,
                     &certifyInfo);

    // Certify specific fields
    certifyInfo.type = TPM_ST_ATTEST_CERTIFY;
    // NOTE: the certified object is not allowed to be TPM_ALG_NULL so
    // 'certifiedObject' will never be NULL
    certifyInfo.attested.certify.name = certifiedObject->name;

    // When using an anonymous signing scheme, need to set the qualified Name to the
    // empty buffer to avoid correlation between keys
    if(CryptIsSchemeAnonymous(in->inScheme.scheme))
        certifyInfo.attested.certify.qualifiedName.t.size = 0;
    else
        certifyInfo.attested.certify.qualifiedName = certifiedObject->qualifiedName;

    // Sign attestation structure.  A NULL signature will be returned if
    // signHandle is TPM_RH_NULL.  A TPM_RC_NV_UNAVAILABLE, TPM_RC_NV_RATE,
    // TPM_RC_VALUE, TPM_RC_SCHEME or TPM_RC_ATTRIBUTES error may be returned
    // by SignAttestInfo()
    return SignAttestInfo(signObject, &in->inScheme, &certifyInfo,
                          &in->qualifyingData, &out->certifyInfo, &out->signature);
}
#endif // CC_Certify
```


18.3 TPM2_CertifyCreation

18.3.1 General Description

This command is used to prove the association between an object and its creation data. The TPM will validate that the ticket was produced by the TPM and that the ticket validates the association between a loaded public area and the provided hash of the creation data (*creationHash*).

NOTE 1 See 18.1 for description of how the signing scheme is selected.

The TPM will create a test ticket using the Name associated with *objectHandle* and *creationHash* as:

$$\mathbf{HMAC}(proof, (\mathrm{TPM\_ST\_CREATION} \mid\mid objectHandle{\rightarrow}Name \mid\mid creationHash)) \quad (4)$$

This ticket is then compared to creation ticket. If the tickets are not the same, the TPM shall return TPM_RC_TICKET.

If the ticket is valid, then the TPM will create a TPMS_ATTEST structure and place *creationHash* of the command in the *creationHash* field of the structure. The Name associated with *objectHandle* will be included in the attestation data that is then signed using the key associated with *signHandle*.

NOTE 2 If *signHandle* is TPM_RH_NULL, the TPMS_ATTEST structure is returned and *signature* is a NULL Signature.

objectHandle may be any object that is loaded with TPM2_Load() or TPM2_CreatePrimary().

18.3.2 Command and Response

Table 88 — TPM2_CertifyCreation Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_CertifyCreation
TPMI_DH_OBJECT+	@signHandle	handle of the key that will sign the attestation block Auth Index: 1 Auth Role: USER
TPMI_DH_OBJECT	objectHandle	the object associated with the creation data Auth Index: None
TPM2B_DATA	qualifyingData	user-provided qualifying data
TPM2B_DIGEST	creationHash	hash of the creation data produced by TPM2_Create() or TPM2_CreatePrimary()
TPMT_SIG_SCHEME+	inScheme	signing scheme to use if the *scheme* for *signHandle* is TPM_ALG_NULL
TPMT_TK_CREATION	creationTicket	ticket produced by TPM2_Create() or TPM2_CreatePrimary()

Table 89 — TPM2_CertifyCreation Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_ATTEST	certifyInfo	the structure that was signed
TPMT_SIGNATURE	signature	the signature over *certifyInfo*

18.3.3 Detailed Actions

```
#include "Tpm.h"
#include "Attest_spt_fp.h"
#include "CertifyCreation_fp.h"
#if CC_CertifyCreation  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_KEY	key referenced by *signHandle* is not a signing key
TPM_RC_SCHEME	*inScheme* is not compatible with *signHandle*
TPM_RC_TICKET	*creationTicket* does not match *objectHandle*
TPM_RC_VALUE	digest generated for *inScheme* is greater or has larger size than the modulus of *signHandle*, or the buffer for the result in *signature* is too small (for an RSA key); invalid commit status (for an ECC key with a split scheme).

```
TPM_RC
TPM2_CertifyCreation(
    CertifyCreation_In      *in,            // IN: input parameter list
    CertifyCreation_Out     *out            // OUT: output parameter list
    )
{
    TPMT_TK_CREATION        ticket;
    TPMS_ATTEST             certifyInfo;
    OBJECT                  *certified = HandleToObject(in->objectHandle);
    OBJECT                  *signObject = HandleToObject(in->signHandle);
// Input Validation
    if(!IsSigningObject(signObject))
        return TPM_RCS_KEY + RC_CertifyCreation_signHandle;
    if(!CryptSelectSignScheme(signObject, &in->inScheme))
        return TPM_RCS_SCHEME + RC_CertifyCreation_inScheme;

    // CertifyCreation specific input validation
    // Re-compute ticket
    TicketComputeCreation(in->creationTicket.hierarchy, &certified->name,
                          &in->creationHash, &ticket);
    // Compare ticket
    if(!MemoryEqual2B(&ticket.digest.b, &in->creationTicket.digest.b))
        return TPM_RCS_TICKET + RC_CertifyCreation_creationTicket;

// Command Output
    // Common fields
    FillInAttestInfo(in->signHandle, &in->inScheme, &in->qualifyingData,
                     &certifyInfo);

    // CertifyCreation specific fields
    // Attestation type
    certifyInfo.type = TPM_ST_ATTEST_CREATION;
    certifyInfo.attested.creation.objectName = certified->name;

    // Copy the creationHash
    certifyInfo.attested.creation.creationHash = in->creationHash;

    // Sign attestation structure.  A NULL signature will be returned if
    // signObject is TPM_RH_NULL.  A TPM_RC_NV_UNAVAILABLE, TPM_RC_NV_RATE,
    // TPM_RC_VALUE, TPM_RC_SCHEME or TPM_RC_ATTRIBUTES error may be returned at
    // this point
    return SignAttestInfo(signObject, &in->inScheme, &certifyInfo,
                          &in->qualifyingData, &out->certifyInfo,
                          &out->signature);
```



```
}
#endif // CC_CertifyCreation
```

18.4 TPM2_Quote

18.4.1 General Description

This command is used to quote PCR values.

The TPM will hash the list of PCR selected by *PCRselect* using the hash algorithm in the selected signing scheme. If the selected signing scheme or the scheme hash algorithm is TPM_ALG_NULL, then the TPM shall return TPM_RC_SCHEME.

NOTE 1 See 18.1 for description of how the signing scheme is selected.

The digest is computed as the hash of the concatenation of all of the digest values of the selected PCR.

The concatenation of PCR is described in TPM 2.0 Part 1, *Selecting Multiple PCR.*

NOTE 2 If *signHandle* is TPM_RH_NULL, the TPMS_ATTEST structure is returned and *signature* is a NULL Signature.

NOTE 3 A TPM may optionally return TPM_RC_SCHEME if *signHandle* is TPM_RH_NULL.

NOTE 4 Unlike TPM 1.2, TPM2_Quote does not return the PCR values. See Part 1, “Attesting to PCR” for a discussion of this issue.

18.4.2 Command and Response

Table 90 — TPM2_Quote Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_Quote
TPMI_DH_OBJECT+	@signHandle	handle of key that will perform signature Auth Index: 1 Auth Role: USER
TPM2B_DATA	qualifyingData	data supplied by the caller
TPMT_SIG_SCHEME+	inScheme	signing scheme to use if the *scheme* for *signHandle* is TPM_ALG_NULL
TPML_PCR_SELECTION	PCRselect	PCR set to quote

Table 91 — TPM2_Quote Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_ATTEST	quoted	the quoted information
TPMT_SIGNATURE	signature	the signature over *quoted*

18.4.3 Detailed Actions

```
#include "Tpm.h"
#include "Attest_spt_fp.h"
#include "Quote_fp.h"
#if CC_Quote  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_KEY	*signHandle* does not reference a signing key;
TPM_RC_SCHEME	the scheme is not compatible with sign key type, or input scheme is not compatible with default scheme, or the chosen scheme is not a valid sign scheme

```
TPM_RC
TPM2_Quote(
    Quote_In        *in,            // IN: input parameter list
    Quote_Out       *out            // OUT: output parameter list
    )
{
    TPMI_ALG_HASH             hashAlg;
    TPMS_ATTEST               quoted;
    OBJECT                   *signObject = HandleToObject(in->signHandle);
// Input Validation
    if(!IsSigningObject(signObject))
        return TPM_RCS_KEY + RC_Quote_signHandle;
    if(!CryptSelectSignScheme(signObject, &in->inScheme))
        return TPM_RCS_SCHEME + RC_Quote_inScheme;

// Command Output

    // Filling in attest information
    // Common fields
    // FillInAttestInfo may return TPM_RC_SCHEME or TPM_RC_KEY
    FillInAttestInfo(in->signHandle, &in->inScheme, &in->qualifyingData, &quoted);

    // Quote specific fields
    // Attestation type
    quoted.type = TPM_ST_ATTEST_QUOTE;

    // Get hash algorithm in sign scheme.  This hash algorithm is used to
    // compute PCR digest. If there is no algorithm, then the PCR cannot
    // be digested and this command returns TPM_RC_SCHEME
    hashAlg = in->inScheme.details.any.hashAlg;

    if(hashAlg == TPM_ALG_NULL)
        return TPM_RCS_SCHEME + RC_Quote_inScheme;

    // Compute PCR digest
    PCRComputeCurrentDigest(hashAlg, &in->PCRselect,
                            &quoted.attested.quote.pcrDigest);

    // Copy PCR select.  "PCRselect" is modified in PCRComputeCurrentDigest
    // function
    quoted.attested.quote.pcrSelect = in->PCRselect;

    // Sign attestation structure.  A NULL signature will be returned if
    // signObject is NULL.
    return SignAttestInfo(signObject, &in->inScheme, &quoted, &in->qualifyingData,
                          &out->quoted, &out->signature);
}
#endif // CC_Quote
```

18.5 TPM2_GetSessionAuditDigest

18.5.1 General Description

This command returns a digital signature of the audit session digest.

NOTE 1 See 18.1 for description of how the signing scheme is selected.

If *sessionHandle* is not an audit session, the TPM shall return TPM_RC_TYPE.

NOTE 2 A session does not become an audit session until the successful completion of the command in which the session is first used as an audit session.

This command requires authorization from the privacy administrator of the TPM (expressed with Endorsement Authorization) as well as authorization to use the key associated with *signHandle*.

If this command is audited, then the audit digest that is signed will not include the digest of this command because the audit digest is only updated when the command completes successfully.

This command does not cause the audit session to be closed and does not reset the digest value.

NOTE 3 If sessionHandle is used as an audit session for this command, the command is audited in the same manner as any other command.

NOTE 4 If *signHandle* is TPM_RH_NULL, the TPMS_ATTEST structure is returned and *signature* is a NULL Signature.

18.5.2 Command and Response

Table 92 — TPM2_GetSessionAuditDigest Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_GetSessionAuditDigest
TPMI_RH_ENDORSEMENT	@privacyAdminHandle	handle of the privacy administrator (TPM_RH_ENDORSEMENT) Auth Index: 1 Auth Role: USER
TPMI_DH_OBJECT+	@signHandle	handle of the signing key Auth Index: 2 Auth Role: USER
TPMI_SH_HMAC	sessionHandle	handle of the audit session Auth Index: None
TPM2B_DATA	qualifyingData	user-provided qualifying data – may be zero-length
TPMT_SIG_SCHEME+	inScheme	signing scheme to use if the *scheme* for *signHandle* is TPM_ALG_NULL

Table 93 — TPM2_GetSessionAuditDigest Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_ATTEST	auditInfo	the audit information that was signed
TPMT_SIGNATURE	signature	the signature over *auditInfo*

18.5.3 Detailed Actions

```
#include "Tpm.h"
#include "Attest_spt_fp.h"
#include "GetSessionAuditDigest_fp.h"
#if CC_GetSessionAuditDigest  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_KEY	key referenced by *signHandle* is not a signing key
TPM_RC_SCHEME	*inScheme* is incompatible with *signHandle* type; or both *scheme* and key's default scheme are empty; or *scheme* is empty while key's default scheme requires explicit input scheme (split signing); or non-empty default key scheme differs from *scheme*
TPM_RC_TYPE	*sessionHandle* does not reference an audit session
TPM_RC_VALUE	digest generated for the given *scheme* is greater than the modulus of *signHandle* (for an RSA key); invalid commit status or failed to generate r value (for an ECC key)

```
TPM_RC
TPM2_GetSessionAuditDigest(
    GetSessionAuditDigest_In    *in,            // IN: input parameter list
    GetSessionAuditDigest_Out   *out            // OUT: output parameter list
    )
{
    SESSION                  *session = SessionGet(in->sessionHandle);
    TPMS_ATTEST               auditInfo;
    OBJECT                  *signObject = HandleToObject(in->signHandle);
// Input Validation
    if(!IsSigningObject(signObject))
        return TPM_RCS_KEY + RC_GetSessionAuditDigest_signHandle;
    if(!CryptSelectSignScheme(signObject, &in->inScheme))
        return TPM_RCS_SCHEME + RC_GetSessionAuditDigest_inScheme;

    // session must be an audit session
    if(session->attributes.isAudit == CLEAR)
        return TPM_RCS_TYPE + RC_GetSessionAuditDigest_sessionHandle;

// Command Output
    // Fill in attest information common fields
    FillInAttestInfo(in->signHandle, &in->inScheme, &in->qualifyingData,
                     &auditInfo);

    // SessionAuditDigest specific fields
    auditInfo.type = TPM_ST_ATTEST_SESSION_AUDIT;
    auditInfo.attested.sessionAudit.sessionDigest = session->u2.auditDigest;

    // Exclusive audit session
    auditInfo.attested.sessionAudit.exclusiveSession
        = (g_exclusiveAuditSession == in->sessionHandle);

    // Sign attestation structure.  A NULL signature will be returned if
    // signObject is NULL.
    return SignAttestInfo(signObject, &in->inScheme, &auditInfo,
                          &in->qualifyingData, &out->auditInfo,
                          &out->signature);
}
#endif // CC_GetSessionAuditDigest
```


18.6 TPM2_GetCommandAuditDigest

18.6.1 General Description

This command returns the current value of the command audit digest, a digest of the commands being audited, and the audit hash algorithm. These values are placed in an attestation structure and signed with the key referenced by *signHandle*.

NOTE 1 See 18.1 for description of how the signing scheme is selected.

When this command completes successfully, and *signHandle* is not TPM_RH_NULL, the audit digest is cleared. If signHandle is TPM_RH_NULL, *signature* is the Empty Buffer and the audit digest is not cleared.

NOTE 2 The way that the TPM tracks that the digest is clear is vendor-dependent. The reference implementation resets the size of the digest to zero.

If this command is being audited, then the signed digest produced by the command will not include the command. At the end of this command, the audit digest will be extended with *cpHash* and the *rpHash* of the command, which would change the command audit digest signed by the next invocation of this command.

This command requires authorization from the privacy administrator of the TPM (expressed with Endorsement Authorization) as well as authorization to use the key associated with *signHandle*.

18.6.2 Command and Response

Table 94 — TPM2_GetCommandAuditDigest Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_GetCommandAuditDigest {NV}
TPMI_RH_ENDORSEMENT	@privacyHandle	handle of the privacy administrator (TPM_RH_ENDORSEMENT) Auth Index: 1 Auth Role: USER
TPMI_DH_OBJECT+	@signHandle	the handle of the signing key Auth Index: 2 Auth Role: USER
TPM2B_DATA	qualifyingData	other data to associate with this audit digest
TPMT_SIG_SCHEME+	inScheme	signing scheme to use if the *scheme* for *signHandle* is TPM_ALG_NULL

Table 95 — TPM2_GetCommandAuditDigest Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_ATTEST	auditInfo	the auditInfo that was signed
TPMT_SIGNATURE	signature	the signature over *auditInfo*

18.6.3 Detailed Actions

```
#include "Tpm.h"
#include "Attest_spt_fp.h"
#include "GetCommandAuditDigest_fp.h"
#if CC_GetCommandAuditDigest  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_KEY	key referenced by *signHandle* is not a signing key
TPM_RC_SCHEME	*inScheme* is incompatible with *signHandle* type; or both *scheme* and key's default scheme are empty; or *scheme* is empty while key's default scheme requires explicit input scheme (split signing); or non-empty default key scheme differs from *scheme*
TPM_RC_VALUE	digest generated for the given *scheme* is greater than the modulus of *signHandle* (for an RSA key); invalid commit status or failed to generate r value (for an ECC key)

```
TPM_RC
TPM2_GetCommandAuditDigest(
    GetCommandAuditDigest_In    *in,            // IN: input parameter list
    GetCommandAuditDigest_Out   *out            // OUT: output parameter list
    )
{
    TPM_RC                  result;
    TPMS_ATTEST             auditInfo;
    OBJECT                 *signObject = HandleToObject(in->signHandle);
// Input validation
    if(!IsSigningObject(signObject))
        return TPM_RCS_KEY + RC_GetCommandAuditDigest_signHandle;
    if(!CryptSelectSignScheme(signObject, &in->inScheme))
        return TPM_RCS_SCHEME + RC_GetCommandAuditDigest_inScheme;

// Command Output
    // Fill in attest information common fields
    FillInAttestInfo(in->signHandle, &in->inScheme, &in->qualifyingData,
                     &auditInfo);

    // CommandAuditDigest specific fields
    auditInfo.type = TPM_ST_ATTEST_COMMAND_AUDIT;
    auditInfo.attested.commandAudit.digestAlg = gp.auditHashAlg;
    auditInfo.attested.commandAudit.auditCounter = gp.auditCounter;

    // Copy command audit log
    auditInfo.attested.commandAudit.auditDigest = gr.commandAuditDigest;
    CommandAuditGetDigest(&auditInfo.attested.commandAudit.commandDigest);

    // Sign attestation structure.  A NULL signature will be returned if
    // signHandle is TPM_RH_NULL.  A TPM_RC_NV_UNAVAILABLE, TPM_RC_NV_RATE,
    // TPM_RC_VALUE, TPM_RC_SCHEME or TPM_RC_ATTRIBUTES error may be returned at
    // this point
    result = SignAttestInfo(signObject, &in->inScheme, &auditInfo,
                            &in->qualifyingData, &out->auditInfo,
                            &out->signature);
    // Internal Data Update
    if(result == TPM_RC_SUCCESS && in->signHandle != TPM_RH_NULL)
        // Reset log
        gr.commandAuditDigest.t.size = 0;

    return result;
}
```



```
#endif // CC_GetCommandAuditDigest
```

18.7 TPM2_GetTime

18.7.1 General Description

This command returns the current values of *Time* and *Clock*.

NOTE 1 See 18.1 for description of how the signing scheme is selected.

The values of *Clock*, *resetCount* and *restartCount* appear in two places in *timeInfo*: once in TPMS_ATTEST.*clockInfo* and again in TPMS_ATTEST.*attested.time.clockInfo.* The firmware version number also appears in two places (TPMS_ATTEST.*firmwareVersion* and TPMS_ATTEST.*attested.time.firmwareVersion*). If *signHandle* is in the endorsement or platform hierarchies, both copies of the data will be the same. However, if *signHandle* is in the storage hierarchy or is TPM_RH_NULL, the values in TPMS_ATTEST.*clockInfo* and TPMS_ATTEST.*firmwareVersion* are obfuscated but the values in TPMS_ATTEST.*attested.time* are not.

NOTE 2 The purpose of this duplication is to allow an entity who is trusted by the privacy Administrator to correlate the obfuscated values with the clear-text values. This command requires Endorsement Authorization.

NOTE 3 If *signHandle* is TPM_RH_NULL, the TPMS_ATTEST structure is returned and *signature* is a NULL Signature.

18.7.2 Command and Response

Table 96 — TPM2_GetTime Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_GetTime
TPMI_RH_ENDORSEMENT	@privacyAdminHandle	handle of the privacy administrator (TPM_RH_ENDORSEMENT) Auth Index: 1 Auth Role: USER
TPMI_DH_OBJECT+	@signHandle	the *keyHandle* identifier of a loaded key that can perform digital signatures Auth Index: 2 Auth Role: USER
TPM2B_DATA	qualifyingData	data to tick stamp
TPMT_SIG_SCHEME+	inScheme	signing scheme to use if the *scheme* for *signHandle* is TPM_ALG_NULL

Table 97 — TPM2_GetTime Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	.
TPM2B_ATTEST	timeInfo	standard TPM-generated attestation block
TPMT_SIGNATURE	signature	the signature over *timeInfo*

18.7.3 Detailed Actions

```
#include "Tpm.h"
#include "Attest_spt_fp.h"
#include "GetTime_fp.h"
#if CC_GetTime  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_KEY	key referenced by *signHandle* is not a signing key
TPM_RC_SCHEME	*inScheme* is incompatible with *signHandle* type; or both *scheme* and key's default scheme are empty; or *scheme* is empty while key's default scheme requires explicit input scheme (split signing); or non-empty default key scheme differs from *scheme*
TPM_RC_VALUE	digest generated for the given *scheme* is greater than the modulus of *signHandle* (for an RSA key); invalid commit status or failed to generate r value (for an ECC key)

```
TPM_RC
TPM2_GetTime(
    GetTime_In      *in,            // IN: input parameter list
    GetTime_Out     *out            // OUT: output parameter list
    )
{
    TPMS_ATTEST             timeInfo;
    OBJECT                 *signObject = HandleToObject(in->signHandle);
// Input Validation
    if(!IsSigningObject(signObject))
        return TPM_RCS_KEY + RC_GetTime_signHandle;
    if(!CryptSelectSignScheme(signObject, &in->inScheme))
        return TPM_RCS_SCHEME + RC_GetTime_inScheme;

// Command Output
    // Fill in attest common fields
    FillInAttestInfo(in->signHandle, &in->inScheme, &in->qualifyingData, &timeInfo);

    // GetClock specific fields
    timeInfo.type = TPM_ST_ATTEST_TIME;
    timeInfo.attested.time.time.time = g_time;
    TimeFillInfo(&timeInfo.attested.time.time.clockInfo);

    // Firmware version in plain text
    timeInfo.attested.time.firmwareVersion
        = (((UINT64)gp.firmwareV1) << 32) + gp.firmwareV2;

    // Sign attestation structure.  A NULL signature will be returned if
    // signObject is NULL.
    return SignAttestInfo(signObject, &in->inScheme, &timeInfo, &in->qualifyingData,
                          &out->timeInfo, &out->signature);
}
#endif // CC_GetTime
```

18.8 TPM2_CertifyX509

18.8.1 General Description

The purpose of this command is to generate an X.509 certificate that proves an object with a specific public key and attributes is loaded in the TPM. In contrast to TPM2_Certify, which uses a TCG-defined data structure to convey attestation information, TPM2_CertifyX509 encodes the attestation information in a DER-encoded X.509 certificate that is compliant with RFC5280 *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile.*

As described in RFC, an X.509 certificate contains a collection of data that is hashed and signed. The full signature is the combination of the *to be signed* (TBS) data, a description of the signature algorithm, and the signature over the TBS data. The elements of the TBS data structure are DER-encoded values. They are:

1) Version [0] – integer value of 2 indicating version 3
2) Certificate Serial Number – integer value
3) Signature Algorithm Identifier – values (usually a collection of OIDs) identifying the algorithm used for the signature
4) Issuer Name – X.501 type *Name* to identify the entity that has authorized the use of *signHandle* to create the certificate.
5) Validity – two time values indicating the period during which the certificate is valid
6) Subject Name – X.501 type *Name* that identifies the entity that authorized the use of *objectHandle*
7) Subject Public Key Info – the public key associated with *objectHandle*,
8) Extensions [3] – a set of values that "provide methods for associating additional attributes with users or public keys and for managing relationships between CAs."

NOTE 1: The numbers in square brackets (e.g., [0]) indicate application-specific tag values that are used to identify the type of the field.

NOTE 2: RFC 5280 describes two fields (issuerUniqueID and subjectUniqueID) but goes on to say: "CAs conforming to this profile MUST NOT generate certificates with unique identifiers." The TPM does not allow them to be present.

The caller provides a partial certificate (*partialCertificate*) parameter that contains four or five of the elements enumerated above in a DER encoded SEQUENCE. They are:

1) Signature Algorithm Identifier (optional)
2) Issuer (mandatory)
3) Validity (mandatory)
4) Subject Name (mandatory)
5) Extensions (mandatory)

The fields are required to be in the order in which they are listed above.

NOTE 3: The TPM determines if the Signature Algorithm Identifier element is present by counting the elements.

The optional Signature Algorithm Identifier may be provided by the caller. If it is not present, the TPM will generate the value based on the selected signing scheme. If the caller provides this value, then the TPM will use it in the completed TBS. The TPM will not validate that the provided values are compatible with the signing scheme. If the caller does not provide this field and the TPM does not have OID values for the signing scheme, then the TPM will return an error (TPM_RC_SCHEME).

NOTE 4: The TPM may implement signing schemes for which OIDs are not defined at the time the TPM was manufactured. Those schemes may still be used if the caller can provide the Signature Algorithm Identifier.

The Extensions element is required to contain a Key Usage extension. The TPM will extract the Key Usage values and verify that the attributes of *objectHandle* are consistent with the selected values (TPM_RC_ATTRIBUTES)(See Part 2, *TPMA_X509_KEY_USAGE*).

The Extensions element may contain a TPMA_OBJECT extension. If present, the TPM will extract the value and verify that the extension value exactly matches the TPMA_OBJECT of *objectKey* (TPM_RC_ATTRIBUTES). The element uses the TCG OID tcg-tpmaObject, 2.23.133.10.1.1.1. It is a SEQUENCE containing that OID and an OCTET STRING encapsulating a 4-byte BIT STRING holding the big endian TPMA_OBJECT.

signHandle is required to have the *sign* attribute SET (TPM_RC_KEY).

NOTE 5: See 18.1 for description of how the signing scheme is selected.

Authorization for *objectHandle* requires ADMIN role authorization. If performed with a policy session, the session shall have a policySession→*commandCode* set to TPM_CC_CertifyX509. This indicates that the policy that is being used is a policy that is for certification, and not a policy that would approve another use. That is, authority to use an object does not grant authority to certify the object.

If *objectHandle* does not have a sensitive area loaded, the TPM will return an error (TPM_RC_AUTH_UNAVAILABLE).

NOTE 6: The command requires that authorization be provided for use of *objectHandle.* An object that only has its *publicArea* loaded does not have an authorization value and the *authPolicy* has no meaning as the sensitive area is not present.

The TPM will create the Version, the Certificate Serial Number, the Subject Public Key Info, and, if not provided by the caller, the Signature Algorithm Identifier. These TPM-created values will be combined with the provided values to make a full TBSCerfificate structure (See RFC 5280, clause 4.1). The TPM will then sign the certificate using the selected signing scheme.

The TPM-created values will be returned in *addedToCertificate.* If the TPM creates the Signature Algorithm Identifier, it will be in *addedToCertificate* before the Subject Public Key Info. The TPM returns *tbsDigest* as a debugging aid.

NOTE 7: These returned fields allow the caller to unambiguously create a full RFC5280-defined TBSCertificate.

NOTE 8: This command was added in revision 01.53.

18.8.2 Command and Response

Table 98 — TPM2_CertifyX509 Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_CertifyX509
TPMI_DH_OBJECT	@objectHandle	handle of the object to be certified Auth Index: 1 Auth Role: ADMIN
TPMI_DH_OBJECT+	@signHandle	handle of the key used to sign the attestation structure Auth Index: 2 Auth Role: USER
TPM2B_DATA	reserved	shall be an Empty Buffer
TPMT_SIG_SCHEME+	inScheme	signing scheme to use if the *scheme* for *signHandle* is TPM_ALG_NULL
TPM2B_MAX_BUFFER	partialCertificate	a DER encoded partial certificate

Table 99 — TPM2_CertifyX509 Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	.
TPM2B_MAX_BUFFER	addedToCertificate	a DER encoded SEQUENCE containing the DER encoded fields added to partialCertificate to make it a complete RFC5280 TBSCertificate.
TPM2B_DIGEST	tbsDigest	the digest that was signed
TPMT_SIGNATURE	signature	The signature over *tbsDigest*

18.8.3 Detailed Actions

```
#include "Tpm.h"
#include "CertifyX509_fp.h"
#include "X509.h"
#include "TpmASN1_fp.h"
#include "X509_spt_fp.h"
#include "Attest_spt_fp.h"
#include "Platform_fp.h"
#if CC_CertifyX509 // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_ATTRIBUTES	the attributes of *objectHandle* are not compatible with the KeyUsage() or TPMA_OBJECT values in the extensions fields
TPM_RC_BINDING	the public and private portions of the key are not properly bound.
TPM_RC_HASH	the hash algorithm in the scheme is not supported
TPM_RC_KEY	*signHandle* does not reference a signing key;
TPM_RC_SCHEME	the scheme is not compatible with sign key type, or input scheme is not compatible with default scheme, or the chosen scheme is not a valid sign scheme
TPM_RC_VALUE	most likely a problem with the format of *partialCertificate*

```
TPM_RC
TPM2_CertifyX509(
    CertifyX509_In          *in,          // IN: input parameter list
    CertifyX509_Out         *out            // OUT: output parameter list
)
{
    TPM_RC                   result;
    OBJECT                  *signKey = HandleToObject(in->signHandle);
    OBJECT                  *object = HandleToObject(in->objectHandle);
    HASH_STATE               hash;
    INT16                    length;        // length for a tagged element
    ASN1UnmarshalContext     ctx;
    ASN1MarshalContext       ctxOut;
    // certTBS holds an array of pointers and lengths. Each entry references the
    // corresponding value in a TBSCertificate structure. For example, the 1th
    // element references the version number
    stringRef                certTBS[REF_COUNT] = {{0}};
#define ALLOWED_SEQUENCES   (SUBJECT_PUBLIC_KEY_REF - SIGNATURE_REF)
    stringRef                partial[ALLOWED_SEQUENCES] = {{0}};
    INT16                    countOfSequences = 0;
    INT16                    i;
    //
#if CERTIFYX509_DEBUG
    DebugFileOpen();
    DebugDumpBuffer(in->partialCertificate.t.size, in->partialCertificate.t.buffer,
        "partialCertificate");
#endif

    // Input Validation
    if(in->reserved.b.size != 0)
        return TPM_RC_SIZE + RC_CertifyX509_reserved;
    // signing key must be able to sign
    if(!IsSigningObject(signKey))
        return TPM_RCS_KEY + RC_CertifyX509_signHandle;
    // Pick a scheme for sign.  If the input sign scheme is not compatible with
    // the default scheme, return an error.
```

```
    if(!CryptSelectSignScheme(signKey, &in->inScheme))
        return TPM_RCS_SCHEME + RC_CertifyX509_inScheme;
    // Make sure that the public Key encoding is known
    if(X509AddPublicKey(NULL, object) == 0)
        return TPM_RCS_ASYMMETRIC + RC_CertifyX509_objectHandle;
    // Unbundle 'partialCertificate'.
        // Initialize the unmarshaling context
    if(!ASN1UnmarshalContextInitialize(&ctx, in->partialCertificate.t.size,
        in->partialCertificate.t.buffer))
        return TPM_RCS_VALUE + RC_CertifyX509_partialCertificate;
    // Make sure that this is a constructed SEQUENCE
    length = ASN1NextTag(&ctx);
    // Must be a constructed SEQUENCE that uses all of the input parameter
    if((ctx.tag != (ASN1_CONSTRUCTED_SEQUENCE))
        || ((ctx.offset + length) != in->partialCertificate.t.size))
        return TPM_RCS_SIZE + RC_CertifyX509_partialCertificate;

    // This scans through the contents of the outermost SEQUENCE. This would be the
    // 'issuer', 'validity', 'subject', 'issuerUniqueID' (optional),
    // 'subjectUniqueID' (optional), and 'extensions.'
    while(ctx.offset < ctx.size)
    {
        INT16           startOfElement = ctx.offset;
        //
            // Read the next tag and length field.
        length = ASN1NextTag(&ctx);
        if(length < 0)
            break;
        if(ctx.tag == ASN1_CONSTRUCTED_SEQUENCE)
        {
            partial[countOfSequences].buf = &ctx.buffer[startOfElement];
            ctx.offset += length;
            partial[countOfSequences].len = (INT16)ctx.offset - startOfElement;
            if(++countOfSequences > ALLOWED_SEQUENCES)
                break;
        }
        else if(ctx.tag  == X509_EXTENSIONS)
        {
            if(certTBS[EXTENSIONS_REF].len != 0)
                return TPM_RCS_VALUE + RC_CertifyX509_partialCertificate;
            certTBS[EXTENSIONS_REF].buf = &ctx.buffer[startOfElement];
            ctx.offset += length;
            certTBS[EXTENSIONS_REF].len =
                (INT16)ctx.offset - startOfElement;
        }
        else
            return TPM_RCS_VALUE + RC_CertifyX509_partialCertificate;
    }
    // Make sure that we used all of the data and found at least the required
    // number of elements.
    if((ctx.offset != ctx.size) || (countOfSequences < 3)
        || (countOfSequences > 4)
        || (certTBS[EXTENSIONS_REF].buf == NULL))
        return TPM_RCS_VALUE + RC_CertifyX509_partialCertificate;
    // Now that we know how many sequences there were, we can put them where they
    // belong
    for(i = 0; i < countOfSequences; i++)
        certTBS[SUBJECT_KEY_REF - i] = partial[countOfSequences - 1 - i];

    // If only three SEQUENCES, then the TPM needs to produce the signature algorithm.
    // See if it can
    if((countOfSequences == 3) &&
        (X509AddSigningAlgorithm(NULL, signKey, &in->inScheme) == 0))
            return TPM_RCS_SCHEME + RC_CertifyX509_signHandle;

    // Process the extensions
```

```
    result = X509ProcessExtensions(object, &certTBS[EXTENSIONS_REF]);
    if(result != TPM_RC_SUCCESS)
        // If the extension has the TPMA_OBJECT extension and the attributes don't
        // match, then the error code will be TPM_RCS_ATTRIBUTES. Otherwise, the error
        // indicates a malformed partialCertificate.
        return result + ((result == TPM_RCS_ATTRIBUTES)
                         ? RC_CertifyX509_objectHandle
                         : RC_CertifyX509_partialCertificate);
// Command Output
// Create the addedToCertificate values

    // Build the addedToCertificate from the bottom up.
    // Initialize the context structure
    ASN1InitializeMarshalContext(&ctxOut, sizeof(out->addedToCertificate.t.buffer),
                                 out->addedToCertificate.t.buffer);
    // Place a marker for the overall context
    ASN1StartMarshalContext(&ctxOut);  // SEQUENCE for addedToCertificate

    // Add the subject public key descriptor
    certTBS[SUBJECT_PUBLIC_KEY_REF].len = X509AddPublicKey(&ctxOut, object);
    certTBS[SUBJECT_PUBLIC_KEY_REF].buf = ctxOut.buffer + ctxOut.offset;
    // If the caller didn't provide the algorithm identifier, create it
    if(certTBS[SIGNATURE_REF].len == 0)
    {
        certTBS[SIGNATURE_REF].len = X509AddSigningAlgorithm(&ctxOut, signKey,
            &in->inScheme);
        certTBS[SIGNATURE_REF].buf = ctxOut.buffer + ctxOut.offset;
    }
    // Create the serial number value. Use the out->tbsDigest as scratch.
    {
        TPM2B                   *digest = &out->tbsDigest.b;
        //
        digest->size = (INT16)CryptHashStart(&hash, signKey->publicArea.nameAlg);
        pAssert(digest->size != 0);

        // The serial number size is the smaller of the digest and the vendor-defined
        // value
        digest->size = MIN(digest->size, SIZE_OF_X509_SERIAL_NUMBER);
        // Add all the parts of the certificate other than the serial number
        // and version number
        for(i = SIGNATURE_REF; i < REF_COUNT; i++)
            CryptDigestUpdate(&hash, certTBS[i].len, certTBS[i].buf);
        // throw in the Name of the signing key...
        CryptDigestUpdate2B(&hash, &signKey->name.b);
        // ...and the Name of the signed key.
        CryptDigestUpdate2B(&hash, &object->name.b);
        // Done
        CryptHashEnd2B(&hash, digest);
    }

    // Add the serial number
    certTBS[SERIAL_NUMBER_REF].len =
        ASN1PushInteger(&ctxOut, out->tbsDigest.t.size, out->tbsDigest.t.buffer);
    certTBS[SERIAL_NUMBER_REF].buf = ctxOut.buffer + ctxOut.offset;

    // Add the static version number
    ASN1StartMarshalContext(&ctxOut);
    ASN1PushUINT(&ctxOut, 2);
    certTBS[VERSION_REF].len =
        ASN1EndEncapsulation(&ctxOut, ASN1_APPLICAIION_SPECIFIC);
    certTBS[VERSION_REF].buf = ctxOut.buffer + ctxOut.offset;

    // Create a fake tag and length for the TBS in the space used for
    // 'addedToCertificate'
    {
        for(length = 0, i = 0; i < REF_COUNT; i++)
```

```
            length += certTBS[i].len;
        // Put a fake tag and length into the buffer for use in the tbsDigest
        certTBS[ENCODED_SIZE_REF].len =
            ASN1PushTagAndLength(&ctxOut, ASN1_CONSTRUCTED_SEQUENCE, length);
        certTBS[ENCODED_SIZE_REF].buf = ctxOut.buffer + ctxOut.offset;
        // Restore the buffer pointer to add back the number of octets used for the
        // tag and length
        ctxOut.offset += certTBS[ENCODED_SIZE_REF].len;
    }
    // sanity check
    if(ctxOut.offset < 0)
        return TPM_RC_FAILURE;
    // Create the tbsDigest to sign
    out->tbsDigest.t.size = CryptHashStart(&hash, in->inScheme.details.any.hashAlg);
    for(i = 0; i < REF_COUNT; i++)
        CryptDigestUpdate(&hash, certTBS[i].len, certTBS[i].buf);
    CryptHashEnd2B(&hash, &out->tbsDigest.b);

#if CERTIFYX509_DEBUG
    {
        BYTE                 fullTBS[4096];
        BYTE                *fill = fullTBS;
        int                  j;
        for (j = 0; j < REF_COUNT; j++)
        {
            MemoryCopy(fill, certTBS[j].buf, certTBS[j].len);
            fill += certTBS[j].len;
        }
        DebugDumpBuffer((int)(fill - &fullTBS[0]), fullTBS, "\nfull TBS");
    }
#endif

// Finish up the processing of addedToCertificate
    // Create the actual tag and length for the addedToCertificate structure
    out->addedToCertificate.t.size =
        ASN1EndEncapsulation(&ctxOut, ASN1_CONSTRUCTED_SEQUENCE);
    // Now move all the addedToContext to the start of the buffer
    MemoryCopy(out->addedToCertificate.t.buffer, ctxOut.buffer + ctxOut.offset,
               out->addedToCertificate.t.size);
#if CERTIFYX509_DEBUG
    DebugDumpBuffer(out->addedToCertificate.t.size, out->addedToCertificate.t.buffer,
                    "\naddedToCertificate");
#endif
    // only thing missing is the signature
    result = CryptSign(signKey, &in->inScheme, &out->tbsDigest, &out->signature);

    return result;
}
#endif // CC_CertifyX509
```

19 Ephemeral EC Keys

19.1 Introduction

The TPM generates keys that have different lifetimes. TPM keys in a hierarchy can be persistent for as long as the seed of the hierarchy is unchanged and these keys may be used multiple times. Other TPM-generated keys are only useful for a single operation. Some of these single-use keys are used in the command in which they are created. Examples of this use are TPM2_Duplicate() where an ephemeral key is created for a single pass key exchange with another TPM. However, there are other cases, such as anonymous attestation, where the protocol requires two passes where the public part of the ephemeral key is used outside of the TPM before the final command "consumes" the ephemeral key.

For these uses, TPM2_Commit() or TPM2_EC_Ephemeral() may be used to have the TPM create an ephemeral EC key and return the public part of the key for external use. Then in a subsequent command, the caller provides a reference to the ephemeral key so that the TPM can retrieve or recreate the associated private key.

When an ephemeral EC key is created, it is assigned a number and that number is returned to the caller as the identifier for the key. This number is not a handle. A handle is assigned to a key that may be context saved but these ephemeral EC keys may not be saved and do not have a full key context. When a subsequent command uses the ephemeral key, the caller provides the number of the ephemeral key. The TPM uses that number to either look up or recompute the associated private key. After the key is used, the TPM records the fact that the key has been used so that it cannot be used again.

As mentioned, the TPM can keep each assigned private ephemeral key in memory until it is used. However, this could consume a large amount of memory. To limit the memory size, the TPM is allowed to restrict the number of pending private keys – keys that have been allocated but not used.

NOTE The minimum number of ephemeral keys is determined by a platform specific specification

To further reduce the memory requirements for the ephemeral private keys, the TPM is allowed to use pseudo-random values for the ephemeral keys. Instead of keeping the full value of the key in memory, the TPM can use a counter as input to a KDF. Incrementing the counter will cause the TPM to generate a new pseudo-random value.

Using the counter to generate pseudo-random private ephemeral keys greatly simplifies tracking of key usage. When a counter value is used to create a key, a bit in an array may be set to indicate that the key use is pending. When the ephemeral key is consumed, the bit is cleared. This prevents the key from being used more than once.

Since the TPM is allowed to restrict the number of pending ephemeral keys, the array size can be limited. For example, a 128 bit array would allow 128 keys to be "pending".

The management of the array is described in greater detail in the *Split Operations* clause in Annex C of TPM 2.0 Part 1.

19.2 TPM2_Commit

19.2.1 General Description

TPM2_Commit() performs the first part of an ECC anonymous signing operation. The TPM will perform the point multiplications on the provided points and return intermediate signing values. The *signHandle* parameter shall refer to an ECC key and the signing scheme must be anonymous (TPM_RC_SCHEME).

NOTE 1 Currently, TPM_ALG_ECDAA is the only defined anonymous scheme.

NOTE 2 This command cannot be used with a sign+decrypt key because that type of key is required to have a scheme of TPM_ALG_NULL.

For this command, *p1*, *s2* and *y2* are optional parameters. If *s2* is an Empty Buffer, then the TPM shall return TPM_RC_SIZE if *y2* is not an Empty Buffer.

The algorithm is specified in the TPM 2.0 Part 1 Annex for ECC, TPM2_Commit().

19.2.2 Command and Response

Table 100 — TPM2_Commit Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_Commit
TPMI_DH_OBJECT	@signHandle	handle of the key that will be used in the signing operation Auth Index: 1 Auth Role: USER
TPM2B_ECC_POINT	P1	a point (M) on the curve used by *signHandle*
TPM2B_SENSITIVE_DATA	s2	octet array used to derive x-coordinate of a base point
TPM2B_ECC_PARAMETER	y2	y coordinate of the point associated with *s2*

Table 101 — TPM2_Commit Response

Type	Name	Description
TPM_ST	tag	see 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_ECC_POINT	K	ECC point $K \coloneqq [d_s](x2, y2)$
TPM2B_ECC_POINT	L	ECC point $L \coloneqq [r](x2, y2)$
TPM2B_ECC_POINT	E	ECC point $E \coloneqq [r]P1$
UINT16	counter	least-significant 16 bits of *commitCount*

19.2.3 Detailed Actions

```
#include "Tpm.h"
#include "Commit_fp.h"
#if CC_Commit // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_ATTRIBUTES	*keyHandle* references a restricted key that is not a signing key
TPM_RC_ECC_POINT	either *P1* or the point derived from *s2* is not on the curve of *keyHandle*
TPM_RC_HASH	invalid name algorithm in *keyHandle*
TPM_RC_KEY	*keyHandle* does not reference an ECC key
TPM_RC_SCHEME	the scheme of *keyHandle* is not an anonymous scheme
TPM_RC_NO_RESULT	*K*, *L* or *E* was a point at infinity; or failed to generate r value
TPM_RC_SIZE	*s2* is empty but *y2* is not or *s2* provided but *y2* is not

```
TPM_RC
TPM2_Commit(
    Commit_In       *in,            // IN: input parameter list
    Commit_Out      *out            // OUT: output parameter list
    )
{
    OBJECT                  *eccKey;
    TPMS_ECC_POINT           P2;
    TPMS_ECC_POINT          *pP2 = NULL;
    TPMS_ECC_POINT          *pP1 = NULL;
    TPM2B_ECC_PARAMETER      r;
    TPM2B_ECC_PARAMETER      p;
    TPM_RC                   result;
    TPMS_ECC_PARMS          *parms;

// Input Validation

    eccKey = HandleToObject(in->signHandle);
    parms = &eccKey->publicArea.parameters.eccDetail;

    // Input key must be an ECC key
    if(eccKey->publicArea.type != TPM_ALG_ECC)
        return TPM_RCS_KEY + RC_Commit_signHandle;

    // This command may only be used with a sign-only key using an anonymous
    // scheme.
    // NOTE: a sign + decrypt key has no scheme so it will not be an anonymous one
    // and an unrestricted sign key might no have a signing scheme but it can't
    // be use in Commit()
    if(!CryptIsSchemeAnonymous(parms->scheme.scheme))
        return TPM_RCS_SCHEME + RC_Commit_signHandle;

// Make sure that both parts of P2 are present if either is present
    if((in->s2.t.size == 0) != (in->y2.t.size == 0))
        return TPM_RCS_SIZE + RC_Commit_y2;

    // Get prime modulus for the curve. This is needed later but getting this now
    // allows confirmation that the curve exists.
    if(!CryptEccGetParameter(&p, 'p', parms->curveID))
        return TPM_RCS_KEY + RC_Commit_signHandle;

    // Get the random value that will be used in the point multiplications
```

```
    // Note: this does not commit the count.
    if(!CryptGenerateR(&r, NULL, parms->curveID, &eccKey->name))
        return TPM_RC_NO_RESULT;

    // Set up P2 if s2 and Y2 are provided
    if(in->s2.t.size != 0)
    {
        TPM2B_DIGEST            x2;

        pP2 = &P2;

        // copy y2 for P2
        P2.y = in->y2;

        // Compute x2  HnameAlg(s2) mod p
        //      do the hash operation on s2 with the size of curve 'p'
        x2.t.size = CryptHashBlock(eccKey->publicArea.nameAlg,
                                   in->s2.t.size,
                                   in->s2.t.buffer,
                                   sizeof(x2.t.buffer),
                                   x2.t.buffer);

        // If there were error returns in the hash routine, indicate a problem
        // with the hash algorithm selection
        if(x2.t.size == 0)
            return TPM_RCS_HASH + RC_Commit_signHandle;
        // The size of the remainder will be same as the size of p. DivideB() will
        // pad the results (leading zeros) if necessary to make the size the same
        P2.x.t.size = p.t.size;
        //  set p2.x = hash(s2) mod p
        if(DivideB(&x2.b, &p.b, NULL, &P2.x.b) != TPM_RC_SUCCESS)
            return TPM_RC_NO_RESULT;

        if(!CryptEccIsPointOnCurve(parms->curveID, pP2))
            return TPM_RCS_ECC_POINT + RC_Commit_s2;

        if(eccKey->attributes.publicOnly == SET)
            return TPM_RCS_KEY + RC_Commit_signHandle;
    }
    // If there is a P1, make sure that it is on the curve
    // NOTE: an "empty" point has two UINT16 values which are the size values
    // for each of the coordinates.
    if(in->P1.size > 4)
    {
        pP1 = &in->P1.point;
        if(!CryptEccIsPointOnCurve(parms->curveID, pP1))
            return TPM_RCS_ECC_POINT + RC_Commit_P1;
    }

    // Pass the parameters to CryptCommit.
    // The work is not done in-line because it does several point multiplies
    // with the same curve.  It saves work by not having to reload the curve
    // parameters multiple times.
    result = CryptEccCommitCompute(&out->K.point,
                                   &out->L.point,
                                   &out->E.point,
                                   parms->curveID,
                                   pP1,
                                   pP2,
                                   &eccKey->sensitive.sensitive.ecc,
                                   &r);
    if(result != TPM_RC_SUCCESS)
        return result;

    // The commit computation was successful so complete the commit by setting
    // the bit
```

```
    out->counter = CryptCommit();

    return TPM_RC_SUCCESS;
}
#endif // CC_Commit
```


19.3 TPM2_EC_Ephemeral

19.3.1 General Description

TPM2_EC_Ephemeral() creates an ephemeral key for use in a two-phase key exchange protocol.

The TPM will use the commit mechanism to assign an ephemeral key r and compute a public point $Q \coloneqq [r]G$ where G is the generator point associated with *curveID*.

19.3.2 Command and Response

Table 102 — TPM2_EC_Ephemeral Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit or encrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_EC_Ephemeral
TPMI_ECC_CURVE	curveID	The curve for the computed ephemeral point

Table 103 — TPM2_EC_Ephemeral Response

Type	Name	Description
TPM_ST	tag	see 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_ECC_POINT	Q	ephemeral public key $Q \coloneqq [r]G$
UINT16	counter	least-significant 16 bits of *commitCount*

19.3.3 Detailed Actions

```
#include "Tpm.h"
#include "EC_Ephemeral_fp.h"
#if CC_EC_Ephemeral  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_NO_RESULT	the TPM is not able to generate an *r* value

```
TPM_RC
TPM2_EC_Ephemeral(
    EC_Ephemeral_In      *in,            // IN: input parameter list
    EC_Ephemeral_Out     *out            // OUT: output parameter list
    )
{
    TPM2B_ECC_PARAMETER      r;
    TPM_RC                   result;
//
    do
    {
        // Get the random value that will be used in the point multiplications
        // Note: this does not commit the count.
        if(!CryptGenerateR(&r, NULL, in->curveID, NULL))
            return TPM_RC_NO_RESULT;
        // do a point multiply
        result = CryptEccPointMultiply(&out->Q.point, in->curveID, NULL, &r,
                                       NULL, NULL);
        // commit the count value if either the r value results in the point at
        // infinity or if the value is good. The commit on the r value for infinity
        // is so that the r value will be skipped.
        if((result == TPM_RC_SUCCESS) || (result == TPM_RC_NO_RESULT))
            out->counter = CryptCommit();
    } while(result == TPM_RC_NO_RESULT);

    return TPM_RC_SUCCESS;
}
#endif // CC_EC_Ephemeral
```


20 Signing and Signature Verification

20.1 TPM2_VerifySignature

20.1.1 General Description

This command uses loaded keys to validate a signature on a message with the message digest passed to the TPM.

If the signature check succeeds, then the TPM will produce a TPMT_TK_VERIFIED. Otherwise, the TPM shall return TPM_RC_SIGNATURE.

If the key is in the NULL hierarchy, then *digest* in the ticket will be the Empty Buffer.

NOTE 1 A valid ticket may be used in subsequent commands to provide proof to the TPM that the TPM has validated the signature over the message using the key referenced by *keyHandle*.

If *keyHandle* references an asymmetric key, only the public portion of the key needs to be loaded. If *keyHandle* references a symmetric key, both the public and private portions need to be loaded.

NOTE 2 The sensitive area of the symmetric object is required to allow verification of the symmetric signature (the HMAC).

20.1.2 Command and Response

Table 104 — TPM2_VerifySignature Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit or encrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_VerifySignature
TPMI_DH_OBJECT	keyHandle	handle of public key that will be used in the validation Auth Index: None
TPM2B_DIGEST	digest	digest of the signed message
TPMT_SIGNATURE	signature	signature to be tested

Table 105 — TPM2_VerifySignature Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPMT_TK_VERIFIED	validation	

20.1.3 Detailed Actions

```
#include "Tpm.h"
#include "VerifySignature_fp.h"
#if CC_VerifySignature  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_ATTRIBUTES	*keyHandle* does not reference a signing key
TPM_RC_SIGNATURE	signature is not genuine
TPM_RC_SCHEME	CryptValidateSignature()
TPM_RC_HANDLE	the input handle is references an HMAC key but the private portion is not loaded

```
TPM_RC
TPM2_VerifySignature(
    VerifySignature_In      *in,            // IN: input parameter list
    VerifySignature_Out     *out            // OUT: output parameter list
    )
{
    TPM_RC                   result;
    OBJECT                  *signObject = HandleToObject(in->keyHandle);
    TPMI_RH_HIERARCHY        hierarchy;

// Input Validation
    // The object to validate the signature must be a signing key.
    if(!IS_ATTRIBUTE(signObject->publicArea.objectAttributes, TPMA_OBJECT, sign))
        return TPM_RCS_ATTRIBUTES + RC_VerifySignature_keyHandle;

    // Validate Signature.  TPM_RC_SCHEME, TPM_RC_HANDLE or TPM_RC_SIGNATURE
    // error may be returned by CryptCVerifySignatrue()
    result = CryptValidateSignature(in->keyHandle, &in->digest, &in->signature);
    if(result != TPM_RC_SUCCESS)
        return RcSafeAddToResult(result, RC_VerifySignature_signature);

// Command Output

    hierarchy = GetHeriarchy(in->keyHandle);
    if(hierarchy == TPM_RH_NULL
       || signObject->publicArea.nameAlg == TPM_ALG_NULL)
    {
        // produce empty ticket if hierarchy is TPM_RH_NULL or nameAlg is
        // TPM_ALG_NULL
        out->validation.tag = TPM_ST_VERIFIED;
        out->validation.hierarchy = TPM_RH_NULL;
        out->validation.digest.t.size = 0;
    }
    else
    {
        // Compute ticket
        TicketComputeVerified(hierarchy, &in->digest, &signObject->name,
                              &out->validation);
    }

    return TPM_RC_SUCCESS;
}
#endif // CC_VerifySignature
```

20.2 TPM2_Sign

20.2.1 General Description

This command causes the TPM to sign an externally provided hash with the specified symmetric or asymmetric signing key.

NOTE 1 If *keyhandle* references an unrestricted signing key, a digest can be signed using either this command or an HMAC command.

If *keyHandle* references a restricted signing key, then *validation* shall be provided, indicating that the TPM performed the hash of the data and *validation* shall indicate that hashed data did not start with TPM_GENERATED_VALUE.

NOTE 2 If the hashed data did start with TPM_GENERATED_VALUE, then the validation will be a NULL ticket.

The *x509sign* attribute of keyHandle may not be SET (TPM_RC_ATTRIBUTES).

If the scheme of *keyHandle* is not TPM_ALG_NULL, then *inScheme* shall either be the same scheme as *keyHandle* or TPM_ALG_NULL. If the *sign* attribute is not SET in the key referenced by *handle* then the TPM shall return TPM_RC_KEY.

If the scheme of *keyHandle* is TPM_ALG_NULL, the TPM will sign using *inScheme*; otherwise, it will sign using the scheme of *keyHandle*.

NOTE 3 When the signing scheme uses a hash algorithm, the algorithm is defined in the qualifying data of the scheme. This is the same algorithm that is required to be used in producing *digest*. The size of *digest* must match that of the hash algorithm in the scheme.

If *inScheme* is not a valid signing scheme for the type of keyHandle (or TPM_ALG_NULL), then the TPM shall return TPM_RC_SCHEME.

If the scheme of *keyHandle* is an anonymous *scheme*, then *inScheme* shall have the same scheme algorithm as *keyHandle* and *inScheme* will contain a counter value that will be used in the signing process.

EXAMPLE For ECDAA, *inScheme.details.ecdaa.count* will contain the count value.

If *validation* is provided, then the hash algorithm used in computing the digest is required to be the hash algorithm specified in the scheme of *keyHandle* (TPM_RC_TICKET).

If the *validation* parameter is not the Empty Buffer*,* then it will be checked even if the key referenced by *keyHandle* is not a restricted signing key.

NOTE 4 If *keyHandle* is both a sign and decrypt key, *keyHandle* will have a scheme of TPM_ALG_NULL. If *validation* is provided, then it must be a NULL validation ticket or the ticket validation will fail.

20.2.2 Command and Response

Table 106 — TPM2_Sign Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_Sign
TPMI_DH_OBJECT	@keyHandle	Handle of key that will perform signing Auth Index: 1 Auth Role: USER
TPM2B_DIGEST	digest	digest to be signed
TPMT_SIG_SCHEME+	inScheme	signing scheme to use if the *scheme* for *keyHandle* is TPM_ALG_NULL
TPMT_TK_HASHCHECK	validation	proof that digest was created by the TPM If *keyHandle* is not a restricted signing key, then this may be a NULL Ticket with *tag* = TPM_ST_CHECKHASH.

Table 107 — TPM2_Sign Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPMT_SIGNATURE	signature	the signature

20.2.3 Detailed Actions

```
#include "Tpm.h"
#include "Sign_fp.h"
#if CC_Sign  // Conditional expansion of this file
#include "Attest_spt_fp.h"
```

Error Returns	Meaning
TPM_RC_BINDING	The public and private portions of the key are not properly bound.
TPM_RC_KEY	*signHandle* does not reference a signing key;
TPM_RC_SCHEME	the scheme is not compatible with sign key type, or input scheme is not compatible with default scheme, or the chosen scheme is not a valid sign scheme
TPM_RC_TICKET	*validation* is not a valid ticket
TPM_RC_VALUE	the value to sign is larger than allowed for the type of *keyHandle*

```
TPM_RC
TPM2_Sign(
    Sign_In         *in,            // IN: input parameter list
    Sign_Out        *out            // OUT: output parameter list
    )
{
    TPM_RC                   result;
    TPMT_TK_HASHCHECK        ticket;
    OBJECT                  *signObject = HandleToObject(in->keyHandle);
//
// Input Validation
    if(!IsSigningObject(signObject))
        return TPM_RCS_KEY + RC_Sign_keyHandle;

    // A key that will be used for x.509 signatures can't be used in TPM2_Sign().
    if(IS_ATTRIBUTE(signObject->publicArea.objectAttributes, TPMA_OBJECT, x509sign))
        return TPM_RCS_ATTRIBUTES + RC_Sign_keyHandle;

    // pick a scheme for sign.  If the input sign scheme is not compatible with
    // the default scheme, return an error.
    if(!CryptSelectSignScheme(signObject, &in->inScheme))
        return TPM_RCS_SCHEME + RC_Sign_inScheme;

    // If validation is provided, or the key is restricted, check the ticket
    if(in->validation.digest.t.size != 0
       || IS_ATTRIBUTE(signObject->publicArea.objectAttributes,
                       TPMA_OBJECT, restricted))
    {
        // Compute and compare ticket
        TicketComputeHashCheck(in->validation.hierarchy,
                               in->inScheme.details.any.hashAlg,
                               &in->digest, &ticket);

        if(!MemoryEqual2B(&in->validation.digest.b, &ticket.digest.b))
            return TPM_RCS_TICKET + RC_Sign_validation;
    }
    else
    // If we don't have a ticket, at least verify that the provided 'digest'
    // is the size of the scheme hashAlg digest.
    // NOTE: this does not guarantee that the 'digest' is actually produced using
    // the indicated hash algorithm, but at least it might be.
    {
        if(in->digest.t.size
```

```
            != CryptHashGetDigestSize(in->inScheme.details.any.hashAlg))
             return TPM_RCS_SIZE + RC_Sign_digest;
    }

// Command Output
    // Sign the hash. A TPM_RC_VALUE or TPM_RC_SCHEME
    // error may be returned at this point
    result = CryptSign(signObject, &in->inScheme, &in->digest, &out->signature);

    return result;
}
#endif // CC_Sign
```


21 Command Audit

21.1 Introduction

If a command has been selected for command audit, the command audit status will be updated when that command completes successfully. The digest is updated as:

$$commandAuditDigest_{new} \coloneqq \mathbf{H}_{auditAlg}(commandAuditDigest_{old} \mathbin{||} cpHash \mathbin{||} rpHash) \tag{5}$$

where

$\boldsymbol{H}_{auditAlg}$	hash function using the algorithm of the audit sequence
$commandAuditDigest$	accumulated digest
$cpHash$	the command parameter hash
$rpHash$	the response parameter hash

auditAlg, the hash algorithm, is set using TPM2_SetCommandCodeAuditStatus().

TPM2_Shutdown() cannot be audited but TPM2_Startup() can be audited. If the *cpHash* of the TPM2_Startup() is TPM_SU_STATE, that would indicate that a TPM2_Shutdown() had been successfully executed.

TPM2_SetCommandCodeAuditStatus() is always audited, except when it is used to change *auditAlg*.

If the TPM is in Failure mode, command audit is not functional.

21.2 TPM2_SetCommandCodeAuditStatus

21.2.1 General Description

This command may be used by the Privacy Administrator or platform to change the audit status of a command or to set the hash algorithm used for the audit digest, but not both at the same time.

If the *auditAlg* parameter is a supported hash algorithm and not the same as the current algorithm, then the TPM will check both *setList* and *clearList* are empty (zero length). If so, then the algorithm is changed, and the audit digest is cleared. If *auditAlg* is TPM_ALG_NULL or the same as the current algorithm, then the algorithm and audit digest are unchanged and the *setList* and *clearList* will be processed.

NOTE 1 Because the audit digest is cleared, the audit counter will increment the next time that an audited command is executed.

Use of TPM2_SetCommandCodeAuditStatus() to change the list of audited commands is an audited event. If TPM_CC_SetCommandCodeAuditStatus is in *clearList,* the fact that it is in *clearList* is ignored.

NOTE 2 Use of this command to change the audit hash algorithm is not audited and the digest is reset when the command completes. The change in the audit hash algorithm is the evidence that this command was used to change the algorithm.

The commands in *setList* indicate the commands to be added to the list of audited commands and the commands in *clearList* indicate the commands that will no longer be audited. It is not an error if a command in *setList* is already audited or is not implemented. It is not an error if a command in *clearList* is not currently being audited or is not implemented.

If a command code is in both *setList* and *clearList*, then it will not be audited (that is, *setList* shall be processed first).

21.2.2 Command and Response

Table 108 — TPM2_SetCommandCodeAuditStatus Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_SetCommandCodeAuditStatus {NV}
TPMI_RH_PROVISION	@auth	TPM_RH_OWNER or TPM_RH_PLATFORM+{PP} Auth Index: 1 Auth Role: USER
TPMI_ALG_HASH+	auditAlg	hash algorithm for the audit digest; if TPM_ALG_NULL, then the hash is not changed
TPML_CC	setList	list of commands that will be added to those that will be audited
TPML_CC	clearList	list of commands that will no longer be audited

Table 109 — TPM2_SetCommandCodeAuditStatus Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

21.2.3 Detailed Actions

```
#include "Tpm.h"
#include "SetCommandCodeAuditStatus_fp.h"
#if CC_SetCommandCodeAuditStatus  // Conditional expansion of this file
TPM_RC
TPM2_SetCommandCodeAuditStatus(
    SetCommandCodeAuditStatus_In    *in             // IN: input parameter list
    )
{

    // The command needs NV update.  Check if NV is available.
    // A TPM_RC_NV_UNAVAILABLE or TPM_RC_NV_RATE error may be returned at
    // this point
    RETURN_IF_NV_IS_NOT_AVAILABLE;

// Internal Data Update

    // Update hash algorithm
    if(in->auditAlg != TPM_ALG_NULL && in->auditAlg != gp.auditHashAlg)
    {
        // Can't change the algorithm and command list at the same time
        if(in->setList.count != 0 || in->clearList.count != 0)
            return TPM_RCS_VALUE + RC_SetCommandCodeAuditStatus_auditAlg;

        // Change the hash algorithm for audit
        gp.auditHashAlg = in->auditAlg;

        // Set the digest size to a unique value that indicates that the digest
        // algorithm has been changed. The size will be cleared to zero in the
        // command audit processing on exit.
        gr.commandAuditDigest.t.size = 1;

        // Save the change of command audit data (this sets g_updateNV so that NV
        // will be updated on exit.)
        NV_SYNC_PERSISTENT(auditHashAlg);
    }
    else
    {
        UINT32          i;
        BOOL            changed = FALSE;

        // Process set list
        for(i = 0; i < in->setList.count; i++)

            // If change is made in CommandAuditSet, set changed flag
            if(CommandAuditSet(in->setList.commandCodes[i]))
                changed = TRUE;

        // Process clear list
        for(i = 0; i < in->clearList.count; i++)
            // If change is made in CommandAuditClear, set changed flag
            if(CommandAuditClear(in->clearList.commandCodes[i]))
                changed = TRUE;

        // if change was made to command list, update NV
        if(changed)
            // this sets g_updateNV so that NV will be updated on exit.
            NV_SYNC_PERSISTENT(auditCommands);
    }

    return TPM_RC_SUCCESS;
}
#endif // CC_SetCommandCodeAuditStatus
```

22 Integrity Collection (PCR)

22.1 Introduction

In TPM 1.2, an Event was hashed using SHA-1 and then the 20-octet digest was extended to a PCR using TPM_Extend(). This specification allows the use of multiple PCR at a given Index, each using a different hash algorithm. Rather than require that the external software generate multiple hashes of the Event with each being extended to a different PCR, the Event data may be sent to the TPM for hashing. This ensures that the resulting digests will properly reflect the algorithms chosen for the PCR even if the calling software is unable to implement the hash algorithm.

NOTE 1 There is continued support for software hashing of events with TPM2_PCR_Extend().

To support recording of an Event that is larger than the TPM input buffer, the caller may use the command sequence described in clause 1.

Change to a PCR requires authorization. The authorization may be with either an authorization value or an authorization policy. The platform-specific specifications determine which PCR may be controlled by policy. All other PCR are controlled by authorization.

If a PCR may be associated with a policy, then the algorithm ID of that policy determines whether the policy is to be applied. If the algorithm ID is not TPM_ALG_NULL, then the policy digest associated with the PCR must match the *policySession→policyDigest* in a policy session. If the algorithm ID is TPM_ALG_NULL, then no policy is present and the authorization requires an EmptyAuth.

If a platform-specific specification indicates that PCR are grouped, then all the PCR in the group use the same authorization policy or authorization value.

pcrUpdateCounter counter will be incremented on the successful completion of any command that modifies (Extends or resets) a PCR unless the platform-specific specification explicitly excludes the PCR from being counted.

NOTE 2 If a command causes PCR in multiple banks to change, the PCR Update Counter must be incremented once for each bank. The commands that extend PCR are: TPM2_PCR_Extend, TPM2_PCR_Event, and TPM2_EventSequenceComplete.

If a command resets PCR in multiple banks, the PCR Update Counter must be incremented only once. The commands that reset PCR are: TPM2_PCR_Reset, and TPM2_Startup.

A platform-specific specification may designate a set of PCR that are under control of the TCB. These PCR may not be modified without the proper authorization. Updates of these PCR shall not cause the PCR Update Counter to increment.

EXAMPLE Updates of the TCB PCR will not cause the PCR update counter to increment because these PCR are changed at the whim of the TCB and may not represent the trust state of the platform.

22.2 TPM2_PCR_Extend

22.2.1 General Description

This command is used to cause an update to the indicated PCR. The *digests* parameter contains one or more tagged digest values identified by an algorithm ID. For each digest, the PCR associated with *pcrHandle* is Extended into the bank identified by the tag (*hashAlg*).

EXAMPLE A SHA1 digest would be Extended into the SHA1 bank and a SHA256 digest would be Extended into the SHA256 bank.

For each list entry, the TPM will check to see if *pcrNum* is implemented for that algorithm. If so, the TPM shall perform the following operation:

$$PCR.digest_{new}[pcrNum][alg] \coloneqq \mathbf{H}_{alg}(PCR.digest_{old}[pcrNum][alg] \mathbin{||} data[alg].buffer) \quad (6)$$

where

$\mathbf{H}_{alg}()$	hash function using the hash algorithm associated with the PCR instance
$PCR.digest$	the digest value in a PCR
$pcrNum$	the PCR numeric selector (*pcrHandle*)
alg	the PCR algorithm selector for the digest
$data[alg].buffer$	the bank-specific data to be extended

If no digest value is specified for a bank, then the PCR in that bank is not modified.

NOTE 1 This allows consistent operation of the digests list for all of the Event recording commands.

If a digest is present and the PCR in that bank is not implemented, the digest value is not used.

NOTE 2 If the caller includes digests for algorithms that are not implemented, then the TPM will fail the call because the unmarshalling of *digests* will fail. Each of the entries in the list is a TPMT_HA, which is a hash algorithm followed by a digest. If the algorithm is not implemented, unmarshalling of the *hashAlg* will fail and the TPM will return TPM_RC_HASH.

If the TPM unmarshals the *hashAlg* of a list entry and the unmarshaled value is not a hash algorithm implemented on the TPM, the TPM shall return TPM_RC_HASH.

The *pcrHandle* parameter is allowed to reference TPM_RH_NULL. If so, the input parameters are processed but no action is taken by the TPM. This permits the caller to probe for implemented hash algorithms as an alternative to TPM2_GetCapability.

NOTE 3 This command allows a list of digests so that PCR in all banks may be updated in a single command. While the semantics of this command allow multiple extends to a single PCR bank, this is not the preferred use and the limit on the number of entries in the list make this use somewhat impractical.

22.2.2 Command and Response

Table 110 — TPM2_PCR_Extend Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PCR_Extend {NV}
TPMI_DH_PCR+	@pcrHandle	handle of the PCR Auth Handle: 1 Auth Role: USER
TPML_DIGEST_VALUES	digests	list of tagged digest values to be extended

Table 111 — TPM2_PCR_Extend Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	.

22.2.3 Detailed Actions

```
#include "Tpm.h"
#include "PCR_Extend_fp.h"
#if CC_PCR_Extend  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_LOCALITY	current command locality is not allowed to extend the PCR referenced by *pcrHandle*

```
TPM_RC
TPM2_PCR_Extend(
    PCR_Extend_In   *in             // IN: input parameter list
    )
{
    UINT32              i;

// Input Validation

    // NOTE: This function assumes that the unmarshaling function for 'digests' will
    // have validated that all of the indicated hash algorithms are valid. If the
    // hash algorithms are correct, the unmarshaling code will unmarshal a digest
    // of the size indicated by the hash algorithm. If the overall size is not
    // consistent, the unmarshaling code will run out of input data or have input
    // data left over. In either case, it will cause an unmarshaling error and this
    // function will not be called.

    // For NULL handle, do nothing and return success
    if(in->pcrHandle == TPM_RH_NULL)
        return TPM_RC_SUCCESS;

    // Check if the extend operation is allowed by the current command locality
    if(!PCRIsExtendAllowed(in->pcrHandle))
        return TPM_RC_LOCALITY;

    // If PCR is state saved and we need to update orderlyState, check NV
    // availability
    if(PCRIsStateSaved(in->pcrHandle))
        RETURN_IF_ORDERLY;

// Internal Data Update

    // Iterate input digest list to extend
    for(i = 0; i < in->digests.count; i++)
    {
        PCRExtend(in->pcrHandle, in->digests.digests[i].hashAlg,
                  CryptHashGetDigestSize(in->digests.digests[i].hashAlg),
                  (BYTE *)&in->digests.digests[i].digest);
    }

    return TPM_RC_SUCCESS;
}
#endif // CC_PCR_Extend
```

22.3 TPM2_PCR_Event

22.3.1 General Description

This command is used to cause an update to the indicated PCR.

The data in *eventData* is hashed using the hash algorithm associated with each bank in which the indicated PCR has been allocated. After the data is hashed, the *digests* list is returned. If the *pcrHandle* references an implemented PCR and not TPM_RH_NULL, the *digests* list is processed as in TPM2_PCR_Extend().

A TPM shall support an *Event.size* of zero through 1,024 inclusive (*Event.size* is an octet count). An *Event.size* of zero indicates that there is no data but the indicated operations will still occur,

EXAMPLE 1 If the command implements PCR[2] in a SHA1 bank and a SHA256 bank, then an extend to PCR[2] will cause *eventData* to be hashed twice, once with SHA1 and once with SHA256. The SHA1 hash of *eventData* will be Extended to PCR[2] in the SHA1 bank and the SHA256 hash of *eventData* will be Extended to PCR[2] of the SHA256 bank.

On successful command completion, *digests* will contain the list of tagged digests of *eventData* that was computed in preparation for extending the data into the PCR. At the option of the TPM, the list may contain a digest for each bank, or it may only contain a digest for each bank in which *pcrHandle* is extant. If *pcrHandle* is TPM_RH_NULL, the TPM may return either an empty list or a digest for each bank.

EXAMPLE 2 Assume a TPM that implements a SHA1 bank and a SHA256 bank and that PCR[22] is only implemented in the SHA1 bank. If *pcrHandle* references PCR[22], then *digests* may contain either a SHA1 and a SHA256 digest or just a SHA1 digest.

22.3.2 Command and Response

Table 112 — TPM2_PCR_Event Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PCR_Event {NV}
TPMI_DH_PCR+	@pcrHandle	Handle of the PCR Auth Handle: 1 Auth Role: USER
TPM2B_EVENT	eventData	Event data in sized buffer

Table 113 — TPM2_PCR_Event Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	.
TPML_DIGEST_VALUES	digests	

22.3.3 Detailed Actions

```
#include "Tpm.h"
#include "PCR_Event_fp.h"
#if CC_PCR_Event // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_LOCALITY	current command locality is not allowed to extend the PCR referenced by *pcrHandle*

```
TPM_RC
TPM2_PCR_Event(
    PCR_Event_In    *in,            // IN: input parameter list
    PCR_Event_Out   *out            // OUT: output parameter list
    )
{
    HASH_STATE          hashState;
    UINT32              i;
    UINT16              size;

// Input Validation

    // If a PCR extend is required
    if(in->pcrHandle != TPM_RH_NULL)
    {
        // If the PCR is not allow to extend, return error
        if(!PCRIsExtendAllowed(in->pcrHandle))
            return TPM_RC_LOCALITY;

        // If PCR is state saved and we need to update orderlyState, check NV
        // availability
        if(PCRIsStateSaved(in->pcrHandle))
            RETURN_IF_ORDERLY;
    }

// Internal Data Update

    out->digests.count = HASH_COUNT;

    // Iterate supported PCR bank algorithms to extend
    for(i = 0; i < HASH_COUNT; i++)
    {
        TPM_ALG_ID  hash = CryptHashGetAlgByIndex(i);
        out->digests.digests[i].hashAlg = hash;
        size = CryptHashStart(&hashState, hash);
        CryptDigestUpdate2B(&hashState, &in->eventData.b);
        CryptHashEnd(&hashState, size,
                     (BYTE *)&out->digests.digests[i].digest);
        if(in->pcrHandle != TPM_RH_NULL)
            PCRExtend(in->pcrHandle, hash, size,
                      (BYTE *)&out->digests.digests[i].digest);
    }

    return TPM_RC_SUCCESS;
}
#endif // CC_PCR_Event
```

22.4 TPM2_PCR_Read

22.4.1 General Description

This command returns the values of all PCR specified in *pcrSelectionIn*.

The TPM will process the list of TPMS_PCR_SELECTION in *pcrSelectionIn* in order. Within each TPMS_PCR_SELECTION, the TPM will process the bits in the *pcrSelect* array in ascending PCR order (see TPM 2.0 Part 1, *Selecting Multiple PCR*). If a bit is SET, and the indicated PCR is present, then the TPM will add the digest of the PCR to the list of values to be returned in *pcrValues.*

The TPM will continue processing bits until all have been processed or until *pcrValues* would be too large to fit into the output buffer if additional values were added.

The returned *pcrSelectionOut* will have a bit SET in its *pcrSelect* structures for each value present in *pcrValues*.

The current value of the PCR Update Counter is returned in *pcrUpdateCounter*.

The returned list may be empty if none of the selected PCR are implemented.

NOTE If no PCR are returned from a bank, the selector for the bank will be present in *pcrSelectionOut*.

No authorization is required to read a PCR and any implemented PCR may be read from any locality.

22.4.2 Command and Response

Table 114 — TPM2_PCR_Read Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PCR_Read
TPML_PCR_SELECTION	pcrSelectionIn	The selection of PCR to read

Table 115 — TPM2_PCR_Read Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
UINT32	pcrUpdateCounter	the current value of the PCR update counter
TPML_PCR_SELECTION	pcrSelectionOut	the PCR in the returned list
TPML_DIGEST	pcrValues	the contents of the PCR indicated in *pcrSelectOut->pcrSelection[]* as tagged digests

22.4.3 Detailed Actions

```
#include "Tpm.h"
#include "PCR_Read_fp.h"
#if CC_PCR_Read  // Conditional expansion of this file
TPM_RC
TPM2_PCR_Read(
    PCR_Read_In     *in,            // IN: input parameter list
    PCR_Read_Out    *out            // OUT: output parameter list
    )
{
// Command Output

    // Call PCR read function.  input pcrSelectionIn parameter could be changed
    // to reflect the actual PCR being returned
    PCRRead(&in->pcrSelectionIn, &out->pcrValues, &out->pcrUpdateCounter);

    out->pcrSelectionOut = in->pcrSelectionIn;

    return TPM_RC_SUCCESS;
}
#endif // CC_PCR_Read
```


22.5 TPM2_PCR_Allocate

22.5.1 General Description

This command is used to set the desired PCR allocation of PCR and algorithms. This command requires Platform Authorization.

The TPM will evaluate the request and, if sufficient memory is available for the requested allocation, the TPM will store the allocation request for use during the next _TPM_Init operation. The PCR allocation in place when this command is executed will be retained until the next _TPM_Init. If this command is received multiple times before a _TPM_Init, each one overwrites the previous stored allocation.

This command will only change the allocations of banks that are listed in *pcrAllocation*.

EXAMPLE 1 If a TPM supports SHA1 and SHA256, then it maintains an allocation for two banks (one of which could be empty). If *pcrAllocation* only has a selector for the SHA1 bank, then only the allocation of the SHA1 bank will be changed and the SHA256 bank will remain unchanged. To change the allocation of a TPM from 24 SHA1 PCR and no SHA256 PCR to 24 SHA256 PCR and no SHA1 PCR, the *pcrAllocation* would have to have two selections: one for the empty SHA1 bank and one for the SHA256 bank with 24 PCR.

If a bank is listed more than once, then the last selection in the *pcrAllocation* list is the one that the TPM will attempt to allocate.

NOTE 1 This does not mean to imply that *pcrAllocation.count* can exceed HASH_COUNT, the number of digests implemented in the TPM.

EXAMPLE 2 If HASH_COUNT is 2, *pcrAllocation* can specify SHA-256 twice, and the second one is used. However, if SHA_256 is specified three times, the unmarshaling may fail and the TPM may return an error.

This command shall not allocate more PCR in any bank than there are PCR attribute definitions. The PCR attribute definitions indicate how a PCR is to be managed – if it is resettable, the locality for update, etc. In the response to this command, the TPM returns the maximum number of PCR allowed for any bank.

When PCR are allocated, if DRTM_PCR is defined, the resulting allocation must have at least one bank with the D-RTM PCR allocated. If HCRTM_PCR is defined, the resulting allocation must have at least one bank with the HCRTM_PCR allocated. If not, the TPM returns TPM_RC_PCR.

The TPM may return TPM_RC_SUCCESS even though the request fails. This is to allow the TPM to return information about the size needed for the requested allocation and the size available. If the *sizeNeeded* parameter in the return is less than or equal to the *sizeAvailable* parameter, then the *allocationSuccess* parameter will be YES. Alternatively, if the request fails, The TPM may return TPM_RC_NO_RESULT.

NOTE 2 An example for this type of failure is a TPM that can only support one bank at a time and cannot support arbitrary distribution of PCR among banks.

After this command, TPM2_Shutdown() is only allowed to have a *startupType* equal to TPM_SU_CLEAR until after the next _TPM_Init.

NOTE 3 Even if this command does not cause the PCR allocation to change, the TPM cannot have its state saved. This is done in order to simplify the implementation. There is no need to optimize this command as it is not expected to be used more than once in the lifetime of the TPM (it can be used any number of times but there is no justification for optimization).

22.5.2 Command and Response

Table 116 — TPM2_PCR_Allocate Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PCR_Allocate {NV}
TPMI_RH_PLATFORM	@authHandle	TPM_RH_PLATFORM+{PP} Auth Index: 1 Auth Role: USER
TPML_PCR_SELECTION	pcrAllocation	the requested allocation

Table 117 — TPM2_PCR_Allocate Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPMI_YES_NO	allocationSuccess	YES if the allocation succeeded
UINT32	maxPCR	maximum number of PCR that may be in a bank
UINT32	sizeNeeded	number of octets required to satisfy the request
UINT32	sizeAvailable	Number of octets available. Computed before the allocation.

22.5.3 Detailed Actions

```
#include "Tpm.h"
#include "PCR_Allocate_fp.h"
#if CC_PCR_Allocate  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_PCR	the allocation did not have required PCR
TPM_RC_NV_UNAVAILABLE	NV is not accessible
TPM_RC_NV_RATE	NV is in a rate-limiting mode

```
TPM_RC
TPM2_PCR_Allocate(
    PCR_Allocate_In     *in,            // IN: input parameter list
    PCR_Allocate_Out    *out            // OUT: output parameter list
    )
{
    TPM_RC      result;

    // The command needs NV update.  Check if NV is available.
    // A TPM_RC_NV_UNAVAILABLE or TPM_RC_NV_RATE error may be returned at
    // this point.
    // Note: These codes are not listed in the return values above because it is
    // an implementation choice to check in this routine rather than in a common
    // function that is called before these actions are called. These return values
    // are described in the Response Code section of Part 3.
    RETURN_IF_NV_IS_NOT_AVAILABLE;

// Command Output

    // Call PCR Allocation function.
    result = PCRAllocate(&in->pcrAllocation, &out->maxPCR,
                         &out->sizeNeeded, &out->sizeAvailable);
    if(result == TPM_RC_PCR)
        return result;

    //
    out->allocationSuccess = (result == TPM_RC_SUCCESS);

    // if re-configuration succeeds, set the flag to indicate PCR configuration is
    // going to be changed in next boot
    if(out->allocationSuccess == YES)
        g_pcrReConfig = TRUE;

    return TPM_RC_SUCCESS;
}
#endif // CC_PCR_Allocate
```

22.6 TPM2_PCR_SetAuthPolicy

22.6.1 General Description

This command is used to associate a policy with a PCR or group of PCR. The policy determines the conditions under which a PCR may be extended or reset.

A policy may only be associated with a PCR that has been defined by a platform-specific specification as allowing a policy. If the TPM implementation does not allow a policy for *pcrNum*, the TPM shall return TPM_RC_VALUE.

A platform-specific specification may group PCR so that they share a common policy. In such case, a *pcrNum* that selects any of the PCR in the group will change the policy for all PCR in the group.

The policy setting is persistent and may only be changed by TPM2_PCR_SetAuthPolicy() or by TPM2_ChangePPS().

Before this command is first executed on a TPM or after TPM2_ChangePPS(), the access control on the PCR will be set to the default value defined in the platform-specific specification.

NOTE 1 It is expected that the typical default will be with the policy hash set to TPM_ALG_NULL and an Empty Buffer for the *authPolicy* value. This will allow an *EmptyAuth* to be used as the authorization value.

If the size of the data buffer in *authPolicy* is not the size of a digest produced by *hashAlg*, the TPM shall return TPM_RC_SIZE.

NOTE 2 If *hashAlg* is TPM_ALG_NULL, then the size is required to be zero.

This command requires platformAuth/platformPolicy.

NOTE 3 If the PCR is in multiple policy sets, the policy will be changed in only one set. The set that is changed will be implementation dependent.

22.6.2 Command and Response

Table 118 — TPM2_PCR_SetAuthPolicy Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PCR_SetAuthPolicy {NV}
TPMI_RH_PLATFORM	@authHandle	TPM_RH_PLATFORM+{PP} Auth Index: 1 Auth Role: USER
TPM2B_DIGEST	authPolicy	the desired *authPolicy*
TPMI_ALG_HASH+	hashAlg	the hash algorithm of the policy
TPMI_DH_PCR	pcrNum	the PCR for which the policy is to be set

Table 119 — TPM2_PCR_SetAuthPolicy Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

22.6.3 Detailed Actions

```
#include "Tpm.h"
#include "PCR_SetAuthPolicy_fp.h"
#if CC_PCR_SetAuthPolicy  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_SIZE	size of *authPolicy* is not the size of a digest produced by *policyDigest*
TPM_RC_VALUE	PCR referenced by *pcrNum* is not a member of a PCR policy group

```
TPM_RC
TPM2_PCR_SetAuthPolicy(
    PCR_SetAuthPolicy_In    *in             // IN: input parameter list
    )
{
    UINT32      groupIndex;

    // The command needs NV update.  Check if NV is available.
    // A TPM_RC_NV_UNAVAILABLE or TPM_RC_NV_RATE error may be returned at
    // this point
    RETURN_IF_NV_IS_NOT_AVAILABLE;

// Input Validation:

    // Check the authPolicy consistent with hash algorithm
    if(in->authPolicy.t.size != CryptHashGetDigestSize(in->hashAlg))
        return TPM_RCS_SIZE + RC_PCR_SetAuthPolicy_authPolicy;

    // If PCR does not belong to a policy group, return TPM_RC_VALUE
    if(!PCRBelongsPolicyGroup(in->pcrNum, &groupIndex))
        return TPM_RCS_VALUE + RC_PCR_SetAuthPolicy_pcrNum;

// Internal Data Update

    // Set PCR policy
    gp.pcrPolicies.hashAlg[groupIndex] = in->hashAlg;
    gp.pcrPolicies.policy[groupIndex] = in->authPolicy;

    // Save new policy to NV
    NV_SYNC_PERSISTENT(pcrPolicies);

    return TPM_RC_SUCCESS;
}
#endif // CC_PCR_SetAuthPolicy
```

22.7 TPM2_PCR_SetAuthValue

22.7.1 General Description

This command changes the *authValue* of a PCR or group of PCR.

An *authValue* may only be associated with a PCR that has been defined by a platform-specific specification as allowing an authorization value. If the TPM implementation does not allow an authorization for *pcrNum*, the TPM shall return TPM_RC_VALUE. A platform-specific specification may group PCR so that they share a common authorization value. In such case, a *pcrNum* that selects any of the PCR in the group will change the *authValue* value for all PCR in the group.

The authorization setting is set to EmptyAuth on each STARTUP(CLEAR) or by TPM2_Clear(). The authorization setting is preserved by SHUTDOWN(STATE).

22.7.2 Command and Response

Table 120 — TPM2_PCR_SetAuthValue Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PCR_SetAuthValue
TPMI_DH_PCR	@pcrHandle	handle for a PCR that may have an authorization value set Auth Index: 1 Auth Role: USER
TPM2B_DIGEST	auth	the desired authorization value

Table 121 — TPM2_PCR_SetAuthValue Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

22.7.3 Detailed Actions

```
#include "Tpm.h"
#include "PCR_SetAuthValue_fp.h"
#if CC_PCR_SetAuthValue  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_VALUE	PCR referenced by *pcrHandle* is not a member of a PCR authorization group

```
TPM_RC
TPM2_PCR_SetAuthValue(
    PCR_SetAuthValue_In     *in             // IN: input parameter list
    )
{
    UINT32      groupIndex;
// Input Validation:

    // If PCR does not belong to an auth group, return TPM_RC_VALUE
    if(!PCRBelongsAuthGroup(in->pcrHandle, &groupIndex))
        return TPM_RC_VALUE;

    // The command may cause the orderlyState to be cleared due to the update of
    // state clear data.  If this is the case, Check if NV is available.
    // A TPM_RC_NV_UNAVAILABLE or TPM_RC_NV_RATE error may be returned at
    // this point
    RETURN_IF_ORDERLY;

// Internal Data Update

    // Set PCR authValue
    MemoryRemoveTrailingZeros(&in->auth);
    gc.pcrAuthValues.auth[groupIndex] = in->auth;

    return TPM_RC_SUCCESS;
}
#endif // CC_PCR_SetAuthValue
```


22.8 TPM2_PCR_Reset

22.8.1 General Description

If the attribute of a PCR allows the PCR to be reset and proper authorization is provided, then this command may be used to set the PCR in all banks to zero. The attributes of the PCR may restrict the locality that can perform the reset operation.

NOTE 1 The definition of TPMI_DH_PCR in TPM 2.0 Part 2 indicates that if pcrHandle is out of the allowed range for PCR, then the appropriate return value is TPM_RC_VALUE.

If *pcrHandle* references a PCR that cannot be reset, the TPM shall return TPM_RC_LOCALITY.

NOTE 2 TPM_RC_LOCALITY is returned because the reset attributes are defined on a per-locality basis.

22.8.2 Command and Response

Table 122 — TPM2_PCR_Reset Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PCR_Reset {NV}
TPMI_DH_PCR	@pcrHandle	the PCR to reset Auth Index: 1 Auth Role: USER

Table 123 — TPM2_PCR_Reset Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

22.8.3 Detailed Actions

```
#include "Tpm.h"
#include "PCR_Reset_fp.h"
#if CC_PCR_Reset  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_LOCALITY	current command locality is not allowed to reset the PCR referenced by *pcrHandle*

```
TPM_RC
TPM2_PCR_Reset(
    PCR_Reset_In    *in             // IN: input parameter list
    )
{
// Input Validation

    // Check if the reset operation is allowed by the current command locality
    if(!PCRIsResetAllowed(in->pcrHandle))
        return TPM_RC_LOCALITY;

    // If PCR is state saved and we need to update orderlyState, check NV
    // availability
    if(PCRIsStateSaved(in->pcrHandle))
        RETURN_IF_ORDERLY;

// Internal Data Update

    // Reset selected PCR in all banks to 0
    PCRSetValue(in->pcrHandle, 0);

    // Indicate that the PCR changed so that pcrCounter will be incremented if
    // necessary.
    PCRChanged(in->pcrHandle);

    return TPM_RC_SUCCESS;
}
#endif // CC_PCR_Reset
```

22.9 _TPM_Hash_Start

22.9.1 Description

This indication from the TPM interface indicates the start of an H-CRTM measurement sequence. On receipt of this indication, the TPM will initialize an H-CRTM Event Sequence context.

If no object memory is available for creation of the sequence context, the TPM will flush the context of an object so that creation of the sequence context will always succeed.

A platform-specific specification may allow this indication before TPM2_Startup().

NOTE If this indication occurs after TPM2_Startup(), it is the responsibility of software to ensure that an object context slot is available or to deal with the consequences of having the TPM select an arbitrary object to be flushed. If this indication occurs before TPM2_Startup() then all context slots are available.

22.9.2 Detailed Actions

```
#include "Tpm.h"
```

This function is called to process a _TPM_Hash_Start() indication.

```
LIB_EXPORT void
_TPM_Hash_Start(
    void
    )
{
    TPM_RC              result;
    TPMI_DH_OBJECT      handle;

    // If a DRTM sequence object exists, free it up
    if(g_DRTMHandle != TPM_RH_UNASSIGNED)
    {
        FlushObject(g_DRTMHandle);
        g_DRTMHandle = TPM_RH_UNASSIGNED;
    }

    // Create an event sequence object and store the handle in global
    // g_DRTMHandle. A TPM_RC_OBJECT_MEMORY error may be returned at this point
    // The NULL value for the first parameter will cause the sequence structure to
    // be allocated without being set as present. This keeps the sequence from
    // being left behind if the sequence is terminated early.
    result = ObjectCreateEventSequence(NULL, &g_DRTMHandle);

    // If a free slot was not available, then free up a slot.
    if(result != TPM_RC_SUCCESS)
    {
        // An implementation does not need to have a fixed relationship between
        // slot numbers and handle numbers. To handle the general case, scan for
        // a handle that is assigned and free it for the DRTM sequence.
        // In the reference implementation, the relationship between handles and
        // slots is fixed. So, if the call to ObjectCreateEvenSequence()
        // failed indicating that all slots are occupied, then the first handle we
        // are going to check (TRANSIENT_FIRST) will be occupied. It will be freed
        // so that it can be assigned for use as the DRTM sequence object.
        for(handle = TRANSIENT_FIRST; handle < TRANSIENT_LAST; handle++)
        {
            // try to flush the first object
            if(IsObjectPresent(handle))
                break;
        }
        // If the first call to find a slot fails but none of the slots is occupied
        // then there's a big problem
        pAssert(handle < TRANSIENT_LAST);

        // Free the slot
        FlushObject(handle);

        // Try to create an event sequence object again.  This time, we must
        // succeed.
        result = ObjectCreateEventSequence(NULL, &g_DRTMHandle);
        if(result != TPM_RC_SUCCESS)
            FAIL(FATAL_ERROR_INTERNAL);
    }

    return;
}
```

22.10 _TPM_Hash_Data

22.10.1 Description

This indication from the TPM interface indicates arrival of one or more octets of data that are to be included in the H-CRTM Event Sequence sequence context created by the _TPM_Hash_Start indication. The context holds data for each hash algorithm for each PCR bank implemented on the TPM.

If no H-CRTM Event Sequence context exists, this indication is discarded and no other action is performed.

22.10.2 Detailed Actions

```
#include "Tpm.h"
```

This function is called to process a _TPM_Hash_Data() indication.

```
LIB_EXPORT void
_TPM_Hash_Data(
    uint32_t         dataSize,      // IN: size of data to be extend
    unsigned char   *data           // IN: data buffer
    )
{
    UINT32           i;
    HASH_OBJECT     *hashObject;
    TPMI_DH_PCR      pcrHandle = TPMIsStarted()
        ? PCR_FIRST + DRTM_PCR : PCR_FIRST + HCRTM_PCR;

// If there is no DRTM sequence object, then _TPM_Hash_Start
// was not called so this function returns without doing
// anything.
    if(g_DRTMHandle == TPM_RH_UNASSIGNED)
        return;

    hashObject = (HASH_OBJECT *)HandleToObject(g_DRTMHandle);
    pAssert(hashObject->attributes.eventSeq);

    // For each of the implemented hash algorithms, update the digest with the
    // data provided.
    for(i = 0; i < HASH_COUNT; i++)
    {
        // make sure that the PCR is implemented for this algorithm
        if(PcrIsAllocated(pcrHandle,
                          hashObject->state.hashState[i].hashAlg))
          // Update sequence object
            CryptDigestUpdate(&hashObject->state.hashState[i], dataSize, data);
    }

    return;
}
```

22.11 _TPM_Hash_End

22.11.1 Description

This indication from the TPM interface indicates the end of the H-CRTM measurement. This indication is discarded and no other action performed if the TPM does not contain an H-CRTM Event Sequence context.

NOTE 1 An H-CRTM Event Sequence context is created by _TPM_Hash_Start().

If the H-CRTM Event Sequence occurs after TPM2_Startup(), the TPM will set all of the PCR designated in the platform-specific specifications as resettable by this event to the value indicated in the platform specific specification and increment *restartCount*. The TPM will then Extend the Event Sequence digest/digests into the designated D-RTM PCR (PCR[17]).

$$\text{PCR}[17][hashAlg] \coloneqq \mathbf{H}_{hashAlg}(initial\_value \,||\, \mathbf{H}_{hashAlg}(hash\_data)) \tag{7}$$

where

$hashAlg$	hash algorithm associated with a bank of PCR
$initial\_value$	initialization value specified in the platform-specific specification (should be 0…0)
$hash\_data$	all the octets of data received in _TPM_Hash_Data indications

A _TPM_Hash_End indication that occurs after TPM2_Startup() will increment *pcrUpdateCounter* unless a platform-specific specification excludes modifications of PCR[DRTM] from causing an increment.

A platform-specific specification may allow an H-CRTM Event Sequence before TPM2_Startup(). If so, _TPM_Hash_End will complete the digest, initialize PCR[0] with a digest-size value of 4, and then extend the H-CRTM Event Sequence data into PCR[0].

$$\text{PCR}[0][hashAlg] \coloneqq \mathbf{H}_{hashAlg}(0...04 \,||\, \mathbf{H}_{hashAlg}(hash\_data)) \tag{8}$$

NOTE 2 The entire sequence of _TPM_Hash_Start, _TPM_Hash_Data, and _TPM_Hash_End are required to complete before TPM2_Startup() or the sequence will have no effect on the TPM.

NOTE 3 PCR[0] does not need to be updated according to (8) until the end of TPM2_Startup().

22.11.2 Detailed Actions

```
#include "Tpm.h"
```

This function is called to process a _TPM_Hash_End() indication.

```
LIB_EXPORT void
_TPM_Hash_End(
    void
    )
{
    UINT32          i;
    TPM2B_DIGEST    digest;
    HASH_OBJECT    *hashObject;
    TPMI_DH_PCR     pcrHandle;

    // If the DRTM handle is not being used, then either _TPM_Hash_Start has not
    // been called, _TPM_Hash_End was previously called, or some other command
    // was executed and the sequence was aborted.
    if(g_DRTMHandle == TPM_RH_UNASSIGNED)
        return;

    // Get DRTM sequence object
    hashObject = (HASH_OBJECT *)HandleToObject(g_DRTMHandle);

    // Is this _TPM_Hash_End after Startup or before
    if(TPMIsStarted())
    {
        // After

        // Reset the DRTM PCR
        PCRResetDynamics();

        // Extend the DRTM_PCR.
        pcrHandle = PCR_FIRST + DRTM_PCR;

        // DRTM sequence increments restartCount
        gr.restartCount++;
    }
    else
    {
        pcrHandle = PCR_FIRST + HCRTM_PCR;
        g_DrtmPreStartup = TRUE;
    }

    // Complete hash and extend PCR, or if this is an HCRTM, complete
    // the hash, reset the H-CRTM register (PCR[0]) to 0...04, and then
    // extend the H-CRTM data
    for(i = 0; i < HASH_COUNT; i++)
    {
        TPMI_ALG_HASH       hash = CryptHashGetAlgByIndex(i);
        // make sure that the PCR is implemented for this algorithm
        if(PcrIsAllocated(pcrHandle,
                          hashObject->state.hashState[i].hashAlg))
        {
            // Complete hash
            digest.t.size = CryptHashGetDigestSize(hash);
            CryptHashEnd2B(&hashObject->state.hashState[i], &digest.b);

            PcrDrtm(pcrHandle, hash, &digest);
        }
    }

    // Flush sequence object.
```

```
    FlushObject(g_DRTMHandle);

    g_DRTMHandle = TPM_RH_UNASSIGNED;

    return;
}
```

23 Enhanced Authorization (EA) Commands

23.1 Introduction

The commands in this clause 1 are used for policy evaluation. When successful, each command will update the *policySession→policyDigest* in a policy session context in order to establish that the authorizations required to use an object have been provided. Many of the commands will also modify other parts of a policy context so that the caller may constrain the scope of the authorization that is provided.

NOTE 1 Many of the terms used in this clause are described in detail in TPM 2.0 Part 1 and are not redefined in this clause.

The *policySession* parameter of the command is the handle of the policy session context to be modified by the command.

If the *policySession* parameter indicates a trial policy session, then the *policySession→policyDigest* will be updated and the indicated validations are not performed. However, any authorizations required to perform the policy command will be checked and dictionary attack logic invoked as necessary.

NOTE 2 If software is used to create policies, no authorization values are used. For example, TPM_PolicySecret requires an authorization in a trial policy session, but not in a policy calculation outside the TPM.

NOTE 3 A policy session is set to a trial policy by TPM2_StartAuthSession(*sessionType* = TPM_SE_TRIAL).

NOTE 4 Unless there is an unmarshaling error in the parameters of the command, these commands will return TPM_RC_SUCCESS when *policySession* references a trial session.

NOTE 5 Policy context other than the *policySession→policyDigest* may be updated for a trial policy but it is not required.

23.2 Signed Authorization Actions

23.2.1 Introduction

The TPM2_PolicySigned, TPM_PolicySecret, and TPM2_PolicyTicket commands use many of the same functions. This clause consolidates those functions to simplify the document and to ensure uniformity of the operations.

23.2.2 Policy Parameter Checks

These parameter checks will be performed when indicated in the description of each of the commands:

a) *nonceTPM* – If this parameter is not the Empty Buffer, and it does not match *policySession→nonceTPM*, then the TPM shall return TPM_RC_VALUE.

b) *expiration* – If this parameter is not zero, then:

   1) if *nonceTPM* is not an Empty Buffer, then the absolute value of *expiration* is converted to milliseconds and added to *policySession→startTime* to create the *timeout* value and proceed to c).

   2) If *nonceTPM* is an Empty Buffer, then the absolute value of *expiration* is converted to milliseconds and used as the *timeout* value and proceed to c).

   However, *timeout* can only be changed to a smaller value.

c) *timeout* – If *timeout* is less than the current value of *Time*, or the current *timeEpoch* is not the same as *policySession→timeEpoch, the* TPM shall return TPM_RC_EXPIRED

d) *cpHashA* – If this parameter is not an Empty Buffer

   NOTE 2 *cpHashA* is the hash of the command to be executed using this policy session in the authorization. The algorithm used to compute this hash is required to be the algorithm of the policy session.

   1) the TPM shall return TPM_RC_CPHASH if *policySession→cpHash* is set and the contents of *policySession→cpHash* are not the same as *cpHashA*; or

      NOTE 3 cpHash is the expected cpHash value held in the policy session context.

   2) the TPM shall return TPM_RC_SIZE if *cpHashA* is not the same size as *policySession→policyDigest*.

      NOTE 4 policySession→policyDigest is the size of the digest produced by the hash algorithm used to compute policyDigest.

23.2.3 Policy Digest Update Function (PolicyUpdate())

This is the update process for *policySession→policyDigest* used by TPM2_PolicySigned(), TPM2_PolicySecret(), TPM2_PolicyTicket(), and TPM2_PolicyAuthorize(). The function prototype for the update function is:

$$\mathbf{PolicyUpdate}(commandCode, arg2, arg3) \tag{9}$$

where

$arg2$	a TPM2B_NAME
$arg3$	a TPM2B

These parameters are used to update *policySession→policyDigest* by

$$policyDigest_{new} \coloneqq \mathbf{H}_{policyAlg}(policyDigest_{old} \mid\mid commandCode \mid\mid arg2.name) \tag{10}$$

followed by

$$policyDigest_{new+1} \coloneqq \mathbf{H}_{\mathrm{policyAlg}}(policyDigest_{new} \mid\mid arg3.buffer) \tag{11}$$

where

$\mathbf{H}_{policyAlg}()$	the hash algorithm chosen when the policy session was started

NOTE 1 If *arg3* is a TPM2B_NAME, then *arg3.buffer* will actually be an *arg3.name*.

NOTE 2 The *arg2.size* and *arg3.size* fields are not included in the hashes.

NOTE 3 **PolicyUpdate**() uses two hash operations because *arg2* and *arg3* are variable-sized and the concatenation of *arg2* and *arg3* in a single hash could produce the same digest even though *arg2* and *arg3* are different. For example, arg2 = 1 2 3 and arg3 = 4 5 6 would produce the same digest as arg2 = 1 2 and arg3 = 3 4 5 6. Processing of the arguments separately in different Extend operation ensures that the digest produced by **PolicyUpdate**() will be different if *arg2* and *arg3* are different.

23.2.4 Policy Context Updates

When a policy command modifies some part of the policy session context other than the *policySession→policyDigest*, the following rules apply.

- ***cpHash*** – this parameter may only be changed if it contains its initialization value (an Empty Buffer). If *cpHash* is not the Empty Buffer when a policy command attempts to update it, the TPM will return an error (TPM_RC_CPHASH) if the current and update values are not the same.
- ***timeOut*** – this parameter may only be changed to a smaller value. If a command attempts to update this value with a larger value (longer into the future), the TPM will discard the update value. This is not an error condition.
- ***commandCode*** – once set by a policy command, this value may not be changed except by TPM2_PolicyRestart(). If a policy command tries to change this to a different value, an error is returned (TPM_RC_POLICY_CC).
- ***pcrUpdateCounter*** – this parameter is updated by TPM2_PolicyPCR(). This value may only be set once during a policy. Each time TPM2_PolicyPCR() executes, it checks to see if *policySession→pcrUpdateCounter* has its default state, indicating that this is the first TPM2_PolicyPCR(). If it has its default value, then *policySession→pcrUpdateCounter* is set to the current value of *pcrUpdateCounter*. If *policySession→pcrUpdateCounter* does not have its default value and its value is not the same as *pcrUpdateCounter*, the TPM shall return TPM_RC_PCR_CHANGED.

  NOTE 1 If this parameter and *pcrUpdateCounter* are not the same, it indicates that PCR have changed since checked by the previous TPM2_PolicyPCR(). Since they have changed, the previous PCR validation is no longer valid.

- ***commandLocality*** – this parameter is the logical AND of all enabled localities. All localities are enabled for a policy when the policy session is created. TPM2_PolicyLocalities() selectively disables localities. Once use of a policy for a locality has been disabled, it cannot be enabled except by TPM2_PolicyRestart().
- ***isPPRequired*** – once SET, this parameter may only be CLEARed by TPM2_PolicyRestart().
- ***isAuthValueNeeded*** – once SET, this parameter may only be CLEARed by TPM2_PolicyPassword() or TPM2_PolicyRestart().
- ***isPasswordNeeded*** – once SET, this parameter may only be CLEARed by TPM2_PolicyAuthValue() or TPM2_PolicyRestart(),

NOTE 2 Both TPM2_PolicyAuthValue() and TPM2_PolicyPassword() change *policySession→policyDigest* in the same way. The different commands simply indicate to the TPM the format used for the *authValue* (HMAC or clear text). Both commands could be in the same policy. The final instance of these commands determines the format.

23.2.5 Policy Ticket Creation

For TPM2_PolicySigned() or TPM2_PolicySecret(), if the caller specified a negative value for *expiration*, then the TPM will return a ticket that includes a value indicating when the authorization expires. Otherwise, the TPM will return a NULL Ticket.

NOTE 1 If the *authHandle* in TPM2_PolicySecret() references a PIN Pass Index, then the command may succeed but a NULL Ticket will be returned.

The required computation for the digest in the authorization ticket is:

$$\mathbf{HMAC}_{contextAlg}(proof, (\text{TPM\_ST\_AUTH\_xxx} \mathbin{||} cpHash \mathbin{||} policyRef \mathbin{||} authName \mathbin{||} timeout \mathbin{||} [timeEpoch] \mathbin{||} [resetCount])) \tag{12}$$

where

$\mathbf{HMAC}_{contextAlg}()$	an HMAC using the context integrity hash
$proof$	a TPM secret value associated with the hierarchy of the object associated with *authName*
TPM_ST_AUTH_xxx	either TPM_ST_AUTH_SIGNED or TPM_ST_AUTH_SECRET; used to ensure that the ticket is properly used
$cpHash$	optional hash of the authorized command
$policyRef$	optional reference to a policy value
$authName$	Name of the object that signed the authorization
$timeout$	implementation-specific value indicating when the authorization expires
$timeEpoch$	implementation-specific representation of the $timeEpoch$ at the time the ticket was created

NOTE 2 Not included if *timeout* is zero.

$resetCount$	implementation-specific representation of the TPM's *totalResetCount*

NOTE 3 Not included it *timeout* is zero or if *nonceTPM* was include in the authorization.

23.3 TPM2_PolicySigned

23.3.1 General Description

This command includes a signed authorization in a policy. The command ties the policy to a signing key by including the Name of the signing key in the *policyDigest*

If *policySession* is a trial session, the TPM will not check the signature and will update *policySession→policyDigest* as described in 23.2.3 as if a properly signed authorization was received, but no ticket will be produced.

If *policySession* is not a trial session, the TPM will validate *auth* and only perform the update if it is a valid signature over the fields of the command.

The authorizing entity will sign a digest of the authorization qualifiers: *nonceTPM*, *expiration*, *cpHashA*, and *policyRef*. The digest is computed as:

$$aHash \coloneqq \mathbf{H}_{authAlg}(nonceTPM \mid\mid expiration \mid\mid cpHashA \mid\mid policyRef) \tag{13}$$

where

$\mathbf{H}_{authAlg}()$ the hash associated with the auth parameter of this command

NOTE 1 Each signature and key combination indicates the scheme and each scheme has an associated hash.

$nonceTPM$ the nonceTPM parameter from the TPM2_StartAuthSession() response. If the authorization is not limited to this session, the size of this value is zero.

$expiration$ time limit on authorization set by authorizing object. This 32-bit value is set to zero if the expiration time is not being set.

$cpHashA$ digest of the command parameters for the command being approved using the hash algorithm of the policy session. Set to an Empty Digest if the authorization is not limited to a specific command.

NOTE 3 This is not the *cpHash* of this TPM2_PolicySigned() command.

$policyRef$ an opaque value determined by the authorizing entity. Set to the Empty Buffer if no value is present.

NOTE 4 The *nonceTPM*, *cpHashA*, and *policyRef* qualifiers used to compute *aHash* use the TPM2B buffer but do not prepend the size.

EXAMPLE The computation for an *aHash* if there are no restrictions is:

$$aHash \coloneqq \mathbf{H}_{authAlg}(\text{00 00 00 00}_{16})$$

which is the hash of an expiration time of zero.

The *aHash* is signed by the key associated with a key whose handle is *authObject*. The signature and signing parameters are combined to create the *auth* parameter.

The TPM will perform the parameter checks listed in 23.2.2

If the parameter checks succeed, the TPM will construct a test digest (*tHash*) over the provided parameters using the same formulation as shown in equation (13) above.

If *tHash* does not match the digest of the signed *aHash,* then the authorization fails and the TPM shall return TPM_RC_POLICY_FAIL and make no change to *policySession→policyDigest*.

When all validations have succeeded, *policySession→policyDigest* is updated by **PolicyUpdate**() (see 23.2.3).

$$\mathbf{PolicyUpdate}(\text{TPM\_CC\_PolicySigned}, authObject{\rightarrow}Name, policyRef) \tag{14}$$

authObject→Name is a TPM2B_NAME. *policySession* is updated as described in 23.2.4. The TPM will optionally produce a ticket as described in 23.2.5.

Authorization to use *authObject* is not required.

23.3.2 Command and Response

Table 124 — TPM2_PolicySigned Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit, encrypt, or decrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PolicySigned
TPMI_DH_OBJECT	authObject	handle for a key that will validate the signature Auth Index: None
TPMI_SH_POLICY	policySession	handle for the policy session being extended Auth Index: None
TPM2B_NONCE	nonceTPM	the policy nonce for the session This can be the Empty Buffer.
TPM2B_DIGEST	cpHashA	digest of the command parameters to which this authorization is limited This is not the *cpHash* for this command but the *cpHash* for the command to which this policy session will be applied. If it is not limited, the parameter will be the Empty Buffer.
TPM2B_NONCE	policyRef	a reference to a policy relating to the authorization – may be the Empty Buffer Size is limited to be no larger than the nonce size supported on the TPM.
INT32	expiration	time when authorization will expire, measured in seconds from the time that *nonceTPM* was generated If *expiration* is non-negative, a NULL Ticket is returned. See 23.2.5.
TPMT_SIGNATURE	auth	signed authorization (not optional)

Table 125 — TPM2_PolicySigned Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_TIMEOUT	timeout	implementation-specific time value, used to indicate to the TPM when the ticket expires NOTE If *policyTicket* is a NULL Ticket, then this shall be the Empty Buffer.
TPMT_TK_AUTH	policyTicket	produced if the command succeeds and *expiration* in the command was non-zero; this ticket will use the TPMT_ST_AUTH_SIGNED structure tag. See 23.2.5

23.3.3 Detailed Actions

```
#include "Tpm.h"
#include "Policy_spt_fp.h"
#include "PolicySigned_fp.h"
#if CC_PolicySigned  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_CPHASH	*cpHash* was previously set to a different value
TPM_RC_EXPIRED	*expiration* indicates a time in the past or *expiration* is non-zero but no *nonceTPM* is present
TPM_RC_NONCE	*nonceTPM* is not the nonce associated with the *policySession*
TPM_RC_SCHEME	the signing scheme of *auth* is not supported by the TPM
TPM_RC_SIGNATURE	the signature is not genuine
TPM_RC_SIZE	input *cpHash* has wrong size

```
TPM_RC
TPM2_PolicySigned(
    PolicySigned_In     *in,            // IN: input parameter list
    PolicySigned_Out    *out            // OUT: output parameter list
    )
{
    TPM_RC                   result = TPM_RC_SUCCESS;
    SESSION                 *session;
    TPM2B_NAME               entityName;
    TPM2B_DIGEST             authHash;
    HASH_STATE               hashState;
    UINT64                   authTimeout = 0;
// Input Validation
    // Set up local pointers
    session = SessionGet(in->policySession);     // the session structure

    // Only do input validation if this is not a trial policy session
    if(session->attributes.isTrialPolicy == CLEAR)
    {
        authTimeout = ComputeAuthTimeout(session, in->expiration, &in->nonceTPM);

        result = PolicyParameterChecks(session, authTimeout,
                                       &in->cpHashA, &in->nonceTPM,
                                       RC_PolicySigned_nonceTPM,
                                       RC_PolicySigned_cpHashA,
                                       RC_PolicySigned_expiration);
        if(result != TPM_RC_SUCCESS)
            return result;
        // Re-compute the digest being signed

        // Start hash
        authHash.t.size = CryptHashStart(&hashState,
                                         CryptGetSignHashAlg(&in->auth));
        // If there is no digest size, then we don't have a verification function
        // for this algorithm (e.g. TPM_ALG_ECDAA) so indicate that it is a
        // bad scheme.
        if(authHash.t.size == 0)
            return TPM_RCS_SCHEME + RC_PolicySigned_auth;

        //  nonceTPM
        CryptDigestUpdate2B(&hashState, &in->nonceTPM.b);

```



```
        //  expiration
        CryptDigestUpdateInt(&hashState, sizeof(UINT32), in->expiration);

        //  cpHashA
        CryptDigestUpdate2B(&hashState, &in->cpHashA.b);

        //  policyRef
        CryptDigestUpdate2B(&hashState, &in->policyRef.b);

        //  Complete digest
        CryptHashEnd2B(&hashState, &authHash.b);

        // Validate Signature.  A TPM_RC_SCHEME, TPM_RC_HANDLE or TPM_RC_SIGNATURE
        // error may be returned at this point
        result = CryptValidateSignature(in->authObject, &authHash, &in->auth);
        if(result != TPM_RC_SUCCESS)
            return RcSafeAddToResult(result, RC_PolicySigned_auth);
    }
// Internal Data Update
    // Update policy with input policyRef and name of authorization key
    // These values are updated even if the session is a trial session
    PolicyContextUpdate(TPM_CC_PolicySigned,
                        EntityGetName(in->authObject, &entityName),
                        &in->policyRef,
                        &in->cpHashA, authTimeout, session);
// Command Output
    // Create ticket and timeout buffer if in->expiration < 0 and this is not
    // a trial session.
    // NOTE: PolicyParameterChecks() makes sure that nonceTPM is present
    // when expiration is non-zero.
    if(in->expiration < 0
       && session->attributes.isTrialPolicy == CLEAR)
    {
        BOOL        expiresOnReset = (in->nonceTPM.t.size == 0);
        // Compute policy ticket
        authTimeout &= ~EXPIRATION_BIT;

        TicketComputeAuth(TPM_ST_AUTH_SIGNED, EntityGetHierarchy(in->authObject),
                          authTimeout, expiresOnReset, &in->cpHashA, &in->policyRef,
                          &entityName, &out->policyTicket);
        // Generate timeout buffer.  The format of output timeout buffer is
        // TPM-specific.
        // Note: In this implementation, the timeout buffer value is computed after
        // the ticket is produced so, when the ticket is checked, the expiration
        // flag needs to be extracted before the ticket is checked.
        // In the Windows compatible version, the least-significant bit of the
        // timeout value is used as a flag to indicate if the authorization expires
        // on reset. The flag is the MSb.
        out->timeout.t.size = sizeof(authTimeout);
        if(expiresOnReset)
            authTimeout |= EXPIRATION_BIT;
        UINT64_TO_BYTE_ARRAY(authTimeout, out->timeout.t.buffer);
    }
    else
    {
        // Generate a null ticket.
        // timeout buffer is null
        out->timeout.t.size = 0;

        // authorization ticket is null
        out->policyTicket.tag = TPM_ST_AUTH_SIGNED;
        out->policyTicket.hierarchy = TPM_RH_NULL;
        out->policyTicket.digest.t.size = 0;
    }
    return TPM_RC_SUCCESS;
}
```



```
#endif // CC_PolicySigned
```

23.4 TPM2_PolicySecret

23.4.1 General Description

This command includes a secret-based authorization to a policy. The caller proves knowledge of the secret value using an authorization session using the *authValue* associated with *authHandle*. A password session, an HMAC session, or a policy session containing TPM2_PolicyAuthValue() or TPM2_PolicyPassword() will satisfy this requirement.

If a policy session is used and use of the *authValue* of *authHandle* is not required, the TPM will return TPM_RC_MODE. That is, the session for *authHandle* must have either *isAuthValueNeeded* or *isPasswordNeeded* SET.

The secret is the *authValue* of the entity whose handle is *authHandle,* which may be any TPM entity with a handle and an associated *authValue*. This includes the reserved handles (for example, Platform, Storage, and Endorsement), NV Indexes, and loaded objects. *authEntity* is the entity referenced by *authHandle*. If *authEntity* references an Ordinary object, it must have *userWithAuth* SET.

NOTE 1 The *userWithAuth* requirement permits the implementation to use common authorization code.

If *authEntity* references a non-PIN Index. TPMA_NV_AUTHREAD is required to be SET in the Index. If *authEntity* references an NV PIN index, TPMA_NV_WRITTEN is required to be SET and *pinCount* must be less than *pinLimit*.

NOTE 2 The authorization value for a hierarchy cannot be used in this command if the hierarchy is disabled.

If the authorization check fails, then the normal dictionary attack logic is invoked.

If the authorization provided by the authorization session is valid, the command parameters are checked as described in 23.2.2.

When all validations have succeeded, *policySession→policyDigest* is updated by **PolicyUpdate**() (see 23.2.3).

$$\mathbf{PolicyUpdate}(\text{TPM\_CC\_PolicySecret}, authEntity{\rightarrow}Name, policyRef) \qquad (15)$$

authEntity→Name is a TPM2B_NAME*. policySession* is updated as described in 23.2.4. The TPM will optionally produce a ticket as described in 23.2.5.

If the session is a trial session, *policySession→policyDigest* is updated if the authorization is valid.

NOTE 2 If an HMAC is used to convey the authorization, a separate session is needed for the authorization. Because the HMAC in that authorization will include a nonce that prevents replay of the authorization, the value of the *nonceTPM* parameter in this command is limited. It is retained mostly to provide processing consistency with TPM2_PolicySigned().

23.4.2 Command and Response

Table 126 — TPM2_PolicySecret Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PolicySecret
TPMI_DH_ENTITY	@authHandle	handle for an entity providing the authorization Auth Index: 1 Auth Role: USER
TPMI_SH_POLICY	policySession	handle for the policy session being extended Auth Index: None
TPM2B_NONCE	nonceTPM	the policy nonce for the session This can be the Empty Buffer.
TPM2B_DIGEST	cpHashA	digest of the command parameters to which this authorization is limited This not the *cpHash* for this command but the *cpHash* for the command to which this policy session will be applied. If it is not limited, the parameter will be the Empty Buffer.
TPM2B_NONCE	policyRef	a reference to a policy relating to the authorization – may be the Empty Buffer Size is limited to be no larger than the nonce size supported on the TPM.
INT32	expiration	time when authorization will expire, measured in seconds from the time that *nonceTPM* was generated If *expiration* is non-negative, a NULL Ticket is returned. See 23.2.5.

Table 127 — TPM2_PolicySecret Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_TIMEOUT	timeout	implementation-specific time value used to indicate to the TPM when the ticket expires
TPMT_TK_AUTH	policyTicket	produced if the command succeeds and *expiration* in the command was non-zero (See 23.2.5). This ticket will use the TPMT_ST_AUTH_SECRET structure tag

23.4.3 Detailed Actions

```
#include "Tpm.h"
#include "PolicySecret_fp.h"
#if CC_PolicySecret  // Conditional expansion of this file
#include "Policy_spt_fp.h"
#include "NV_spt_fp.h"
```

Error Returns	Meaning
TPM_RC_CPHASH	*cpHash* for policy was previously set to a value that is not the same as *cpHashA*
TPM_RC_EXPIRED	*expiration* indicates a time in the past
TPM_RC_NONCE	*nonceTPM* does not match the nonce associated with *policySession*
TPM_RC_SIZE	*cpHashA* is not the size of a digest for the hash associated with *policySession*

```
TPM_RC
TPM2_PolicySecret(
    PolicySecret_In     *in,            // IN: input parameter list
    PolicySecret_Out    *out            // OUT: output parameter list
    )
{
    TPM_RC                   result;
    SESSION                 *session;
    TPM2B_NAME               entityName;
    UINT64                   authTimeout = 0;
// Input Validation
    // Get pointer to the session structure
    session = SessionGet(in->policySession);

    //Only do input validation if this is not a trial policy session
    if(session->attributes.isTrialPolicy == CLEAR)
    {
        authTimeout = ComputeAuthTimeout(session, in->expiration, &in->nonceTPM);

        result = PolicyParameterChecks(session, authTimeout,
                                       &in->cpHashA, &in->nonceTPM,
                                       RC_PolicySecret_nonceTPM,
                                       RC_PolicySecret_cpHashA,
                                       RC_PolicySecret_expiration);
        if(result != TPM_RC_SUCCESS)
            return result;
    }
// Internal Data Update
    // Update policy context with input policyRef and name of authorizing key
    // This value is computed even for trial sessions. Possibly update the cpHash
    PolicyContextUpdate(TPM_CC_PolicySecret,
                        EntityGetName(in->authHandle, &entityName), &in->policyRef,
                        &in->cpHashA, authTimeout, session);
// Command Output
    // Create ticket and timeout buffer if in->expiration < 0 and this is not
    // a trial session.
    // NOTE: PolicyParameterChecks() makes sure that nonceTPM is present
    // when expiration is non-zero.
    if(in->expiration < 0
       && session->attributes.isTrialPolicy == CLEAR
       && !NvIsPinPassIndex(in->authHandle))
    {
        BOOL        expiresOnReset = (in->nonceTPM.t.size == 0);
        // Compute policy ticket
```

```
        authTimeout &= ~EXPIRATION_BIT;
        TicketComputeAuth(TPM_ST_AUTH_SECRET, EntityGetHierarchy(in->authHandle),
                          authTimeout, expiresOnReset, &in->cpHashA, &in->policyRef,
                          &entityName, &out->policyTicket);
        // Generate timeout buffer.  The format of output timeout buffer is
        // TPM-specific.
        // Note: In this implementation, the timeout buffer value is computed after
        // the ticket is produced so, when the ticket is checked, the expiration
        // flag needs to be extracted before the ticket is checked.
        out->timeout.t.size = sizeof(authTimeout);
        // In the Windows compatible version, the least-significant bit of the
        // timeout value is used as a flag to indicate if the authorization expires
        // on reset. The flag is the MSb.
        if(expiresOnReset)
            authTimeout |= EXPIRATION_BIT;
        UINT64_TO_BYTE_ARRAY(authTimeout, out->timeout.t.buffer);
    }
    else
    {
        // timeout buffer is null
        out->timeout.t.size = 0;

        // authorization ticket is null
        out->policyTicket.tag = TPM_ST_AUTH_SECRET;
        out->policyTicket.hierarchy = TPM_RH_NULL;
        out->policyTicket.digest.t.size = 0;
    }
    return TPM_RC_SUCCESS;
}
#endif // CC_PolicySecret
```

23.5 TPM2_PolicyTicket

23.5.1 General Description

This command is similar to TPM2_PolicySigned() except that it takes a ticket instead of a signed authorization. The ticket represents a validated authorization that had an expiration time associated with it.

The parameters of this command are checked as described in 23.2.2.

If the checks succeed, the TPM uses the *timeout*, *cpHashA*, *policyRef*, and *authName* to construct a ticket to compare with the value in *ticket*. If these tickets match, then the TPM will create a TPM2B_NAME (*objectName*) using *authName* and update the context of *policySession* by **PolicyUpdate**() (see 23.2.3).

$$\mathbf{PolicyUpdate}(commandCode, authName, policyRef) \tag{16}$$

If the structure tag of ticket is TPM_ST_AUTH_SECRET, then *commandCode* will be TPM_CC_PolicySecret. If the structure tag of ticket is TPM_ST_AUTH_SIGNED, then *commandCode* will be TPM_CC_PolicySIgned.

policySession is updated as described in 23.2.4.

23.5.2 Command and Response

Table 128 — TPM2_PolicyTicket Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit or decrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PolicyTicket
TPMI_SH_POLICY	policySession	handle for the policy session being extended Auth Index: None
TPM2B_TIMEOUT	timeout	time when authorization will expire The contents are TPM specific. This shall be the value returned when ticket was produced.
TPM2B_DIGEST	cpHashA	digest of the command parameters to which this authorization is limited If it is not limited, the parameter will be the Empty Buffer.
TPM2B_NONCE	policyRef	reference to a qualifier for the policy – may be the Empty Buffer
TPM2B_NAME	authName	name of the object that provided the authorization
TPMT_TK_AUTH	ticket	an authorization ticket returned by the TPM in response to a TPM2_PolicySigned() or TPM2_PolicySecret()

Table 129 — TPM2_PolicyTicket Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

23.5.3 Detailed Actions

```
#include "Tpm.h"
#include "PolicyTicket_fp.h"
#if CC_PolicyTicket  // Conditional expansion of this file
#include "Policy_spt_fp.h"
```

Error Returns	Meaning
TPM_RC_CPHASH	policy's *cpHash* was previously set to a different value
TPM_RC_EXPIRED	*timeout* value in the ticket is in the past and the ticket has expired
TPM_RC_SIZE	*timeout* or *cpHash* has invalid size for the
TPM_RC_TICKET	*ticket* is not valid

```
TPM_RC
TPM2_PolicyTicket(
    PolicyTicket_In     *in             // IN: input parameter list
    )
{
    TPM_RC                  result;
    SESSION                *session;
    UINT64                  authTimeout;
    TPMT_TK_AUTH            ticketToCompare;
    TPM_CC                  commandCode = TPM_CC_PolicySecret;
    BOOL                    expiresOnReset;

// Input Validation

    // Get pointer to the session structure
    session = SessionGet(in->policySession);

    // NOTE: A trial policy session is not allowed to use this command.
    // A ticket is used in place of a previously given authorization. Since
    // a trial policy doesn't actually authenticate, the validated
    // ticket is not necessary and, in place of using a ticket, one
    // should use the intended authorization for which the ticket
    // would be a substitute.
    if(session->attributes.isTrialPolicy)
        return TPM_RCS_ATTRIBUTES + RC_PolicyTicket_policySession;
    // Restore timeout data.  The format of timeout buffer is TPM-specific.
    // In this implementation, the most significant bit of the timeout value is
    // used as the flag to indicate that the ticket expires on TPM Reset or
    // TPM Restart. The flag has to be removed before the parameters and ticket
    // are checked.
    if(in->timeout.t.size != sizeof(UINT64))
        return TPM_RCS_SIZE + RC_PolicyTicket_timeout;
    authTimeout = BYTE_ARRAY_TO_UINT64(in->timeout.t.buffer);

    // extract the flag
    expiresOnReset = (authTimeout & EXPIRATION_BIT) != 0;
    authTimeout &= ~EXPIRATION_BIT;

    // Do the normal checks on the cpHashA and timeout values
    result = PolicyParameterChecks(session, authTimeout,
                                   &in->cpHashA,
                                   NULL,                     // no nonce
                                   0,                        // no bad nonce return
                                   RC_PolicyTicket_cpHashA,
                                   RC_PolicyTicket_timeout);
    if(result != TPM_RC_SUCCESS)
        return result;
```



```
    // Validate Ticket
    // Re-generate policy ticket by input parameters
    TicketComputeAuth(in->ticket.tag, in->ticket.hierarchy,
                      authTimeout, expiresOnReset, &in->cpHashA, &in->policyRef,
                      &in->authName, &ticketToCompare);
    // Compare generated digest with input ticket digest
    if(!MemoryEqual2B(&in->ticket.digest.b, &ticketToCompare.digest.b))
        return TPM_RCS_TICKET + RC_PolicyTicket_ticket;

// Internal Data Update

    // Is this ticket to take the place of a TPM2_PolicySigned() or
    // a TPM2_PolicySecret()?
    if(in->ticket.tag == TPM_ST_AUTH_SIGNED)
        commandCode = TPM_CC_PolicySigned;
    else if(in->ticket.tag == TPM_ST_AUTH_SECRET)
        commandCode = TPM_CC_PolicySecret;
    else
        // There could only be two possible tag values.  Any other value should
        // be caught by the ticket validation process.
        FAIL(FATAL_ERROR_INTERNAL);

    // Update policy context
    PolicyContextUpdate(commandCode, &in->authName, &in->policyRef,
                        &in->cpHashA, authTimeout, session);

    return TPM_RC_SUCCESS;
}
#endif // CC_PolicyTicket
```

23.6 TPM2_PolicyOR

23.6.1 General Description

This command allows options in authorizations without requiring that the TPM evaluate all of the options. If a policy may be satisfied by different sets of conditions, the TPM need only evaluate one set that satisfies the policy. This command will indicate that one of the required sets of conditions has been satisfied.

PolicySession→policyDigest is compared against the list of provided values. If the current *policySession→policyDigest* does not match any value in the list, the TPM shall return TPM_RC_VALUE. Otherwise, the TPM will reset *policySession→policyDigest* to a Zero Digest. Then *policySession→policyDigest* is extended by the concatenation of TPM_CC_PolicyOR and the concatenation of all of the digests.

If *policySession* is a trial session, the TPM will assume that *policySession→policyDigest* matches one of the list entries and compute the new value of *policyDigest*.

The algorithm for computing the new value for *policyDigest* of *policySession* is:

a) Concatenate all the digest values in *pHashList*:

$$digests \coloneqq pHashList.digests[1].buffer \parallel \ldots \parallel pHashList.digests[n].buffer \tag{17}$$

NOTE 1 The TPM will not return an error if the size of an entry is not the same as the size of the digest of the policy. However, that entry cannot match *policyDigest*.

b) Reset *policyDigest* to a Zero Digest.

c) Extend the command code and the hashes computed in step a) above:

$$policyDigest_{new} \coloneqq \mathbf{H}_{policyAlg}(policyDigest_{old} \parallel \text{TPM\_CC\_PolicyOR} \parallel digests) \tag{18}$$

NOTE 2 The computation in b) and c) above is equivalent to:

$$policyDigest_{new} \coloneqq \mathbf{H}_{policyAlg}(0\ldots0 \parallel \text{TPM\_CC\_PolicyOR} \parallel digests)$$

A TPM shall support a list with at least eight tagged digest values.

NOTE 3 If policies are to be portable between TPMs, then they should not use more than eight values.

23.6.2 Command and Response

Table 130 — TPM2_PolicyOR Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PolicyOR
TPMI_SH_POLICY	policySession	handle for the policy session being extended Auth Index: None
TPML_DIGEST	pHashList	the list of hashes to check for a match

Table 131 — TPM2_PolicyOR Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

23.6.3 Detailed Actions

```
#include "Tpm.h"
#include "PolicyOR_fp.h"
#if CC_PolicyOR  // Conditional expansion of this file
#include "Policy_spt_fp.h"
```

Error Returns	Meaning
TPM_RC_VALUE	no digest in *pHashList* matched the current value of *policyDigest* for *policySession*

```
TPM_RC
TPM2_PolicyOR(
    PolicyOR_In     *in              // IN: input parameter list
    )
{
    SESSION     *session;
    UINT32       i;

// Input Validation and Update

    // Get pointer to the session structure
    session = SessionGet(in->policySession);

    // Compare and Update Internal Session policy if match
    for(i = 0; i < in->pHashList.count; i++)
    {
        if(session->attributes.isTrialPolicy == SET
           || (MemoryEqual2B(&session->u2.policyDigest.b,
                             &in->pHashList.digests[i].b)))
        {
            // Found a match
            HASH_STATE      hashState;
            TPM_CC          commandCode = TPM_CC_PolicyOR;

            // Start hash
            session->u2.policyDigest.t.size
                = CryptHashStart(&hashState, session->authHashAlg);
            // Set policyDigest to 0 string and add it to hash
            MemorySet(session->u2.policyDigest.t.buffer, 0,
                      session->u2.policyDigest.t.size);
            CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);

            // add command code
            CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), commandCode);

            // Add each of the hashes in the list
            for(i = 0; i < in->pHashList.count; i++)
            {
                // Extend policyDigest
                CryptDigestUpdate2B(&hashState, &in->pHashList.digests[i].b);
            }
            // Complete digest
            CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);

            return TPM_RC_SUCCESS;
        }
    }
    // None of the values in the list matched the current policyDigest
    return TPM_RCS_VALUE + RC_PolicyOR_pHashList;
}
#endif // CC_PolicyOR
```


23.7 TPM2_PolicyPCR

23.7.1 General Description

This command is used to cause conditional gating of a policy based on PCR. This command together with TPM2_PolicyOR() allows one group of authorizations to occur when PCR are in one state and a different set of authorizations when the PCR are in a different state.

The TPM will modify the *pcrs* parameter so that bits that correspond to unimplemented PCR are CLEAR. If *policySession* is not a trial policy session, the TPM will use the modified value of pcrs to select PCR values to hash according to TPM 2.0 Part 1, *Selecting Multiple PCR.* The hash algorithm of the policy session is used to compute a digest (*digestTPM*) of the selected PCR. If *pcrDigest* does not have a length of zero, then it is compared to *digestTPM*; and if the values do not match, the TPM shall return TPM_RC_VALUE and make no change to *policySession→policyDigest*. If the values match, or if the length of *pcrDigest* is zero, then *policySession→policyDigest* is extended by:

$$policyDigest_{new} \coloneqq \mathbf{H}_{policyAlg}(policyDigest_{old} \mathbin{||} \text{TPM\_CC\_PolicyPCR} \mathbin{||} pcrs \mathbin{||} digestTPM) \qquad (19)$$

where

$pcrs$	the pcrs parameter with bits corresponding to unimplemented PCR set to 0
$digestTPM$	the digest of the selected PCR using the hash algorithm of the policy session

NOTE 1 If the caller provides the expected PCR value, the intention is that the policy evaluation stop at that point if the PCR do not match. If the caller does not provide the expected PCR value, then the validity of the settings will not be determined until an attempt is made to use the policy for authorization. If the policy is constructed such that the PCR check comes before user authorization checks, this early termination would allow software to avoid unnecessary prompts for user input to satisfy a policy that would fail later due to incorrect PCR values.

After this command completes successfully, the TPM shall return TPM_RC_PCR_CHANGED if the policy session is used for authorization and the PCR are not known to be correct.

The TPM uses a "generation" number (*pcrUpdateCounter*) that is incremented each time PCR are updated (unless the PCR being changed is specified not to cause a change to this counter). The value of this counter is stored in the policy session context (*policySession→pcrUpdateCounter*) when this command is executed. When the policy is used for authorization, the current value of the counter is compared to the value in the policy session context and the authorization will fail if the values are not the same.

When this command is executed, *policySession→pcrUpdateCounter* is checked to see if it has been previously set (in the reference implementation, it has a value of zero if not previously set). If it has been set, it will be compared with the current value of *pcrUpdateCounter* to determine if any PCR changes have occurred. If the values are different, the TPM shall return TPM_RC_PCR_CHANGED.

NOTE 2 Since the pcrUpdateCounter is updated if any PCR is extended (except those specified not to do so), this means that the command will fail even if a PCR not specified in the policy is updated. This is an optimization for the purposes of conserving internal TPM memory. This would be a rare occurrence, and, if this should occur, the policy could be reset using the TPM2_PolicyRestart command and rerun.

If *policySession→pcrUpdateCounter* has not been set, then it is set to the current value of *pcrUpdateCounter*.

If this command is used for a trial *policySession*, *policySession→policyDigest* will be updated using the values from the command rather than the values from a digest of the TPM PCR. If the caller does not provide PCR settings (*pcrDigest* has a length of zero), the TPM may (and it is preferred to) use the current TPM PCR settings (*digestTPM)* in the calculation for the new *policyDigest*. The TPM may return an error if the caller does not provide a PCR digest for a trial policy session but this is not the preferred behavior.

The TPM will not check any PCR and will compute:

$$policyDigest_{new} \coloneqq \mathbf{H}_{policyAlg}(policyDigest_{old} \mathbin{||} \text{TPM\_CC\_PolicyPCR} \mathbin{||} pcrs \mathbin{||} pcrDigest) \tag{20}$$

In this computation, pcrs is the input parameter without modification.

NOTE 3 The pcrs parameter is expected to match the configuration of the TPM for which the policy is being computed which may not be the same as the TPM on which the trial policy is being computed.

NOTE 4 Although no PCR are checked in a trial policy session, *pcrDigest* is expected to correspond to some useful PCR values. It is legal, but pointless, to have the TPM aid in calculating a *policyDigest* corresponding to PCR values that are not useful in practice.

23.7.2 Command and Response

Table 132 — TPM2_PolicyPCR Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit or decrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PolicyPCR
TPMI_SH_POLICY	policySession	handle for the policy session being extended Auth Index: None
TPM2B_DIGEST	pcrDigest	expected digest value of the selected PCR using the hash algorithm of the session; may be zero length
TPML_PCR_SELECTION	pcrs	the PCR to include in the check digest

Table 133 — TPM2_PolicyPCR Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

23.7.3 Detailed Actions

```
#include "Tpm.h"
#include "PolicyPCR_fp.h"
#if CC_PolicyPCR  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_VALUE	if provided, *pcrDigest* does not match the current PCR settings
TPM_RC_PCR_CHANGED	a previous TPM2_PolicyPCR() set *pcrCounter* and it has changed

```
TPM_RC
TPM2_PolicyPCR(
    PolicyPCR_In    *in             // IN: input parameter list
    )
{
    SESSION         *session;
    TPM2B_DIGEST     pcrDigest;
    BYTE             pcrs[sizeof(TPML_PCR_SELECTION)];
    UINT32           pcrSize;
    BYTE            *buffer;
    TPM_CC           commandCode = TPM_CC_PolicyPCR;
    HASH_STATE       hashState;

// Input Validation

    // Get pointer to the session structure
    session = SessionGet(in->policySession);

    // Compute current PCR digest
    PCRComputeCurrentDigest(session->authHashAlg, &in->pcrs, &pcrDigest);

    // Do validation for non trial session
    if(session->attributes.isTrialPolicy == CLEAR)
    {
        // Make sure that this is not going to invalidate a previous PCR check
        if(session->pcrCounter != 0 && session->pcrCounter != gr.pcrCounter)
            return TPM_RC_PCR_CHANGED;

        // If the caller specified the PCR digest and it does not
        // match the current PCR settings, return an error..
        if(in->pcrDigest.t.size != 0)
        {
            if(!MemoryEqual2B(&in->pcrDigest.b, &pcrDigest.b))
                return TPM_RCS_VALUE + RC_PolicyPCR_pcrDigest;
        }
    }
    else
    {
        // For trial session, just use the input PCR digest if one provided
        // Note: It can't be too big because it is a TPM2B_DIGEST and the size
        // would have been checked during unmarshaling
        if(in->pcrDigest.t.size != 0)
            pcrDigest = in->pcrDigest;
    }
// Internal Data Update
    // Update policy hash
    // policyDigestnew = hash(   policyDigestold || TPM_CC_PolicyPCR
    //                      || PCRS || pcrDigest)
    //  Start hash
    CryptHashStart(&hashState, session->authHashAlg);

```



```
    //  add old digest
    CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);

    //  add commandCode
    CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), commandCode);

    //  add PCRS
    buffer = pcrs;
    pcrSize = TPML_PCR_SELECTION_Marshal(&in->pcrs, &buffer, NULL);
    CryptDigestUpdate(&hashState, pcrSize, pcrs);

    //  add PCR digest
    CryptDigestUpdate2B(&hashState, &pcrDigest.b);

    //  complete the hash and get the results
    CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);

    //  update pcrCounter in session context for non trial session
    if(session->attributes.isTrialPolicy == CLEAR)
    {
        session->pcrCounter = gr.pcrCounter;
    }

    return TPM_RC_SUCCESS;
}
#endif // CC_PolicyPCR
```


23.8 TPM2_PolicyLocality

23.8.1 General Description

This command indicates that the authorization will be limited to a specific locality.

policySession→commandLocality is a parameter kept in the session context. When the policy session is started, this parameter is initialized to a value that allows the policy to apply to any locality.

If *locality* has a value greater than 31, then an extended locality is indicated. For an extended locality, the TPM will validate that *policySession→commandLocality* has not previously been set or that the current value of *policySession→commandLocality* is the same as *locality* (TPM_RC_RANGE).

When *locality* is not an extended locality, the TPM will validate that the *policySession→commandLocality* is not set to an extended locality value (TPM_RC_RANGE). If not the TPM will disable any locality not SET in the *locality* parameter. If the result of disabling localities results in no locality being enabled, the TPM will return TPM_RC_RANGE.

If no error occurred in the validation of *locality*, *policySession→policyDigest* is extended with

$$policyDigest_{new} \coloneqq \mathbf{H}_{policyAlg}(policyDigest_{old} \mathbin{||} \text{TPM\_CC\_PolicyLocality} \mathbin{||} locality) \qquad (21)$$

Then *policySession→commandLocality* is updated to indicate which localities are still allowed after execution of TPM2_PolicyLocality().

When the policy session is used to authorize a command, the authorization will fail if the locality used for the command is not one of the enabled localities in *policySession→commandLocality*.

23.8.2 Command and Response

Table 134 — TPM2_PolicyLocality Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PolicyLocality
TPMI_SH_POLICY	policySession	handle for the policy session being extended Auth Index: None
TPMA_LOCALITY	locality	the allowed localities for the policy

Table 135 — TPM2_PolicyLocality Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

23.8.3 Detailed Actions

```
#include "Tpm.h"
#include "PolicyLocality_fp.h"
#if CC_PolicyLocality  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_RANGE	all the locality values selected by *locality* have been disabled by previous TPM2_PolicyLocality() calls.

```
TPM_RC
TPM2_PolicyLocality(
    PolicyLocality_In   *in             // IN: input parameter list
    )
{
    SESSION     *session;
    BYTE         marshalBuffer[sizeof(TPMA_LOCALITY)];
    BYTE         prevSetting[sizeof(TPMA_LOCALITY)];
    UINT32       marshalSize;
    BYTE        *buffer;
    TPM_CC       commandCode = TPM_CC_PolicyLocality;
    HASH_STATE   hashState;

// Input Validation

    // Get pointer to the session structure
    session = SessionGet(in->policySession);

    // Get new locality setting in canonical form
    marshalBuffer[0] = 0;   // Code analysis says that this is not initialized
    buffer = marshalBuffer;
    marshalSize = TPMA_LOCALITY_Marshal(&in->locality, &buffer, NULL);

    // Its an error if the locality parameter is zero
    if(marshalBuffer[0] == 0)
        return TPM_RCS_RANGE + RC_PolicyLocality_locality;

    // Get existing locality setting in canonical form
    prevSetting[0] = 0;     // Code analysis says that this is not initialized
    buffer = prevSetting;
    TPMA_LOCALITY_Marshal(&session->commandLocality, &buffer, NULL);

    // If the locality has previously been set
    if(prevSetting[0] != 0
        // then the current locality setting and the requested have to be the same
        // type (that is, either both normal or both extended
        && ((prevSetting[0] < 32) != (marshalBuffer[0] < 32)))
        return TPM_RCS_RANGE + RC_PolicyLocality_locality;

    // See if the input is a regular or extended locality
    if(marshalBuffer[0] < 32)
    {
        // if there was no previous setting, start with all normal localities
        // enabled
        if(prevSetting[0] == 0)
            prevSetting[0] = 0x1F;

        // AND the new setting with the previous setting and store it in prevSetting
        prevSetting[0] &= marshalBuffer[0];

        // The result setting can not be 0
        if(prevSetting[0] == 0)
```

```
            return TPM_RCS_RANGE + RC_PolicyLocality_locality;
    }
    else
    {
        // for extended locality
        // if the locality has already been set, then it must match the
        if(prevSetting[0] != 0 && prevSetting[0] != marshalBuffer[0])
            return TPM_RCS_RANGE + RC_PolicyLocality_locality;

        // Setting is OK
        prevSetting[0] = marshalBuffer[0];
    }

// Internal Data Update

    // Update policy hash
    // policyDigestnew = hash(policyDigestold || TPM_CC_PolicyLocality || locality)
    // Start hash
    CryptHashStart(&hashState, session->authHashAlg);

    // add old digest
    CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);

    // add commandCode
    CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), commandCode);

    // add input locality
    CryptDigestUpdate(&hashState, marshalSize, marshalBuffer);

    // complete the digest
    CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);

    // update session locality by unmarshal function.  The function must succeed
    // because both input and existing locality setting have been validated.
    buffer = prevSetting;
    TPMA_LOCALITY_Unmarshal(&session->commandLocality, &buffer,
                            (INT32 *)&marshalSize);

    return TPM_RC_SUCCESS;
}
#endif // CC_PolicyLocality
```


23.9 TPM2_PolicyNV

23.9.1 General Description

This command is used to cause conditional gating of a policy based on the contents of an NV Index. It is an immediate assertion. The NV index is validated during the TPM2_PolicyNV() command, not when the session is used for authorization.

The authorization to read the NV Index must succeed even if *policySession* is a trial policy session.

If *policySession* is a trial policy session, the TPM will update *policySession→policyDigest* as shown in equations (22) and (23) below and return TPM_RC_SUCCESS. It will not perform any further validation. The remainder of this general description would apply only if *policySession* is not a trial policy session.

An authorization session providing authorization to read the NV Index shall be provided.

If TPMA_NV_WRITTEN is not SET in the NV Index, the TPM shall return TPM_RC_NV_UNINITIALIZED. If TPMA_NV_READLOCKED of the NV Index is SET, then the TPM shall return TPM_RC_NV_LOCKED.

For an NV Index with the TPM_NT_COUNTER or TPM_NT_BITS attribute SET, the TPM may ignore the *offset* parameter and use an offset of 0. Therefore, it is recommended that the caller set the *offset* parameter to 0 for interoperability.

If *offset* and the *size* field of *data* add to a value that is greater than the *dataSize* field of the NV Index referenced by *nvIndex*, the TPM shall return an error (TPM_RC_NV_RANGE). The implementation may return an error (TPM_RC_VALUE) if it performs an additional check and determines that *offset* is greater than the *dataSize* field of the NV Index.

operandA begins at *offset* into the NV index contents and has a size equal to the size of *operandB*. The TPM will perform the indicated arithmetic check using *operandA* and *operandB*. If the check fails, the TPM shall return TPM_RC_POLICY and not change *policySession→policyDigest*. If the check succeeds, the TPM will hash the arguments:

$$args \coloneqq \mathbf{H}_{policyAlg}(operandB.buffer \mid\mid offset \mid\mid operation) \tag{22}$$

where

$\mathbf{H}_{policyAlg}()$	hash function using the algorithm of the policy session
$operandB$	the value used for the comparison
$offset$	offset from the start of the NV Index data to start the comparison
$operation$	the operation parameter indicating the comparison being performed

The value of args and the Name of the NV Index are extended to *policySession→policyDigest* by

$$policyDigest_{new} \coloneqq \mathbf{H}_{policyAlg}(policyDigest_{old} \mid\mid \text{TPM\_CC\_PolicyNV} \mid\mid args \mid\mid nvIndex{\to}Name) \tag{23}$$

where

$\mathbf{H}_{policyAlg}()$	hash function using the algorithm of the policy session
$args$	value computed in equation (22)
$nvIndex{\to}Name$	the Name of the NV Index

The signed arithmetic operations are performed using twos-compliment.

Magnitude comparisons assume that the octet at offset zero in the referenced NV location and in *operandB* contain the most significant octet of the data.

23.9.2 Command and Response

Table 136 — TPM2_PolicyNV Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PolicyNV
TPMI_RH_NV_AUTH	@authHandle	handle indicating the source of the authorization value Auth Index: 1 Auth Role: USER
TPMI_RH_NV_INDEX	nvIndex	the NV Index of the area to read Auth Index: None
TPMI_SH_POLICY	policySession	handle for the policy session being extended Auth Index: None
TPM2B_OPERAND	operandB	the second operand
UINT16	offset	the octet offset in the NV Index for the start of operand A
TPM_EO	operation	the comparison to make

Table 137 — TPM2_PolicyNV Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

23.9.3 Detailed Actions

```
#include "Tpm.h"
#include "PolicyNV_fp.h"
#if CC_PolicyNV  // Conditional expansion of this file
#include "Policy_spt_fp.h"
```

Error Returns	Meaning
TPM_RC_AUTH_TYPE	NV index authorization type is not correct
TPM_RC_NV_LOCKED	NV index read locked
TPM_RC_NV_UNINITIALIZED	the NV index has not been initialized
TPM_RC_POLICY	the comparison to the NV contents failed
TPM_RC_SIZE	the size of *nvIndex* data starting at *offset* is less than the size of *operandB*
TPM_RC_VALUE	*offset* is too large

```
TPM_RC
TPM2_PolicyNV(
    PolicyNV_In     *in             // IN: input parameter list
    )
{
    TPM_RC               result;
    SESSION             *session;
    NV_REF               locator;
    NV_INDEX            *nvIndex;
    BYTE                 nvBuffer[sizeof(in->operandB.t.buffer)];
    TPM2B_NAME           nvName;
    TPM_CC               commandCode = TPM_CC_PolicyNV;
    HASH_STATE           hashState;
    TPM2B_DIGEST         argHash;

// Input Validation

    // Get pointer to the session structure
    session = SessionGet(in->policySession);

    //If this is a trial policy, skip all validations and the operation
    if(session->attributes.isTrialPolicy == CLEAR)
    {
        // No need to access the actual NV index information for a trial policy.
        nvIndex = NvGetIndexInfo(in->nvIndex, &locator);

        // Common read access checks. NvReadAccessChecks() may return
        // TPM_RC_NV_AUTHORIZATION, TPM_RC_NV_LOCKED, or TPM_RC_NV_UNINITIALIZED
        result = NvReadAccessChecks(in->authHandle,
                                    in->nvIndex,
                                    nvIndex->publicArea.attributes);
        if(result != TPM_RC_SUCCESS)
            return result;

        // Make sure that offset is withing range
        if(in->offset > nvIndex->publicArea.dataSize)
            return TPM_RCS_VALUE + RC_PolicyNV_offset;

        // Valid NV data size should not be smaller than input operandB size
        if((nvIndex->publicArea.dataSize - in->offset) < in->operandB.t.size)
            return TPM_RCS_SIZE + RC_PolicyNV_operandB;

```



```
        // Get NV data.  The size of NV data equals the input operand B size
        NvGetIndexData(nvIndex, locator, in->offset, in->operandB.t.size, nvBuffer);

        // Check to see if the condition is valid
        if(!PolicySptCheckCondition(in->operation, nvBuffer,
                                    in->operandB.t.buffer, in->operandB.t.size))
            return TPM_RC_POLICY;
    }
// Internal Data Update

    // Start argument hash
    argHash.t.size = CryptHashStart(&hashState, session->authHashAlg);

    //  add operandB
    CryptDigestUpdate2B(&hashState, &in->operandB.b);

    //  add offset
    CryptDigestUpdateInt(&hashState, sizeof(UINT16), in->offset);

    //  add operation
    CryptDigestUpdateInt(&hashState, sizeof(TPM_EO), in->operation);

    //  complete argument digest
    CryptHashEnd2B(&hashState, &argHash.b);

    // Update policyDigest
    //  Start digest
    CryptHashStart(&hashState, session->authHashAlg);

    //  add old digest
    CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);

    //  add commandCode
    CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), commandCode);

    //  add argument digest
    CryptDigestUpdate2B(&hashState, &argHash.b);

    // Adding nvName
    CryptDigestUpdate2B(&hashState, &EntityGetName(in->nvIndex, &nvName)->b);

    // complete the digest
    CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);

    return TPM_RC_SUCCESS;
}
#endif // CC_PolicyNV
```

23.10 TPM2_PolicyCounterTimer

23.10.1 General Description

This command is used to cause conditional gating of a policy based on the contents of the TPMS_TIME_INFO structure.

If *policySession* is a trial policy session, the TPM will update *policySession→policyDigest* as shown in equations (24) and (25) below and return TPM_RC_SUCCESS. It will not perform any validation. The remainder of this general description would apply only if *policySession* is not a trial policy session.

The TPM will perform the indicated arithmetic check on the indicated portion of the TPMS_TIME_INFO structure. If the check fails, the TPM shall return TPM_RC_POLICY and not change *policySession→policyDigest*. If the check succeeds, the TPM will hash the arguments:

$$args \coloneqq \mathbf{H}_{policyAlg}(operandB.buffer \mathbin{||} offset \mathbin{||} operation) \tag{24}$$

where

$\mathbf{H}_{policyAlg}()$	hash function using the algorithm of the policy session
$operandB.buffer$	the value used for the comparison
$offset$	offset from the start of the TPMS_TIME_INFO structure at which the comparison starts
$operation$	the operation parameter indicating the comparison being performed

NOTE There is no security related reason for the double hash.

The value of *args* is extended to *policySession→policyDigest* by

$$policyDigest_{new} \coloneqq \mathbf{H}_{policyAlg}(policyDigest_{old} \mathbin{||} \text{TPM\_CC\_PolicyCounterTimer} \mathbin{||} args) \tag{25}$$

where

$\mathbf{H}_{policyAlg}()$	hash function using the algorithm of the policy session
$args$	value computed in equation (24)

The signed arithmetic operations are performed using twos-compliment. The indicated portion of the TPMS_TIME_INFO structure begins at *offset* and has a length of *operandB.size*. If the number of octets to be compared overflows the TPMS_TIME_INFO structure, the TPM returns TPM_RC_RANGE. If *offset* is greater than the size of the marshaled TPMS_TIME_INFO structure, the TPM returns TPM_RC_VALUE. The structure is marshaled into its canonical form with no padding. The TPM does not check for alignment of the offset with a TPMS_TIME_INFO structure member.

Magnitude comparisons assume that the octet at offset zero in the referenced location and in *operandB* contain the most significant octet of the data.

23.10.2 Command and Response

Table 138 — TPM2_PolicyCounterTimer Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit or decrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PolicyCounterTimer
TPMI_SH_POLICY	policySession	handle for the policy session being extended Auth Index: None
TPM2B_OPERAND	operandB	the second operand
UINT16	offset	the octet offset in the TPMS_TIME_INFO structure for the start of operand A
TPM_EO	operation	the comparison to make

Table 139 — TPM2_PolicyCounterTimer Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

23.10.3 Detailed Actions

```
#include "Tpm.h"
#include "PolicyCounterTimer_fp.h"
#if CC_PolicyCounterTimer  // Conditional expansion of this file
#include "Policy_spt_fp.h"
```

Error Returns	Meaning
TPM_RC_POLICY	the comparison of the selected portion of the TPMS_TIME_INFO with *operandB* failed
TPM_RC_RANGE	*offset* + *size* exceed size of TPMS_TIME_INFO structure

```
TPM_RC
TPM2_PolicyCounterTimer(
    PolicyCounterTimer_In   *in             // IN: input parameter list
    )
{
    SESSION                 *session;
    TIME_INFO                infoData;          // data buffer of  TPMS_TIME_INFO
    BYTE                    *pInfoData = (BYTE *)&infoData;
    UINT16                   infoDataSize;
    TPM_CC                   commandCode = TPM_CC_PolicyCounterTimer;
    HASH_STATE               hashState;
    TPM2B_DIGEST             argHash;

// Input Validation
    // Get a marshaled time structure
    infoDataSize = TimeGetMarshaled(&infoData);
    // Make sure that the referenced stays within the bounds of the structure.
    // NOTE: the offset checks are made even for a trial policy because the policy
    // will not make any sense if the references are out of bounds of the timer
    // structure.
    if(in->offset > infoDataSize)
        return TPM_RCS_VALUE + RC_PolicyCounterTimer_offset;
    if((UINT32)in->offset + (UINT32)in->operandB.t.size > infoDataSize)
        return TPM_RCS_RANGE;
    // Get pointer to the session structure
    session = SessionGet(in->policySession);

    //If this is a trial policy, skip the check to see if the condition is met.
    if(session->attributes.isTrialPolicy == CLEAR)
    {
        // If the command is going to use any part of the counter or timer, need
        // to verify that time is advancing.
        // The time and clock vales are the first two 64-bit values in the clock
        if(in->offset < sizeof(UINT64) + sizeof(UINT64))
        {
            // Using Clock or Time so see if clock is running. Clock doesn't
            // run while NV is unavailable.
            // TPM_RC_NV_UNAVAILABLE or TPM_RC_NV_RATE error may be returned here.
            RETURN_IF_NV_IS_NOT_AVAILABLE;
        }
        // offset to the starting position
        pInfoData = (BYTE *)infoData;
        // Check to see if the condition is valid
        if(!PolicySptCheckCondition(in->operation, pInfoData + in->offset,
                                    in->operandB.t.buffer, in->operandB.t.size))
            return TPM_RC_POLICY;
    }
// Internal Data Update
    // Start argument list hash
```

```
    argHash.t.size = CryptHashStart(&hashState, session->authHashAlg);
    //  add operandB
    CryptDigestUpdate2B(&hashState, &in->operandB.b);
    //  add offset
    CryptDigestUpdateInt(&hashState, sizeof(UINT16), in->offset);
    //  add operation
    CryptDigestUpdateInt(&hashState, sizeof(TPM_EO), in->operation);
    //  complete argument hash
    CryptHashEnd2B(&hashState, &argHash.b);

    // update policyDigest
    //  start hash
    CryptHashStart(&hashState, session->authHashAlg);

    //  add old digest
    CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);

    //  add commandCode
    CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), commandCode);

    //  add argument digest
    CryptDigestUpdate2B(&hashState, &argHash.b);

    // complete the digest
    CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);

    return TPM_RC_SUCCESS;
}
#endif // CC_PolicyCounterTimer
```

23.11 TPM2_PolicyCommandCode

23.11.1 General Description

This command indicates that the authorization will be limited to a specific command code.

If *policySession→commandCode* has its default value, then it will be set to *code.* If *policySession→commandCode* does not have its default value, then the TPM will return TPM_RC_VALUE if the two values are not the same.

If *code* is not implemented, the TPM will return TPM_RC_POLICY_CC.

If the TPM does not return an error, it will update *policySession→policyDigest* by

$$policyDigest_{new} \coloneqq \mathbf{H}_{policyAlg}(policyDigest_{old} \mathbin{||} \text{TPM\_CC\_PolicyCommandCode} \mathbin{||} code) \tag{26}$$

NOTE 1 If a previous TPM2_PolicyCommandCode() had been executed, then it is probable that the policy expression is improperly formed but the TPM does not return an error if *code* is the same.

NOTE 2 A TPM2_PolicyOR() would be used to allow an authorization to be used for multiple commands.

When the policy session is used to authorize a command, the TPM will fail the command if the *commandCode* of that command does not match *policySession→commandCode*.

This command, or TPM2_PolicyDuplicationSelect(), is required to enable the policy to be used for ADMIN role authorization.

EXAMPLE Before TPM2_Certify() can be executed, TPM2_PolicyCommandCode() with code set to TPM_CC_Certify is required.

23.11.2 Command and Response

Table 140 — TPM2_PolicyCommandCode Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PolicyCommandCode
TPMI_SH_POLICY	policySession	handle for the policy session being extended Auth Index: None
TPM_CC	code	the allowed *commandCode*

Table 141 — TPM2_PolicyCommandCode Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

23.11.3 Detailed Actions

```
#include "Tpm.h"
#include "PolicyCommandCode_fp.h"
#if CC_PolicyCommandCode  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_VALUE	*commandCode* of *policySession* previously set to a different value

```
TPM_RC
TPM2_PolicyCommandCode(
    PolicyCommandCode_In    *in             // IN: input parameter list
    )
{
    SESSION     *session;
    TPM_CC      commandCode = TPM_CC_PolicyCommandCode;
    HASH_STATE  hashState;

// Input validation

    // Get pointer to the session structure
    session = SessionGet(in->policySession);

    if(session->commandCode != 0 && session->commandCode != in->code)
            return TPM_RCS_VALUE + RC_PolicyCommandCode_code;
    if(CommandCodeToCommandIndex(in->code) == UNIMPLEMENTED_COMMAND_INDEX)
        return TPM_RCS_POLICY_CC + RC_PolicyCommandCode_code;

// Internal Data Update
    // Update policy hash
    // policyDigestnew = hash(policyDigestold || TPM_CC_PolicyCommandCode || code)
    //  Start hash
    CryptHashStart(&hashState, session->authHashAlg);

    //  add old digest
    CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);

    //  add commandCode
    CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), commandCode);

    //  add input commandCode
    CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), in->code);

    //  complete the hash and get the results
    CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);

    // update commandCode value in session context
    session->commandCode = in->code;

    return TPM_RC_SUCCESS;
}
#endif // CC_PolicyCommandCode
```

23.12 TPM2_PolicyPhysicalPresence

23.12.1 General Description

This command indicates that physical presence will need to be asserted at the time the authorization is performed.

If this command is successful, *policySession→isPPRequired* will be SET to indicate that this check is required when the policy is used for authorization. Additionally, *policySession→policyDigest* is extended with

$$policyDigest_{new} \coloneqq \mathbf{H}_{policyAlg}(policyDigest_{old} \mathbin{||} \text{TPM\_CC\_PolicyPhysicalPresence}) \quad (27)$$

23.12.2 Command and Response

Table 142 — TPM2_PolicyPhysicalPresence Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PolicyPhysicalPresence
TPMI_SH_POLICY	policySession	handle for the policy session being extended Auth Index: None

Table 143 — TPM2_PolicyPhysicalPresence Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

23.12.3 Detailed Actions

```
#include "Tpm.h"
#include "PolicyPhysicalPresence_fp.h"
#if CC_PolicyPhysicalPresence  // Conditional expansion of this file
TPM_RC
TPM2_PolicyPhysicalPresence(
    PolicyPhysicalPresence_In   *in             // IN: input parameter list
    )
{
    SESSION     *session;
    TPM_CC      commandCode = TPM_CC_PolicyPhysicalPresence;
    HASH_STATE  hashState;

// Internal Data Update

    // Get pointer to the session structure
    session = SessionGet(in->policySession);

    // Update policy hash
    // policyDigestnew = hash(policyDigestold || TPM_CC_PolicyPhysicalPresence)
    //  Start hash
    CryptHashStart(&hashState, session->authHashAlg);

    //  add old digest
    CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);

    //  add commandCode
    CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), commandCode);

    //  complete the digest
    CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);

    // update session attribute
    session->attributes.isPPRequired = SET;

    return TPM_RC_SUCCESS;
}
#endif // CC_PolicyPhysicalPresence
```


23.13 TPM2_PolicyCpHash

23.13.1 General Description

This command is used to allow a policy to be bound to a specific command and command parameters.

TPM2_PolicySigned(), TPM2_PolicySecret(), and TPM2_PolicyTIcket() are designed to allow an authorizing entity to execute an arbitrary command as the *cpHashA* parameter of those commands is not included in *policySession→policyDigest*. TPM2_PolicyCommandCode() allows the policy to be bound to a specific Command Code so that only certain entities may authorize specific command codes. This command allows the policy to be restricted such that an entity may only authorize a command with a specific set of parameters.

If *policySession→cpHash* is already set and not the same as *cpHashA*, then the TPM shall return TPM_RC_CPHASH. If *cpHashA* does not have the size of the *policySession→policyDigest*, the TPM shall return TPM_RC_SIZE.

NOTE 1 If a previous TPM2_PolicyCpHash() had been executed, then it is probable that the policy expression is improperly formed but the TPM does not return an error if *cpHash* is the same.

If the *cpHashA* checks succeed, *policySession→cpHash* is set to *cpHashA* and *policySession→policyDigest* is updated with

$$policyDigest_{new} \coloneqq \mathbf{H}_{policyAlg}(policyDigest_{old} \,||\, \text{TPM\_CC\_PolicyCpHash} \,||\, cpHashA) \tag{28}$$

23.13.2 Command and Response

Table 144 — TPM2_PolicyCpHash Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit or decrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PolicyCpHash
TPMI_SH_POLICY	policySession	handle for the policy session being extended Auth Index: None
TPM2B_DIGEST	cpHashA	the *cpHash* added to the policy

Table 145 — TPM2_PolicyCpHash Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

23.13.3 Detailed Actions

```
#include "Tpm.h"
#include "PolicyCpHash_fp.h"
#if CC_PolicyCpHash  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_CPHASH	*cpHash* of *policySession* has previously been set to a different value
TPM_RC_SIZE	*cpHashA* is not the size of a digest produced by the hash algorithm associated with *policySession*

```
TPM_RC
TPM2_PolicyCpHash(
    PolicyCpHash_In     *in             // IN: input parameter list
    )
{
    SESSION     *session;
    TPM_CC      commandCode = TPM_CC_PolicyCpHash;
    HASH_STATE  hashState;

// Input Validation

    // Get pointer to the session structure
    session = SessionGet(in->policySession);

    // A valid cpHash must have the same size as session hash digest
    // NOTE: the size of the digest can't be zero because TPM_ALG_NULL
    // can't be used for the authHashAlg.
    if(in->cpHashA.t.size != CryptHashGetDigestSize(session->authHashAlg))
        return TPM_RCS_SIZE + RC_PolicyCpHash_cpHashA;

    // error if the cpHash in session context is not empty and is not the same
    // as the input or is not a cpHash
    if((session->u1.cpHash.t.size != 0)
       && (!session->attributes.isCpHashDefined
           || !MemoryEqual2B(&in->cpHashA.b, &session->u1.cpHash.b)))
        return TPM_RC_CPHASH;

// Internal Data Update

    // Update policy hash
    // policyDigestnew = hash(policyDigestold || TPM_CC_PolicyCpHash || cpHashA)
    //  Start hash
    CryptHashStart(&hashState, session->authHashAlg);

    //  add old digest
    CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);

    //  add commandCode
    CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), commandCode);

    //  add cpHashA
    CryptDigestUpdate2B(&hashState, &in->cpHashA.b);

    //  complete the digest and get the results
    CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);

    // update cpHash in session context
    session->u1.cpHash = in->cpHashA;
    session->attributes.isCpHashDefined = SET;

```

```
    return TPM_RC_SUCCESS;
}
#endif // CC_PolicyCpHash
```


23.14 TPM2_PolicyNameHash

23.14.1 General Description

This command allows a policy to be bound to a specific set of TPM entities without being bound to the parameters of the command. This is most useful for commands such as TPM2_Duplicate() and for TPM2_PCR_Event() when the referenced PCR requires a policy.

The *nameHash* parameter should contain the digest of the Names associated with the handles to be used in the authorized command.

EXAMPLE For the TPM2_Duplicate() command, two handles are provided. One is the handle of the object being duplicated and the other is the handle of the new parent. For that command, *nameHash* would contain:

$$nameHash \coloneqq \mathbf{H}_{policyAlg}(objectHandle \rightarrow Name \mid\mid newParentHandle \rightarrow Name)$$

If *policySession→cpHash* is already set, the TPM shall return TPM_RC_CPHASH. If the size of *nameHash* is not the size of *policySession→policyDigest*, the TPM shall return TPM_RC_SIZE. Otherwise, *policySession→cpHash* is set to *nameHash*.

If this command completes successfully, the *cpHash* of the authorized command will not be used for validation. Only the digest of the Names associated with the handles in the command will be used.

NOTE 1 This allows the space normally used to hold *policySession→cpHash* to be used for *policySession→nameHash* instead.

The *policySession→policyDigest* will be updated with

$$policyDigest_{new} \coloneqq \mathbf{H}_{policyAlg}(policyDigest_{old} \mid\mid \text{TPM\_CC\_PolicyNameHash} \mid\mid nameHash) \qquad (29)$$

NOTE 2 This command can only be used with TPM2_PolicyAuthorize() or TPM2_PolicyAuthorizeNV. The owner of the object being duplicated provides approval for their object to be migrated to a specific new parent.

Without this approval, the Name of the Object would need to be known at the time that Object's policy is created. However, since the Name of the Object includes its policy, the Name is not known. The Name can be known by the authorizing entity.

23.14.2 Command and Response

Table 146 — TPM2_PolicyNameHash Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit or decrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PolicyNameHash
TPMI_SH_POLICY	policySession	handle for the policy session being extended Auth Index: None
TPM2B_DIGEST	nameHash	the digest to be added to the policy

Table 147 — TPM2_PolicyNameHash Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

23.14.3 Detailed Actions

```c
#include "Tpm.h"
#include "PolicyNameHash_fp.h"
#if CC_PolicyNameHash  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_CPHASH	*nameHash* has been previously set to a different value
TPM_RC_SIZE	*nameHash* is not the size of the digest produced by the hash algorithm associated with *policySession*

```c
TPM_RC
TPM2_PolicyNameHash(
    PolicyNameHash_In    *in              // IN: input parameter list
    )
{
    SESSION                 *session;
    TPM_CC                   commandCode = TPM_CC_PolicyNameHash;
    HASH_STATE               hashState;

// Input Validation

    // Get pointer to the session structure
    session = SessionGet(in->policySession);

    // A valid nameHash must have the same size as session hash digest
    // Since the authHashAlg for a session cannot be TPM_ALG_NULL, the digest size
    // is always non-zero.
    if(in->nameHash.t.size != CryptHashGetDigestSize(session->authHashAlg))
        return TPM_RCS_SIZE + RC_PolicyNameHash_nameHash;

    // u1 in the policy session context cannot otherwise be occupied
    if(session->u1.cpHash.b.size != 0
       || session->attributes.isBound
       || session->attributes.isCpHashDefined
       || session->attributes.isTemplateSet)
        return TPM_RC_CPHASH;

// Internal Data Update

    // Update policy hash
    // policyDigestnew = hash(policyDigestold || TPM_CC_PolicyNameHash || nameHash)
    //  Start hash
    CryptHashStart(&hashState, session->authHashAlg);

    //  add old digest
    CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);

    //  add commandCode
    CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), commandCode);

    //  add nameHash
    CryptDigestUpdate2B(&hashState, &in->nameHash.b);

    //  complete the digest
    CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);

    // update nameHash in session context
    session->u1.cpHash = in->nameHash;

    return TPM_RC_SUCCESS;
```



```
}
#endif // CC_PolicyNameHash
```

23.15 TPM2_PolicyDuplicationSelect

23.15.1 General Description

This command allows qualification of duplication to allow duplication to a selected new parent.

If this command not used in conjunction with a PolicyAuthorize Command, then only the new parent is selected and *includeObject* should be CLEAR.

EXAMPLE When an object is created when the list of allowed duplication targets is known, the policy would be created with *includeObject* CLEAR.

NOTE 1 Only the new parent may be selected because, without TPM2_PolicyAuthorize(), the Name of the Object to be duplicated would need to be known at the time that Object's policy is created. However, since the Name of the Object includes its policy, the Name is not known. The Name can be known by the authorizing entity (a PolicyAuthorize Command) in which case *includeObject* may be SET.

If used in conjunction with TPM2_PolicyAuthorize(), then the authorizer of the new policy has the option of selecting just the new parent or of selecting both the new parent and the duplication Object.

NOTE 2 If the authorizing entity for an TPM2_PolicyAuthorize() only specifies the new parent, then that authorization may be applied to the duplication of any number of other Objects. If the authorizing entity specifies both a new parent and the duplicated Object, then the authorization only applies to that pairing of Object and new parent.

If either *policySession→cpHash* or *policySession→nameHash* has been previously set, the TPM shall return TPM_RC_CPHASH. Otherwise, *policySession→nameHash* will be set to:

$$nameHash \coloneqq \mathbf{H}_{policyAlg}(objectName.name \,||\, newParentName.name) \quad (30)$$

NOTE 3 It is allowed that policySesion→nameHash and policySession→cpHash share the same memory space.

NOTE 4 The Name in these equations uses Name.name, indicating that the UINT16 size is not included in the hash.

The *policySession→policyDigest* will be updated according to the setting of *includeObject*. If equal to YES, *policySession→policyDigest* is updated by:

$$policyDigest_{new} \coloneqq \mathbf{H}_{policyAlg}(policyDigest_{old} \,||\, \text{TPM\_CC\_PolicyDuplicationSelect} \,||\, objectName\,.name \,||\, newParentName.name \,||\, includeObject) \quad (31)$$

If includeObject is NO, *policySession→policyDigest* is updated by:

$$policyDigest_{new} \coloneqq \mathbf{H}_{policyAlg}(policyDigest_{old} \,||\, \text{TPM\_CC\_PolicyDuplicationSelect} \,||\, newParentName.name \,||\, includeObject) \quad (32)$$

NOTE 5 *policySession→nameHash* receives the digest of both Names so that the check performed in TPM2_Duplicate() may be the same regardless of which Names are included in *policySession→policyDigest*. This means that, when TPM2_PolicyDuplicationSelect() is executed, it is only valid for a specific pair of duplication object and new parent.

If the command succeeds, *policySession→commandCode* is set to TPM_CC_Duplicate.

NOTE 6 The normal use of this command is before a TPM2_PolicyAuthorize(). An authorized entity would approve a *policyDigest* that allowed duplication to a specific new parent. The authorizing entity may want to limit the authorization so that the approval allows only a specific object to be duplicated to the new parent. In that case, the authorizing entity would approve the *policyDigest* of *equation* (31).

23.15.2 Command and Response

Table 148 — TPM2_PolicyDuplicationSelect Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit or decrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PolicyDuplicationSelect
TPMI_SH_POLICY	policySession	handle for the policy session being extended Auth Index: None
TPM2B_NAME	objectName	the Name of the object to be duplicated
TPM2B_NAME	newParentName	the Name of the new parent
TPMI_YES_NO	includeObject	if YES, the *objectName* will be included in the value in *policySession→policyDigest*

Table 149 — TPM2_PolicyDuplicationSelect Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

23.15.3 Detailed Actions

```c
#include "Tpm.h"
#include "PolicyDuplicationSelect_fp.h"
#if CC_PolicyDuplicationSelect  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_COMMAND_CODE	*commandCode* of '*policySession*; is not empty
TPM_RC_CPHASH	*cpHash* of *policySession* is not empty

```c
TPM_RC
TPM2_PolicyDuplicationSelect(
    PolicyDuplicationSelect_In  *in             // IN: input parameter list
    )
{
    SESSION         *session;
    HASH_STATE      hashState;
    TPM_CC          commandCode = TPM_CC_PolicyDuplicationSelect;

// Input Validation

    // Get pointer to the session structure
    session = SessionGet(in->policySession);

    // cpHash in session context must be empty
    if(session->u1.cpHash.t.size != 0)
        return TPM_RC_CPHASH;

    // commandCode in session context must be empty
    if(session->commandCode != 0)
        return TPM_RC_COMMAND_CODE;

// Internal Data Update

    // Update name hash
    session->u1.cpHash.t.size = CryptHashStart(&hashState, session->authHashAlg);

    //  add objectName
    CryptDigestUpdate2B(&hashState, &in->objectName.b);

    //  add new parent name
    CryptDigestUpdate2B(&hashState, &in->newParentName.b);

    //  complete hash
    CryptHashEnd2B(&hashState, &session->u1.cpHash.b);

    // update policy hash
    // Old policyDigest size should be the same as the new policyDigest size since
    // they are using the same hash algorithm
    session->u2.policyDigest.t.size
        = CryptHashStart(&hashState, session->authHashAlg);
//  add old policy
    CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);

    //  add command code
    CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), commandCode);

    //  add objectName
    if(in->includeObject == YES)
        CryptDigestUpdate2B(&hashState, &in->objectName.b);

```



```
    //  add new parent name
    CryptDigestUpdate2B(&hashState, &in->newParentName.b);

    //  add includeObject
    CryptDigestUpdateInt(&hashState, sizeof(TPMI_YES_NO), in->includeObject);

    //  complete digest
    CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);

    // set commandCode in session context
    session->commandCode = TPM_CC_Duplicate;

    return TPM_RC_SUCCESS;
}
#endif // CC_PolicyDuplicationSelect
```

23.16 TPM2_PolicyAuthorize

23.16.1 General Description

This command allows policies to change. If a policy were static, then it would be difficult to add users to a policy. This command lets a policy authority sign a new policy so that it may be used in an existing policy.

The authorizing entity signs a structure that contains

$$aHash \coloneqq \mathbf{H}_{aHashAlg}(approvedPolicy \;||\; policyRef) \tag{33}$$

The *aHashAlg* is required to be the *nameAlg* of the key used to sign the *aHash*. The *aHash* value is then signed (symmetric or asymmetric) by *keySign*. That signature is then checked by the TPM in 20.1 TPM2_VerifySignature() which produces a ticket by

$$\mathbf{HMAC}(proof, (\text{TPM\_ST\_VERIFIED} \;||\; aHash \;||\; keySign\rightarrow Name)) \tag{34}$$

NOTE 1 The reason for the validation is because of the expectation that the policy will be used multiple times and it is more efficient to check a ticket than to load an object each time to check a signature.

The ticket is then used in TPM2_PolicyAuthorize() to validate the parameters.

The *keySign* parameter is required to be a valid object name using nameAlg other than TPM_ALG_NULL. If the first two octets of *keySign* are not a valid hash algorithm, the TPM shall return TPM_RC_HASH. If the remainder of the Name is not the size of the indicated digest, the TPM shall return TPM_RC_SIZE.

The TPM validates that the *approvedPolicy* matches the current value of *policySession→policyDigest* and if not, shall return TPM_RC_VALUE.

The TPM then validates that the parameters to TPM2_PolicyAuthorize() match the values used to generate the ticket. If so, the TPM will reset *policySession→policyDigest* to a Zero Digest. Then it will update *policySession→policyDigest* with **PolicyUpdate**() (see 23.2.3).

$$\mathbf{PolicyUpdate}(\text{TPM\_CC\_PolicyAuthorize}, keySign, policyRef) \tag{35}$$

If the ticket is not valid, the TPM shall return TPM_RC_POLICY.

If *policySession* is a trial session, *policySession→policyDigest* is extended as if the ticket is valid without actual verification.

NOTE 2 The unmarshaling process requires that a proper TPMT_TK_VERIFIED be provided for *checkTicket* but it may be a NULL Ticket. A NULL ticket is useful in a trial policy, where the caller uses the TPM to perform policy calculations but does not have a valid authorization ticket.

23.16.2 Command and Response

Table 150 — TPM2_PolicyAuthorize Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit or decrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PolicyAuthorize
TPMI_SH_POLICY	policySession	handle for the policy session being extended Auth Index: None
TPM2B_DIGEST	approvedPolicy	digest of the policy being approved
TPM2B_NONCE	policyRef	a policy qualifier
TPM2B_NAME	keySign	Name of a key that can sign a policy addition
TPMT_TK_VERIFIED	checkTicket	ticket validating that *approvedPolicy* and *policyRef* were signed by *keySign*

Table 151 — TPM2_PolicyAuthorize Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

23.16.3 Detailed Actions

```
#include "Tpm.h"
#include "PolicyAuthorize_fp.h"
#if CC_PolicyAuthorize  // Conditional expansion of this file
#include "Policy_spt_fp.h"
```

Error Returns	Meaning
TPM_RC_HASH	hash algorithm in *keyName* is not supported
TPM_RC_SIZE	*keyName* is not the correct size for its hash algorithm
TPM_RC_VALUE	the current *policyDigest* of *policySession* does not match *approvedPolicy*; or *checkTicket* doesn't match the provided values

```
TPM_RC
TPM2_PolicyAuthorize(
    PolicyAuthorize_In  *in             // IN: input parameter list
    )
{
    SESSION                 *session;
    TPM2B_DIGEST             authHash;
    HASH_STATE               hashState;
    TPMT_TK_VERIFIED         ticket;
    TPM_ALG_ID               hashAlg;
    UINT16                   digestSize;

// Input Validation

    // Get pointer to the session structure
    session = SessionGet(in->policySession);

    // Extract from the Name of the key, the algorithm used to compute it's Name
    hashAlg = BYTE_ARRAY_TO_UINT16(in->keySign.t.name);

    // 'keySign' parameter needs to use a supported hash algorithm, otherwise
    // can't tell how large the digest should be
    if(!CryptHashIsValidAlg(hashAlg, FALSE))
        return TPM_RCS_HASH + RC_PolicyAuthorize_keySign;

    digestSize = CryptHashGetDigestSize(hashAlg);
    if(digestSize != (in->keySign.t.size - 2))
        return TPM_RCS_SIZE + RC_PolicyAuthorize_keySign;

    //If this is a trial policy, skip all validations
    if(session->attributes.isTrialPolicy == CLEAR)
    {
        // Check that "approvedPolicy" matches the current value of the
        // policyDigest in policy session
        if(!MemoryEqual2B(&session->u2.policyDigest.b,
                          &in->approvedPolicy.b))
            return TPM_RCS_VALUE + RC_PolicyAuthorize_approvedPolicy;

        // Validate ticket TPMT_TK_VERIFIED
        // Compute aHash.  The authorizing object sign a digest
        //  aHash := hash(approvedPolicy || policyRef).
        // Start hash
        authHash.t.size = CryptHashStart(&hashState, hashAlg);

        // add approvedPolicy
        CryptDigestUpdate2B(&hashState, &in->approvedPolicy.b);

```

```
        // add policyRef
        CryptDigestUpdate2B(&hashState, &in->policyRef.b);

        // complete hash
        CryptHashEnd2B(&hashState, &authHash.b);

        // re-compute TPMT_TK_VERIFIED
        TicketComputeVerified(in->checkTicket.hierarchy, &authHash,
                              &in->keySign, &ticket);

        // Compare ticket digest.  If not match, return error
        if(!MemoryEqual2B(&in->checkTicket.digest.b, &ticket.digest.b))
            return TPM_RCS_VALUE + RC_PolicyAuthorize_checkTicket;
    }

// Internal Data Update

    // Set policyDigest to zero digest
    PolicyDigestClear(session);

    // Update policyDigest
    PolicyContextUpdate(TPM_CC_PolicyAuthorize, &in->keySign, &in->policyRef,
                        NULL, 0, session);

    return TPM_RC_SUCCESS;
}
#endif // CC_PolicyAuthorize
```

23.17 TPM2_PolicyAuthValue

23.17.1 General Description

This command allows a policy to be bound to the authorization value of the authorized entity.

When this command completes successfully, *policySession→isAuthValueNeeded* is SET to indicate that the *authValue* will be included in *hmacKey* when the authorization HMAC is computed for the command being authorized using this session. Additionally, *policySession→isPasswordNeeded* will be CLEAR.

NOTE If a policy does not use this command, then the *hmacKey* for the authorized command would only use *sessionKey.* If *sessionKey* is not present, then the *hmacKey* is an Empty Buffer and no HMAC would be computed.

If successful, *policySession→policyDigest* will be updated with

$$policyDigest_{new} \coloneqq \mathbf{H}_{policyAlg}(policyDigest_{old} \mid\mid \text{TPM\_CC\_PolicyAuthValue}) \tag{36}$$

23.17.2 Command and Response

Table 152 — TPM2_PolicyAuthValue Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PolicyAuthValue
TPMI_SH_POLICY	policySession	handle for the policy session being extended Auth Index: None

Table 153 — TPM2_PolicyAuthValue Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

23.17.3 Detailed Actions

```
#include "Tpm.h"
#include "PolicyAuthValue_fp.h"
#if CC_PolicyAuthValue  // Conditional expansion of this file
#include "Policy_spt_fp.h"
TPM_RC
TPM2_PolicyAuthValue(
    PolicyAuthValue_In  *in             // IN: input parameter list
    )
{
    SESSION             *session;
    TPM_CC               commandCode = TPM_CC_PolicyAuthValue;
    HASH_STATE           hashState;

// Internal Data Update

    // Get pointer to the session structure
    session = SessionGet(in->policySession);

    // Update policy hash
    // policyDigestnew = hash(policyDigestold || TPM_CC_PolicyAuthValue)
    //   Start hash
    CryptHashStart(&hashState, session->authHashAlg);

    //  add old digest
    CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);

    //  add commandCode
    CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), commandCode);

    //  complete the hash and get the results
    CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);

    // update isAuthValueNeeded bit in the session context
    session->attributes.isAuthValueNeeded = SET;
    session->attributes.isPasswordNeeded = CLEAR;

    return TPM_RC_SUCCESS;
}
#endif // CC_PolicyAuthValue
```

23.18 TPM2_PolicyPassword

23.18.1 General Description

This command allows a policy to be bound to the authorization value of the authorized object.

When this command completes successfully, *policySession→isPasswordNeeded* is SET to indicate that *authValue* of the authorized object will be checked when the session is used for authorization. The caller will provide the *authValue* in clear text in the *hmac* parameter of the authorization. The comparison of *hmac* to *authValue* is performed as if the authorization is a password.

NOTE 1 The parameter field in the policy session where the authorization value is provided is called *hmac*. If TPM2_PolicyPassword() is part of the sequence, then the field will contain a password and not an HMAC.

If successful, *policySession→policyDigest* will be updated with

$$policyDigest_{new} \coloneqq \mathbf{H}_{policyAlg}(policyDigest_{old} \mathbin{||} \text{TPM\_CC\_PolicyAuthValue}) \tag{37}$$

NOTE 2 This is the same extend value as used with TPM2_PolicyAuthValue so that the evaluation may be done using either an HMAC or a password with no change to the *authPolicy* of the object. The reason that two commands are present is to indicate to the TPM if the *hmac* field in the authorization will contain an HMAC or a password value.

When this command is successful, *policySession→isAuthValueNeeded* will be CLEAR.

23.18.2 Command and Response

Table 154 — TPM2_PolicyPassword Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PolicyPassword
TPMI_SH_POLICY	policySession	handle for the policy session being extended Auth Index: None

Table 155 — TPM2_PolicyPassword Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

23.18.3 Detailed Actions

```
#include "Tpm.h"
#include "PolicyPassword_fp.h"
#if CC_PolicyPassword  // Conditional expansion of this file
#include "Policy_spt_fp.h"
TPM_RC
TPM2_PolicyPassword(
    PolicyPassword_In   *in             // IN: input parameter list
    )
{
    SESSION             *session;
    TPM_CC               commandCode = TPM_CC_PolicyAuthValue;
    HASH_STATE           hashState;

// Internal Data Update

    // Get pointer to the session structure
    session = SessionGet(in->policySession);

    // Update policy hash
    // policyDigestnew = hash(policyDigestold || TPM_CC_PolicyAuthValue)
    //  Start hash
    CryptHashStart(&hashState, session->authHashAlg);

    //  add old digest
    CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);

    //  add commandCode
    CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), commandCode);

    //  complete the digest
    CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);

    //  Update isPasswordNeeded bit
    session->attributes.isPasswordNeeded = SET;
    session->attributes.isAuthValueNeeded = CLEAR;

    return TPM_RC_SUCCESS;
}
#endif // CC_PolicyPassword
```

23.19 TPM2_PolicyGetDigest

23.19.1 General Description

This command returns the current *policyDigest* of the session. This command allows the TPM to be used to perform the actions required to pre-compute the *authPolicy* for an object.

23.19.2 Command and Response

Table 156 — TPM2_PolicyGetDigest Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit or encrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PolicyGetDigest
TPMI_SH_POLICY	policySession	handle for the policy session Auth Index: None

Table 157 — TPM2_PolicyGetDigest Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_DIGEST	policyDigest	the current value of the *policySession→policyDigest*

23.19.3 Detailed Actions

```
#include "Tpm.h"
#include "PolicyGetDigest_fp.h"
#if CC_PolicyGetDigest  // Conditional expansion of this file
TPM_RC
TPM2_PolicyGetDigest(
    PolicyGetDigest_In      *in,            // IN: input parameter list
    PolicyGetDigest_Out     *out            // OUT: output parameter list
    )
{
    SESSION     *session;

// Command Output

    // Get pointer to the session structure
    session = SessionGet(in->policySession);

    out->policyDigest = session->u2.policyDigest;

    return TPM_RC_SUCCESS;
}
#endif // CC_PolicyGetDigest
```

23.20 TPM2_PolicyNvWritten

23.20.1 General Description

This command allows a policy to be bound to the TPMA_NV_WRITTEN attributes. This is a deferred assertion. Values are stored in the policy session context and checked when the policy is used for authorization.

If *policySession→checkNVWritten* is CLEAR, it is SET and *policySession→nvWrittenState* is set to *writtenSet*. If *policySession→checkNVWritten* is SET, the TPM will return TPM_RC_VALUE if *policySession→nvWrittenState* and *writtenSet* are not the same.

If the TPM does not return an error, it will update *policySession→policyDigest* by

$$policyDigest_{new} \coloneqq \mathbf{H}_{policyAlg}(policyDigest_{old} \mathbin{||} \text{TPM\_CC\_PolicyNvWritten} \mathbin{||} writtenSet) \tag{38}$$

When the policy session is used to authorize a command, the TPM will fail the command if *policySession→checkNVWritten* is SET and *nvIndex→attributes→TPMA_NV_WRITTEN does not match policySession→nvWrittenState.*

NOTE 1 A typical use case is a simple policy for the first write during manufacturing provisioning that would require TPMA_NV_WRITTEN CLEAR and a more complex policy for later use that would require TPMA_NV_WRITTEN SET.

NOTE 2 When an Index is written, it has a different authorization name than an Index that has not been written. It is possible to use this change in the NV Index to create a write-once Index.

23.20.2 Command and Response

Table 158 — TPM2_PolicyNvWritten Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PolicyNvWritten
TPMI_SH_POLICY	policySession	handle for the policy session being extended Auth Index: None
TPMI_YES_NO	writtenSet	YES if NV Index is required to have been written NO if NV Index is required not to have been written

Table 159 — TPM2_PolicyNvWritten Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

23.20.3 Detailed Actions

```
#include "Tpm.h"
#include "PolicyNvWritten_fp.h"
#if CC_PolicyNvWritten  // Conditional expansion of this file
```

Make an NV Index policy dependent on the state of the TPMA_NV_WRITTEN attribute of the index.

Error Returns	Meaning
TPM_RC_VALUE	a conflicting request for the attribute has already been processed

```
TPM_RC
TPM2_PolicyNvWritten(
    PolicyNvWritten_In  *in             // IN: input parameter list
    )
{
    SESSION     *session;
    TPM_CC       commandCode = TPM_CC_PolicyNvWritten;
    HASH_STATE   hashState;

// Input Validation

    // Get pointer to the session structure
    session = SessionGet(in->policySession);

    // If already set is this a duplicate (the same setting)? If it
    // is a conflicting setting, it is an error
    if(session->attributes.checkNvWritten == SET)
    {
        if(((session->attributes.nvWrittenState == SET)
            != (in->writtenSet == YES)))
            return TPM_RCS_VALUE + RC_PolicyNvWritten_writtenSet;
    }

// Internal Data Update

    // Set session attributes so that the NV Index needs to be checked
    session->attributes.checkNvWritten = SET;
    session->attributes.nvWrittenState = (in->writtenSet == YES);

    // Update policy hash
    // policyDigestnew = hash(policyDigestold || TPM_CC_PolicyNvWritten
    //                          || writtenSet)
    // Start hash
    CryptHashStart(&hashState, session->authHashAlg);

    // add old digest
    CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);

    // add commandCode
    CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), commandCode);

    // add the byte of writtenState
    CryptDigestUpdateInt(&hashState, sizeof(TPMI_YES_NO), in->writtenSet);

    // complete the digest
    CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);

    return TPM_RC_SUCCESS;
}
#endif // CC_PolicyNvWritten
```

23.21 TPM2_PolicyTemplate

23.21.1 General Description

This command allows a policy to be bound to a specific creation template. This is most useful for an object creation command such as TPM2_Create(), TPM2_CreatePrimary(), or TPM2_CreateLoaded().

The *templateHash* parameter should contain the digest of the template that will be required for the *inPublic* parameter of an Object creation command.

If *policySession→isTemplateHash* is SET and *policySession→cpHash* is not equal to *templateHash,* the TPM shall return TPM_RC_VALUE.

NOTE 1 Revision 01.38 of this specification permitted the TPM to return TPM_RC_CPHASH.

Otherwise, if *policySession→cpHash* is already set, the TPM shall return TPM_RC_CPHASH.

NOTE 2 Revision 01.38 of this specification permitted the TPM to return TPM_RC_VALUE.

If the size of *templateHash* is not the size of *policySession→policyDigest*, the TPM shall return TPM_RC_SIZE. Otherwise, *policySession→cpHash* is set to *templateHash*.

NOTE 3 The digest calculation includes the TPM2B buffer but not the TPM2B size.

If this command completes successfully, the *cpHash* of the authorized command will not be used for validation. Only the digest of the *inPublic* parameter will be used.

NOTE 4 This allows the space normally used to hold *policySession→cpHash* to be used for *policySession→templateHash* instead.

The *policySession→policyDigest* will be updated with

$$policyDigest_{new} \coloneqq \mathbf{H}_{policyAlg}(policyDigest_{old} \mathbin{||} \text{TPM\_CC\_PolicyTemplate} \mathbin{||} templateHash) \qquad (39)$$

23.21.2 Command and Response

Table 160 — TPM2_PolicyTemplate Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit or decrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PolicyTemplate
TPMI_SH_POLICY	policySession	handle for the policy session being extended Auth Index: None
TPM2B_DIGEST	templateHash	the digest to be added to the policy

Table 161 — TPM2_PolicyTemplate Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

23.21.3 Detailed Actions

```
#include "Tpm.h"
#include "PolicyTemplate_fp.h"
#if CC_PolicyTemplate  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_CPHASH	*cpHash* of *policySession* has previously been set to a different value
TPM_RC_SIZE	*templateHash* is not the size of a digest produced by the hash algorithm associated with *policySession*

```
TPM_RC
TPM2_PolicyTemplate(
    PolicyTemplate_In     *in             // IN: input parameter list
    )
{
    SESSION     *session;
    TPM_CC      commandCode = TPM_CC_PolicyTemplate;
    HASH_STATE  hashState;

// Input Validation

    // Get pointer to the session structure
    session = SessionGet(in->policySession);

    // If the template is set, make sure that it is the same as the input value
    if(session->attributes.isTemplateSet)
    {
        if(!MemoryEqual2B(&in->templateHash.b, &session->u1.cpHash.b))
            return TPM_RCS_VALUE + RC_PolicyTemplate_templateHash;
    }
    // error if cpHash contains something that is not a template
    else if(session->u1.templateHash.t.size != 0)
        return TPM_RC_CPHASH;

    // A valid templateHash must have the same size as session hash digest
    if(in->templateHash.t.size != CryptHashGetDigestSize(session->authHashAlg))
        return TPM_RCS_SIZE + RC_PolicyTemplate_templateHash;

// Internal Data Update
    // Update policy hash
    // policyDigestnew = hash(policyDigestold || TPM_CC_PolicyCpHash
    //  || cpHashA.buffer)
    //  Start hash
    CryptHashStart(&hashState, session->authHashAlg);

    //  add old digest
    CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);

    //  add commandCode
    CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), commandCode);

    //  add cpHashA
    CryptDigestUpdate2B(&hashState, &in->templateHash.b);

    //  complete the digest and get the results
    CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);

    // update cpHash in session context
    session->u1.templateHash = in->templateHash;
    session->attributes.isTemplateSet = SET;
```

```
    return TPM_RC_SUCCESS;
}
#endif // CC_PolicyTemplateHash
```

23.22 TPM2_PolicyAuthorizeNV

23.22.1 General Description

This command provides a capability that is the equivalent of a revocable policy. With TPM2_PolicyAuthorize(), the authorization ticket never expires, so the authorization may not be withdrawn. With this command, the approved policy is kept in an NV Index location so that the policy may be changed as needed to render the old policy unusable.

NOTE 1 This command is useful for Objects but of limited value for other policies that are persistently stored in TPM NV, such as the OwnerPolicy.

An authorization session providing authorization to read the NV Index shall be provided.

The authorization to read the NV Index must succeed even if *policySession* is a trial policy session.

If *policySession* is a trial policy session, the TPM will update *policySession→policyDigest* as shown in equation (40) below and return TPM_RC_SUCCESS. It will not perform any further validation. The remainder of this general description would apply only if *policySession* is not a trial policy session.

NOTE 2 If read access is controlled by policy, the policy should include a branch that authorizes a TPM2_PolicyAuthorizeNV().

If TPMA_NV_WRITTEN is not SET in the Index referenced by *nvIndex*, the TPM shall return TPM_RC_NV_UNINITIALIZED. If TPMA_NV_READLOCKED of the NV Index is SET, then the TPM shall return TPM_RC_NV_LOCKED.

The *dataSize* of the NV Index referenced by *nvIndex* is required to be at least large enough to hold a properly formatted TPMT_HA (TPM_RC_INSUFFICIENT).

NOTE 3 A TPMT_HA contains a TPM_ALG_ID followed a digest that is consistent in size with the hash algorithm indicated by the TPM_ALG_ID.

It is an error (TPM_RC_HASH) if the first two octets of the Index are not a TPM_ALG_ID for a hash algorithm implemented on the TPM or if the indicated hash algorithm does not match *policySession→authHash*.

NOTE 4 The TPM_ALG_ID is stored in the first two octets in big endian format.

The TPM will compare *policySession→policyDigest* to the contents of the NV Index, starting at the first octet after the TPM_ALG_ID (the third octet) and return TPM_RC_VALUE if they are not the same.

NOTE 5 If the Index does not contain enough bytes for the compare, then TPM_RC_INSUFFICENT is generated as indicated above.

NOTE 6 The *dataSize* of the Index may be larger than is required for this command. This permits the Index to include metadata.

If the comparison is successful, the TPM will reset *policySession→policyDigest* to a Zero Digest. Then it will update *policySession→policyDigest* with

$$policyDigest_{new} \coloneqq \mathbf{H}_{policyAlg}(policyDigest_{old} \mathbin{||} \text{TPM\_CC\_PolicyAuthorizeNV} \mathbin{||} nvIndex{\to}Name) \quad (40)$$

23.22.2 Command and Response

Table 162 — TPM2_PolicyAuthorizeNV Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PolicyAuthorizeNV
TPMI_RH_NV_AUTH	@authHandle	handle indicating the source of the authorization value Auth Index: 1 Auth Role: USER
TPMI_RH_NV_INDEX	nvIndex	the NV Index of the area to read Auth Index: None
TPMI_SH_POLICY	policySession	handle for the policy session being extended Auth Index: None

Table 163 — TPM2_PolicyAuthorizeNV Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

23.22.3 Detailed Actions

```
#include "Tpm.h"
#if CC_PolicyAuthorizeNV  // Conditional expansion of this file
#include "PolicyAuthorizeNV_fp.h"
#include "Policy_spt_fp.h"
```

Error Returns	Meaning
TPM_RC_HASH	hash algorithm in *keyName* is not supported or is not the same as the hash algorithm of the policy session
TPM_RC_SIZE	*keyName* is not the correct size for its hash algorithm
TPM_RC_VALUE	the current *policyDigest* of *policySession* does not match *approvedPolicy*; or *checkTicket* doesn't match the provided values

```
TPM_RC
TPM2_PolicyAuthorizeNV(
    PolicyAuthorizeNV_In    *in
    )
{
    SESSION                 *session;
    TPM_RC                   result;
    NV_REF                   locator;
    NV_INDEX                *nvIndex = NvGetIndexInfo(in->nvIndex, &locator);
    TPM2B_NAME               name;
    TPMT_HA                  policyInNv;
    BYTE                     nvTemp[sizeof(TPMT_HA)];
    BYTE                    *buffer = nvTemp;
    INT32                    size;

// Input Validation
    // Get pointer to the session structure
    session = SessionGet(in->policySession);

    // Skip checks if this is a trial policy
    if(!session->attributes.isTrialPolicy)
    {
        // Check the authorizations for reading
        // Common read access checks. NvReadAccessChecks() returns
        // TPM_RC_NV_AUTHORIZATION, TPM_RC_NV_LOCKED, or TPM_RC_NV_UNINITIALIZED
        // error may be returned at this point
        result = NvReadAccessChecks(in->authHandle, in->nvIndex,
                                    nvIndex->publicArea.attributes);
        if(result != TPM_RC_SUCCESS)
            return result;

        // Read the contents of the index into a temp buffer
        size = MIN(nvIndex->publicArea.dataSize, sizeof(TPMT_HA));
        NvGetIndexData(nvIndex, locator, 0, (UINT16)size, nvTemp);

        // Unmarshal the contents of the buffer into the internal format of a
        // TPMT_HA so that the hash and digest elements can be accessed from the
        // structure rather than the byte array that is in the Index (written by
        // user of the Index).
        result = TPMT_HA_Unmarshal(&policyInNv, &buffer, &size, FALSE);
        if(result != TPM_RC_SUCCESS)
            return result;

        // Verify that the hash is the same
        if(policyInNv.hashAlg != session->authHashAlg)
            return TPM_RC_HASH;
```

```

        // See if the contents of the digest in the Index matches the value
        // in the policy
        if(!MemoryEqual(&policyInNv.digest, &session->u2.policyDigest.t.buffer,
                        session->u2.policyDigest.t.size))
            return TPM_RC_VALUE;
    }

// Internal Data Update

    // Set policyDigest to zero digest
    PolicyDigestClear(session);

    // Update policyDigest
    PolicyContextUpdate(TPM_CC_PolicyAuthorizeNV, EntityGetName(in->nvIndex, &name),
                        NULL, NULL, 0, session);

    return TPM_RC_SUCCESS;
}
#endif // CC_PolicyAuthorize
```

24 Hierarchy Commands

24.1 TPM2_CreatePrimary

24.1.1 General Description

This command is used to create a Primary Object under one of the Primary Seeds or a Temporary Object under TPM_RH_NULL. The command uses a TPM2B_PUBLIC as a template for the object to be created. The size of the *unique* field shall not be checked for consistency with the other object parameters. The command will create and load a Primary Object. The sensitive area is not returned.

NOTE 1 Since the sensitive data is not returned, the key cannot be reloaded. It can either be made persistent or it can be recreated.

NOTE 2 For interoperability, the *unique* field should not be set to a value that is larger than allowed by object parameters, so that the unmarshaling will not fail.

NOTE 3 An Empty Buffer is a legal *unique* field value.

EXAMPLE 1 A TPM_ALG_RSA object with a *keyBits* of 2048 in the objects parameters should have a *unique* field that is no larger than 256 bytes.

EXAMPLE 2 A TPM_ALG_KEYEDHASH or a TPM_ALG_SYMCIPHER object should have a *unique* field this is no larger than the digest produced by the object's *nameAlg*.

Any type of object and attributes combination that is allowed by TPM2_Create() may be created by this command. The constraints on templates and parameters are the same as TPM2_Create() except that a Primary Storage Key and a Temporary Storage Key are not constrained to use the algorithms of their parents.

For setting of the attributes of the created object, *fixedParent*, *fixedTPM*, decrypt, and restricted are implied to be SET in the parent (a Permanent Handle). The remaining attributes are implied to be CLEAR.

The TPM will derive the object from the Primary Seed indicated in *primaryHandle* using an approved KDF. All of the bits of the template are used in the creation of the Primary Key. Methods for creating a Primary Object from a Primary Seed are described in TPM 2.0 Part 1 and implemented in TPM 2.0 Part 4.

If this command is called multiple times with the same *inPublic* parameter, *inSensitive.data*, and Primary Seed, the TPM shall produce the same Primary Object.

NOTE 4 If the Primary Seed is changed, the Primary Objects generated with the new seed shall be statistically unique even if the parameters of the call are the same.

This command requires authorization. Authorization for a Primary Object attached to the Platform Primary Seed (PPS) shall be provided by *platformAuth* or *platformPolicy*. Authorization for a Primary Object attached to the Storage Primary Seed (SPS) shall be provided by *ownerAuth* or *ownerPolicy*. Authorization for a Primary Key attached to the Endorsement Primary Seed (EPS) shall be provided by *endorsementAuth* or *endorsementPolicy*.

24.1.2 Command and Response

Table 164 — TPM2_CreatePrimary Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_CreatePrimary
TPMI_RH_HIERARCHY+	@primaryHandle	TPM_RH_ENDORSEMENT, TPM_RH_OWNER, TPM_RH_PLATFORM+{PP}, or TPM_RH_NULL Auth Index: 1 Auth Role: USER
TPM2B_SENSITIVE_CREATE	inSensitive	the sensitive data, see TPM 2.0 Part 1 Sensitive Values
TPM2B_PUBLIC	inPublic	the public template
TPM2B_DATA	outsideInfo	data that will be included in the creation data for this object to provide permanent, verifiable linkage between this object and some object owner data
TPML_PCR_SELECTION	creationPCR	PCR that will be used in creation data

Table 165 — TPM2_CreatePrimary Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM_HANDLE	objectHandle	handle of type TPM_HT_TRANSIENT for created Primary Object
TPM2B_PUBLIC	outPublic	the public portion of the created object
TPM2B_CREATION_DATA	creationData	contains a TPMT_CREATION_DATA
TPM2B_DIGEST	creationHash	digest of *creationData* using *nameAlg* of *outPublic*
TPMT_TK_CREATION	creationTicket	ticket used by TPM2_CertifyCreation() to validate that the creation data was produced by the TPM
TPM2B_NAME	name	the name of the created object

24.1.3 Detailed Actions

```
#include "Tpm.h"
#include "CreatePrimary_fp.h"
#if CC_CreatePrimary  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_ATTRIBUTES	*sensitiveDataOrigin* is CLEAR when sensitive.data is an Empty Buffer *fixedTPM*, *fixedParent*, or *encryptedDuplication* attributes are inconsistent between themselves or with those of the parent object; inconsistent *restricted*, *decrypt* and *sign* attributes attempt to inject sensitive data for an asymmetric key;
TPM_RC_KDF	incorrect KDF specified for decrypting keyed hash object
TPM_RC_KEY	a provided symmetric key value is not allowed
TPM_RC_OBJECT_MEMORY	there is no free slot for the object
TPM_RC_SCHEME	inconsistent attributes *decrypt*, *sign*, *restricted* and key's scheme ID; or hash algorithm is inconsistent with the scheme ID for keyed hash object
TPM_RC_SIZE	size of public authorization policy or sensitive authorization value does not match digest size of the name algorithm; or sensitive data size for the keyed hash object is larger than is allowed for the scheme
TPM_RC_SYMMETRIC	a storage key with no symmetric algorithm specified; or non-storage key with symmetric algorithm different from TPM_ALG_NULL
TPM_RC_TYPE	unknown object type

```
TPM_RC
TPM2_CreatePrimary(
    CreatePrimary_In     *in,             // IN: input parameter list
    CreatePrimary_Out    *out             // OUT: output parameter list
    )
{
    TPM_RC               result = TPM_RC_SUCCESS;
    TPMT_PUBLIC         *publicArea;
    DRBG_STATE           rand;
    OBJECT              *newObject;
    TPM2B_NAME           name;

// Input Validation
    // Will need a place to put the result
    newObject = FindEmptyObjectSlot(&out->objectHandle);
    if(newObject == NULL)
        return TPM_RC_OBJECT_MEMORY;
    // Get the address of the public area in the new object
    // (this is just to save typing)
    publicArea = &newObject->publicArea;

    *publicArea = in->inPublic.publicArea;

    // Check attributes in input public area. CreateChecks() checks the things that
    // are unique to creation and then validates the attributes and values that are
    // common to create and load.
    result = CreateChecks(NULL, publicArea,
                          in->inSensitive.sensitive.data.t.size);
    if(result != TPM_RC_SUCCESS)
        return RcSafeAddToResult(result, RC_CreatePrimary_inPublic);
    // Validate the sensitive area values
```



```
    if(!AdjustAuthSize(&in->inSensitive.sensitive.userAuth,
                       publicArea->nameAlg))
        return TPM_RCS_SIZE + RC_CreatePrimary_inSensitive;
// Command output
    // Compute the name using out->name as a scratch area (this is not the value
    // that ultimately will be returned, then instantiate the state that will be
    // used as a random number generator during the object creation.
    // The caller does not know the seed values so the actual name does not have
    // to be over the input, it can be over the unmarshaled structure.
    result = DRBG_InstantiateSeeded(&rand,
                           &HierarchyGetPrimarySeed(in->primaryHandle)->b,
                           PRIMARY_OBJECT_CREATION,
                           (TPM2B *)PublicMarshalAndComputeName(publicArea, &name),
                           &in->inSensitive.sensitive.data.b);
    if(result == TPM_RC_SUCCESS)
    {
        newObject->attributes.primary = SET;
        if(in->primaryHandle == TPM_RH_ENDORSEMENT)
            newObject->attributes.epsHierarchy = SET;

        // Create the primary object.
        result = CryptCreateObject(newObject, &in->inSensitive.sensitive,
            (RAND_STATE *)&rand);
    }
    if(result != TPM_RC_SUCCESS)
        return result;

    // Set the publicArea and name from the computed values
    out->outPublic.publicArea = newObject->publicArea;
    out->name = newObject->name;

    // Fill in creation data
    FillInCreationData(in->primaryHandle, publicArea->nameAlg,
                       &in->creationPCR, &in->outsideInfo, &out->creationData,
                       &out->creationHash);

    // Compute creation ticket
    TicketComputeCreation(EntityGetHierarchy(in->primaryHandle), &out->name,
                          &out->creationHash, &out->creationTicket);

    // Set the remaining attributes for a loaded object
    ObjectSetLoadedAttributes(newObject, in->primaryHandle);
    return result;
}
#endif // CC_CreatePrimary
```

24.2 TPM2_HierarchyControl

24.2.1 General Description

This command enables and disables use of a hierarchy and its associated NV storage. The command allows *phEnable*, *phEnableNV*, *shEnable*, and *ehEnable* to be changed when the proper authorization is provided.

This command may be used to CLEAR *phEnable* and *phEnableNV* if *platformAuth/platformPolicy* is provided. *phEnable* may not be SET using this command.

This command may be used to CLEAR *shEnable* if either *platformAuth/platformPolicy* or *ownerAuth/ownerPolicy* is provided. *shEnable* may be SET if *platformAuth*/*platformPolicy* is provided.

This command may be used to CLEAR *ehEnable* if either *platformAuth*/*platformPolicy* or *endorsementAuth*/*endorsementPolicy* is provided. *ehEnable* may be SET if *platformAuth*/*platformPolicy* is provided.

When this command is used to CLEAR *phEnable*, *shEnable*, or *ehEnable*, the TPM will disable use of any persistent entity associated with the disabled hierarchy and will flush any transient objects associated with the disabled hierarchy.

When this command is used to CLEAR *shEnable,* the TPM will disable access to any NV index that has TPMA_NV_PLATFORMCREATE CLEAR (indicating that the NV Index was defined using Owner Authorization). As long as *shEnable* is CLEAR, the TPM will return an error in response to any command that attempts to operate upon an NV index that has TPMA_NV_PLATFORMCREATE CLEAR.

When this command is used to CLEAR *phEnableNV*, the TPM will disable access to any NV index that has TPMA_NV_PLATFORMCREATE SET (indicating that the NV Index was defined using Platform Authorization). As long as *phEnableNV* is CLEAR, the TPM will return an error in response to any command that attempts to operate upon an NV index that has TPMA_NV_PLATFORMCREATE SET.

24.2.2 Command and Response

Table 166 — TPM2_HierarchyControl Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_HierarchyControl {NV E}
TPMI_RH_HIERARCHY	@authHandle	TPM_RH_ENDORSEMENT, TPM_RH_OWNER or TPM_RH_PLATFORM+{PP} Auth Index: 1 Auth Role: USER
TPMI_RH_ENABLES	enable	the enable being modified TPM_RH_ENDORSEMENT, TPM_RH_OWNER, TPM_RH_PLATFORM, or TPM_RH_PLATFORM_NV
TPMI_YES_NO	state	YES if the enable should be SET, NO if the enable should be CLEAR

Table 167 — TPM2_HierarchyControl Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

24.2.3 Detailed Actions

```
#include "Tpm.h"
#include "HierarchyControl_fp.h"
#if CC_HierarchyControl  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_AUTH_TYPE	*authHandle* is not applicable to *hierarchy* in its current state

```
TPM_RC
TPM2_HierarchyControl(
    HierarchyControl_In     *in              // IN: input parameter list
    )
{
    BOOL        select = (in->state == YES);
    BOOL        *selected = NULL;

// Input Validation
    switch(in->enable)
    {
        // Platform hierarchy has to be disabled by PlatformAuth
        // If the platform hierarchy has already been disabled, only a reboot
        // can enable it again
        case TPM_RH_PLATFORM:
        case TPM_RH_PLATFORM_NV:
            if(in->authHandle != TPM_RH_PLATFORM)
                return TPM_RC_AUTH_TYPE;
            break;

        // ShEnable may be disabled if PlatformAuth/PlatformPolicy or
        // OwnerAuth/OwnerPolicy is provided.  If ShEnable is disabled, then it
        // may only be enabled if PlatformAuth/PlatformPolicy is provided.
        case TPM_RH_OWNER:
            if(in->authHandle != TPM_RH_PLATFORM
               && in->authHandle != TPM_RH_OWNER)
                return TPM_RC_AUTH_TYPE;
            if(gc.shEnable == FALSE && in->state == YES
               && in->authHandle != TPM_RH_PLATFORM)
                return TPM_RC_AUTH_TYPE;
            break;

        // EhEnable may be disabled if either PlatformAuth/PlatformPolicy or
        // EndosementAuth/EndorsementPolicy is provided.  If EhEnable is disabled,
        // then it may only be enabled if PlatformAuth/PlatformPolicy is
        // provided.
        case TPM_RH_ENDORSEMENT:
            if(in->authHandle != TPM_RH_PLATFORM
               && in->authHandle != TPM_RH_ENDORSEMENT)
                return TPM_RC_AUTH_TYPE;
            if(gc.ehEnable == FALSE && in->state == YES
               && in->authHandle != TPM_RH_PLATFORM)
                return TPM_RC_AUTH_TYPE;
            break;
        default:
            FAIL(FATAL_ERROR_INTERNAL);
            break;
    }

// Internal Data Update

    // Enable or disable the selected hierarchy
    // Note: the authorization processing for this command may keep these
```

```
    // command actions from being executed. For example, if phEnable is
    // CLEAR, then platformAuth cannot be used for authorization. This
    // means that would not be possible to use platformAuth to change the
    // state of phEnable from CLEAR to SET.
    // If it is decided that platformPolicy can still be used when phEnable
    // is CLEAR, then this code could SET phEnable when proper platform
    // policy is provided.
    switch(in->enable)
    {
        case TPM_RH_OWNER:
            selected = &gc.shEnable;
            break;
        case TPM_RH_ENDORSEMENT:
            selected = &gc.ehEnable;
            break;
        case TPM_RH_PLATFORM:
            selected = &g_phEnable;
            break;
        case TPM_RH_PLATFORM_NV:
            selected = &gc.phEnableNV;
            break;
        default:
            FAIL(FATAL_ERROR_INTERNAL);
            break;
    }
    if(selected != NULL && *selected != select)
    {
        // Before changing the internal state, make sure that NV is available.
        // Only need to update NV if changing the orderly state
        RETURN_IF_ORDERLY;

        // state is changing and NV is available so modify
        *selected = select;
        // If a hierarchy was just disabled, flush it
        if(select == CLEAR && in->enable != TPM_RH_PLATFORM_NV)
        // Flush hierarchy
            ObjectFlushHierarchy(in->enable);

        // orderly state should be cleared because of the update to state clear data
        // This gets processed in ExecuteCommand() on the way out.
        g_clearOrderly = TRUE;
    }
    return TPM_RC_SUCCESS;
}
#endif // CC_HierarchyControl
```


24.3 TPM2_SetPrimaryPolicy

24.3.1 General Description

This command allows setting of the authorization policy for the lockout (*lockoutPolicy*), the platform hierarchy (*platformPolicy*), the storage hierarchy (*ownerPolicy*), and the endorsement hierarchy (*endorsementPolicy*). On TPMs implementing Authenticated Countdown Timers (ACT), this command may also be used to set the authorization policy for an ACT.

The command requires an authorization session. The session shall use the current *authValue* or satisfy the current *authPolicy* for the referenced hierarchy, or the ACT.

The policy that is changed is the policy associated with *authHandle*.

If the enable associated with *authHandle* is not SET, then the associated authorization values (*authValue* or *authPolicy*) may not be used, and the TPM returns TPM_RC_HIERARCHY.

When *hashAlg* is not TPM_ALG_NULL, if the size of *authPolicy* is not consistent with the hash algorithm, the TPM returns TPM_RC_SIZE.

24.3.2 Command and Response

Table 168 — TPM2_SetPrimaryPolicy Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_SetPrimaryPolicy {NV}
TPMI_RH_HIERARCHY_POLICY	@authHandle	TPM_RH_LOCKOUT, TPM_RH_ENDORSEMENT, TPM_RH_OWNER, TPMI_RH_ACT or TPM_RH_PLATFORM+{PP} Auth Index: 1 Auth Role: USER
TPM2B_DIGEST	authPolicy	an authorization policy digest; may be the Empty Buffer If *hashAlg* is TPM_ALG_NULL, then this shall be an Empty Buffer.
TPMI_ALG_HASH+	hashAlg	the hash algorithm to use for the policy If the *authPolicy* is an Empty Buffer, then this field shall be TPM_ALG_NULL.

Table 169 — TPM2_SetPrimaryPolicy Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

24.3.3 Detailed Actions

```
#include "Tpm.h"
#include "SetPrimaryPolicy_fp.h"
#if CC_SetPrimaryPolicy  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_SIZE	size of input *authPolicy* is not consistent with input hash algorithm

```
TPM_RC
TPM2_SetPrimaryPolicy(
    SetPrimaryPolicy_In     *in             // IN: input parameter list
    )
{
// Input Validation

    // Check the authPolicy consistent with hash algorithm. If the policy size is
    // zero, then the algorithm is required to be TPM_ALG_NULL
    if(in->authPolicy.t.size != CryptHashGetDigestSize(in->hashAlg))
        return TPM_RCS_SIZE + RC_SetPrimaryPolicy_authPolicy;

    // The command need NV update for OWNER and ENDORSEMENT hierarchy, and
    // might need orderlyState update for PLATFROM hierarchy.
    // Check if NV is available.  A TPM_RC_NV_UNAVAILABLE or TPM_RC_NV_RATE
    // error may be returned at this point
    RETURN_IF_NV_IS_NOT_AVAILABLE;

// Internal Data Update

    // Set hierarchy policy
    switch(in->authHandle)
    {
        case TPM_RH_OWNER:
            gp.ownerAlg = in->hashAlg;
            gp.ownerPolicy = in->authPolicy;
            NV_SYNC_PERSISTENT(ownerAlg);
            NV_SYNC_PERSISTENT(ownerPolicy);
            break;
        case TPM_RH_ENDORSEMENT:
            gp.endorsementAlg = in->hashAlg;
            gp.endorsementPolicy = in->authPolicy;
            NV_SYNC_PERSISTENT(endorsementAlg);
            NV_SYNC_PERSISTENT(endorsementPolicy);
            break;
        case TPM_RH_PLATFORM:
            gc.platformAlg = in->hashAlg;
            gc.platformPolicy = in->authPolicy;
            // need to update orderly state
            g_clearOrderly = TRUE;
            break;
        case TPM_RH_LOCKOUT:
            gp.lockoutAlg = in->hashAlg;
            gp.lockoutPolicy = in->authPolicy;
            NV_SYNC_PERSISTENT(lockoutAlg);
            NV_SYNC_PERSISTENT(lockoutPolicy);
            break;

#define SET_ACT_POLICY(N)                                                   \
        case TPM_RH_ACT_##N:                                                \
            go.ACT_##N.hashAlg = in->hashAlg;                               \
            go.ACT_##N.authPolicy = in->authPolicy;                         \
            g_clearOrderly = TRUE;                                          \
```



```
            break;

            FOR_EACH_ACT(SET_ACT_POLICY)

        default:
            FAIL(FATAL_ERROR_INTERNAL);
            break;
    }

    return TPM_RC_SUCCESS;
}
#endif // CC_SetPrimaryPolicy
```


24.4 TPM2_ChangePPS

24.4.1 General Description

This replaces the current platform primary seed (PPS) with a value from the RNG and sets *platformPolicy* to the default initialization value (the Empty Buffer).

NOTE 1 A policy that is the Empty Buffer can match no policy.

NOTE 2 Platform Authorization is not changed.

All resident transient and persistent objects in the Platform hierarchy are flushed.

Saved contexts in the Platform hierarchy that were created under the old PPS will no longer be able to be loaded.

The policy hash algorithm for PCR is reset to TPM_ALG_NULL.

This command does not clear any NV Index values.

NOTE 3 Index values belonging to the Platform are preserved because the indexes may have configuration information that will be the same after the PPS changes. The Platform may remove the indexes that are no longer needed using TPM2_NV_UndefineSpace().

This command requires Platform Authorization.

24.4.2 Command and Response

Table 170 — TPM2_ChangePPS Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_ChangePPS {NV E}
TPMI_RH_PLATFORM	@authHandle	TPM_RH_PLATFORM+{PP} Auth Index: 1 Auth Role: USER

Table 171 — TPM2_ChangePPS Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

24.4.3 Detailed Actions

```
#include "Tpm.h"
#include "ChangePPS_fp.h"
#if CC_ChangePPS  // Conditional expansion of this file
TPM_RC
TPM2_ChangePPS(
    ChangePPS_In    *in             // IN: input parameter list
    )
{
    UINT32          i;

    // Check if NV is available.  A TPM_RC_NV_UNAVAILABLE or TPM_RC_NV_RATE
    // error may be returned at this point
    RETURN_IF_NV_IS_NOT_AVAILABLE;

    // Input parameter is not reference in command action
    NOT_REFERENCED(in);

// Internal Data Update

    // Reset platform hierarchy seed from RNG
    CryptRandomGenerate(sizeof(gp.PPSeed.t.buffer), gp.PPSeed.t.buffer);

    // Create a new phProof value from RNG to prevent the saved platform
    // hierarchy contexts being loaded
    CryptRandomGenerate(sizeof(gp.phProof.t.buffer), gp.phProof.t.buffer);

    // Set platform authPolicy to null
    gc.platformAlg = TPM_ALG_NULL;
    gc.platformPolicy.t.size = 0;

    // Flush loaded object in platform hierarchy
    ObjectFlushHierarchy(TPM_RH_PLATFORM);

    // Flush platform evict object and index in NV
    NvFlushHierarchy(TPM_RH_PLATFORM);

    // Save hierarchy changes to NV
    NV_SYNC_PERSISTENT(PPSeed);
    NV_SYNC_PERSISTENT(phProof);

    // Re-initialize PCR policies
#if defined NUM_POLICY_PCR_GROUP && NUM_POLICY_PCR_GROUP > 0
    for(i = 0; i < NUM_POLICY_PCR_GROUP; i++)
    {
        gp.pcrPolicies.hashAlg[i] = TPM_ALG_NULL;
        gp.pcrPolicies.policy[i].t.size = 0;
    }
    NV_SYNC_PERSISTENT(pcrPolicies);
#endif

    // orderly state should be cleared because of the update to state clear data
    g_clearOrderly = TRUE;

    return TPM_RC_SUCCESS;
}
#endif // CC_ChangePPS
```


24.5 TPM2_ChangeEPS

24.5.1 General Description

This replaces the current endorsement primary seed (EPS) with a value from the RNG and sets the Endorsement hierarchy controls to their default initialization values: *ehEnable* is SET, *endorsementAuth* and *endorsementPolicy* are both set to the Empty Buffer. It will flush any resident objects (transient or persistent) in the Endorsement hierarchy and not allow objects in the hierarchy associated with the previous EPS to be loaded.

NOTE In the reference implementation, *ehProof* is a non-volatile value from the RNG. It is allowed that the *ehProof* be generated by a KDF using both the EPS and SPS as inputs. If generated with a KDF, the ehProof can be generated on an as-needed basis or made a non-volatile value.

This command requires Platform Authorization.

24.5.2 Command and Response

Table 172 — TPM2_ChangeEPS Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_ChangeEPS {NV E}
TPMI_RH_PLATFORM	@authHandle	TPM_RH_PLATFORM+{PP} Auth Handle: 1 Auth Role: USER

Table 173 — TPM2_ChangeEPS Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

24.5.3 Detailed Actions

```
#include "Tpm.h"
#include "ChangeEPS_fp.h"
#if CC_ChangeEPS  // Conditional expansion of this file
TPM_RC
TPM2_ChangeEPS(
    ChangeEPS_In    *in             // IN: input parameter list
    )
{
    // The command needs NV update.  Check if NV is available.
    // A TPM_RC_NV_UNAVAILABLE or TPM_RC_NV_RATE error may be returned at
    // this point
    RETURN_IF_NV_IS_NOT_AVAILABLE;

    // Input parameter is not reference in command action
    NOT_REFERENCED(in);

// Internal Data Update

    // Reset endorsement hierarchy seed from RNG
    CryptRandomGenerate(sizeof(gp.EPSeed.t.buffer), gp.EPSeed.t.buffer);

    // Create new ehProof value from RNG
    CryptRandomGenerate(sizeof(gp.ehProof.t.buffer), gp.ehProof.t.buffer);

    // Enable endorsement hierarchy
    gc.ehEnable = TRUE;

    // set authValue buffer to zeros
    MemorySet(gp.endorsementAuth.t.buffer, 0, gp.endorsementAuth.t.size);
    // Set endorsement authValue to null
    gp.endorsementAuth.t.size = 0;

    // Set endorsement authPolicy to null
    gp.endorsementAlg = TPM_ALG_NULL;
    gp.endorsementPolicy.t.size = 0;

    // Flush loaded object in endorsement hierarchy
    ObjectFlushHierarchy(TPM_RH_ENDORSEMENT);

    // Flush evict object of endorsement hierarchy stored in NV
    NvFlushHierarchy(TPM_RH_ENDORSEMENT);

    // Save hierarchy changes to NV
    NV_SYNC_PERSISTENT(EPSeed);
    NV_SYNC_PERSISTENT(ehProof);
    NV_SYNC_PERSISTENT(endorsementAuth);
    NV_SYNC_PERSISTENT(endorsementAlg);
    NV_SYNC_PERSISTENT(endorsementPolicy);

    // orderly state should be cleared because of the update to state clear data
    g_clearOrderly = TRUE;

    return TPM_RC_SUCCESS;
}
#endif // CC_ChangeEPS
```

24.6 TPM2_Clear

24.6.1 General Description

This command removes all TPM context associated with a specific Owner.

The clear operation will:

- flush resident objects (persistent and volatile) in the Storage and Endorsement hierarchies;
- delete any NV Index with TPMA_NV_PLATFORMCREATE == CLEAR;
- change the storage primary seed (SPS) to a new value from the TPM's random number generator (RNG),
- change *shProof* and *ehProof*,

  NOTE 1 The proof values may be set from the RNG or derived from the associated new Primary Seed. If derived from the Primary Seeds, the derivation of *ehProof* shall use both the SPS and EPS. The computation shall use the SPS as an HMAC key and the derived value may then be a parameter in a second HMAC in which the EPS is the HMAC key. The reference design uses values from the RNG.

- SET *shEnable* and *ehEnable*;
- set *ownerAuth*, *endorsementAuth*, and *lockoutAuth* to the Empty Buffer;
- set *ownerPolicy, endorsementPolicy,* and *lockoutPolicy* to the Empty Buffer;
- set *Clock* to zero;
- set *resetCount* to zero;
- set *restartCount* to zero; and
- set *Safe* to YES.
- increment *pcrUpdateCounter*

  NOTE 2 This permits an application to create a policy session that is invalidated on TPM2_Clear. The policy needs, ideally as the first term, TPM2_PolicyPCR(). The session is invalidated even if the PCR selection is empty.

This command requires Platform Authorization or Lockout Authorization. If TPM2_ClearControl() has disabled this command, the TPM shall return TPM_RC_DISABLED.

If this command is authorized using *lockoutAuth*, the HMAC in the response shall use the new *lockoutAuth* value (that is, the Empty Buffer) when computing the response HMAC.

24.6.2 Command and Response

Table 174 — TPM2_Clear Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_Clear {NV E}
TPMI_RH_CLEAR	@authHandle	TPM_RH_LOCKOUT or TPM_RH_PLATFORM+{PP} Auth Handle: 1 Auth Role: USER

Table 175 — TPM2_Clear Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

24.6.3 Detailed Actions

```
#include "Tpm.h"
#include "Clear_fp.h"
#if CC_Clear  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_DISABLED	Clear command has been disabled

```
TPM_RC
TPM2_Clear(
    Clear_In        *in             // IN: input parameter list
    )
{
    // Input parameter is not reference in command action
    NOT_REFERENCED(in);

    // The command needs NV update.  Check if NV is available.
    // A TPM_RC_NV_UNAVAILABLE or TPM_RC_NV_RATE error may be returned at
    // this point
    RETURN_IF_NV_IS_NOT_AVAILABLE;

// Input Validation

    // If Clear command is disabled, return an error
    if(gp.disableClear)
        return TPM_RC_DISABLED;

// Internal Data Update

    // Reset storage hierarchy seed from RNG
    CryptRandomGenerate(sizeof(gp.SPSeed.t.buffer), gp.SPSeed.t.buffer);

    // Create new shProof and ehProof value from RNG
    CryptRandomGenerate(sizeof(gp.shProof.t.buffer), gp.shProof.t.buffer);
    CryptRandomGenerate(sizeof(gp.ehProof.t.buffer), gp.ehProof.t.buffer);

    // Enable storage and endorsement hierarchy
    gc.shEnable = gc.ehEnable = TRUE;

    // set the authValue buffers to zero
    MemorySet(&gp.ownerAuth, 0, sizeof(gp.ownerAuth));
    MemorySet(&gp.endorsementAuth, 0, sizeof(gp.endorsementAuth));
    MemorySet(&gp.lockoutAuth, 0, sizeof(gp.lockoutAuth));

    // Set storage, endorsement, and lockout authPolicy to null
    gp.ownerAlg = gp.endorsementAlg = gp.lockoutAlg = TPM_ALG_NULL;
    MemorySet(&gp.ownerPolicy, 0, sizeof(gp.ownerPolicy));
    MemorySet(&gp.endorsementPolicy, 0, sizeof(gp.endorsementPolicy));
    MemorySet(&gp.lockoutPolicy, 0, sizeof(gp.lockoutPolicy));

    // Flush loaded object in storage and endorsement hierarchy
    ObjectFlushHierarchy(TPM_RH_OWNER);
    ObjectFlushHierarchy(TPM_RH_ENDORSEMENT);

    // Flush owner and endorsement object and owner index in NV
    NvFlushHierarchy(TPM_RH_OWNER);
    NvFlushHierarchy(TPM_RH_ENDORSEMENT);

    // Initialize dictionary attack parameters
    DAPreInstall_Init();

```

```
    // Reset clock
    go.clock = 0;
    go.clockSafe = YES;
    NvWrite(NV_ORDERLY_DATA, sizeof(ORDERLY_DATA), &go);

    // Reset counters
    gp.resetCount = gr.restartCount = gr.clearCount = 0;
    gp.auditCounter = 0;

    // Save persistent data changes to NV
    // Note: since there are so many changes to the persistent data structure, the
    // entire PERSISTENT_DATA structure is written as a unit
    NvWrite(NV_PERSISTENT_DATA, sizeof(PERSISTENT_DATA), &gp);

    // Reset the PCR authValues (this does not change the PCRs)
    PCR_ClearAuth();

    // Bump the PCR counter
    PCRChanged(0);

    // orderly state should be cleared because of the update to state clear data
    g_clearOrderly = TRUE;

    return TPM_RC_SUCCESS;
}
#endif // CC_Clear
```


24.7 TPM2_ClearControl

24.7.1 General Description

TPM2_ClearControl() disables and enables the execution of TPM2_Clear().

The TPM will SET the TPM's TPMA_PERMANENT.*disableClear* attribute if *disable* is YES and will CLEAR the attribute if *disable* is NO. When the attribute is SET, TPM2_Clear() may not be executed.

NOTE This is to simplify the logic of TPM2_Clear(). TPM2_ClearControl() can be called using Platform Authorization to CLEAR the *disableClear* attribute and then execute TPM2_Clear().

Lockout Authorization may be used to SET *disableClear* but not to CLEAR it.

Platform Authorization may be used to SET or CLEAR *disableClear*.

24.7.2 Command and Response

Table 176 — TPM2_ClearControl Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_ClearControl {NV}
TPMI_RH_CLEAR	@auth	TPM_RH_LOCKOUT or TPM_RH_PLATFORM+{PP} Auth Handle: 1 Auth Role: USER
TPMI_YES_NO	disable	YES if the *disableOwnerClear* flag is to be SET, NO if the flag is to be CLEAR.

Table 177 — TPM2_ClearControl Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

24.7.3 Detailed Actions

```
#include "Tpm.h"
#include "ClearControl_fp.h"
#if CC_ClearControl  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_AUTH_FAIL	authorization is not properly given

```
TPM_RC
TPM2_ClearControl(
    ClearControl_In     *in             // IN: input parameter list
    )
{
    // The command needs NV update.
    RETURN_IF_NV_IS_NOT_AVAILABLE;

// Input Validation

    // LockoutAuth may be used to set disableLockoutClear to TRUE but not to FALSE
    if(in->auth == TPM_RH_LOCKOUT && in->disable == NO)
        return TPM_RC_AUTH_FAIL;

// Internal Data Update

    if(in->disable == YES)
        gp.disableClear = TRUE;
    else
        gp.disableClear = FALSE;

    // Record the change to NV
    NV_SYNC_PERSISTENT(disableClear);

    return TPM_RC_SUCCESS;
}
#endif // CC_ClearControl
```

24.8 TPM2_HierarchyChangeAuth

24.8.1 General Description

This command allows the authorization secret for a hierarchy or lockout to be changed using the current authorization value as the command authorization.

If *authHandle* is TPM_RH_PLATFORM, then *platformAuth* is changed. If *authHandle* is TPM_RH_OWNER, then o*wnerAuth* is changed. If *authHandle* is TPM_RH_ENDORSEMENT, then e*ndorsementAuth* is changed. If *authHandle* is TPM_RH_LOCKOUT, then *lockoutAuth* is changed. The HMAC in the response shall use the new authorization value when computing the response HMAC.

If *authHandle* is TPM_RH_PLATFORM, then Physical Presence may need to be asserted for this command to succeed (see 26.2, *TPM2_PP_Commands*).

The authorization value may be no larger than the digest produced by the hash algorithm used for context integrity.

EXAMPLE If SHA384 is used in the computation of the integrity values for saved contexts, then the largest authorization value is 48 octets.

24.8.2 Command and Response

Table 178 — TPM2_HierarchyChangeAuth Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_HierarchyChangeAuth {NV}
TPMI_RH_HIERARCHY_AUTH	@authHandle	TPM_RH_LOCKOUT, TPM_RH_ENDORSEMENT, TPM_RH_OWNER or TPM_RH_PLATFORM+{PP} Auth Index: 1 Auth Role: USER
TPM2B_AUTH	newAuth	new authorization value

Table 179 — TPM2_HierarchyChangeAuth Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

24.8.3 Detailed Actions

```
#include "Tpm.h"
#include "HierarchyChangeAuth_fp.h"
#if CC_HierarchyChangeAuth  // Conditional expansion of this file
#include "Object_spt_fp.h"
```

Error Returns	Meaning
TPM_RC_SIZE	*newAuth* size is greater than that of integrity hash digest

```
TPM_RC
TPM2_HierarchyChangeAuth(
    HierarchyChangeAuth_In   *in              // IN: input parameter list
    )
{
    // The command needs NV update.
    RETURN_IF_NV_IS_NOT_AVAILABLE;

    // Make sure that the authorization value is a reasonable size (not larger than
    // the size of the digest produced by the integrity hash. The integrity
    // hash is assumed to produce the longest digest of any hash implemented
    // on the TPM. This will also remove trailing zeros from the authValue.
    if(MemoryRemoveTrailingZeros(&in->newAuth) > CONTEXT_INTEGRITY_HASH_SIZE)
        return TPM_RCS_SIZE + RC_HierarchyChangeAuth_newAuth;

    // Set hierarchy authValue
    switch(in->authHandle)
    {
        case TPM_RH_OWNER:
            gp.ownerAuth = in->newAuth;
            NV_SYNC_PERSISTENT(ownerAuth);
            break;
        case TPM_RH_ENDORSEMENT:
            gp.endorsementAuth = in->newAuth;
            NV_SYNC_PERSISTENT(endorsementAuth);
            break;
        case TPM_RH_PLATFORM:
            gc.platformAuth = in->newAuth;
            // orderly state should be cleared
            g_clearOrderly = TRUE;
            break;
        case TPM_RH_LOCKOUT:
            gp.lockoutAuth = in->newAuth;
            NV_SYNC_PERSISTENT(lockoutAuth);
            break;
        default:
            FAIL(FATAL_ERROR_INTERNAL);
            break;
    }

    return TPM_RC_SUCCESS;
}
#endif // CC_HierarchyChangeAuth
```


25 Dictionary Attack Functions

25.1 Introduction

A TPM is required to have support for logic that will help prevent a dictionary attack on an authorization value. The protection is provided by a counter that increments when a password authorization or an HMAC authorization fails. When the counter reaches a predefined value, the TPM will not accept, for some time interval, further requests that require authorization and the TPM is in Lockout mode. While the TPM is in Lockout mode, the TPM will return TPM_RC_LOCKOUT if the command requires use of an object's or Index's *authValue* unless the authorization applies to an entry in the Platform hierarchy.

NOTE 1 Authorizations for objects and NV Index values in the Platform hierarchy are never locked out. However, a command that requires multiple authorizations will not be accepted when the TPM is in Lockout mode unless all of the authorizations reference objects and indexes in the Platform hierarchy.

If the TPM is continuously powered for the duration of *newRecoveryTime* and no authorization failures occur, the authorization failure counter will be decremented by one. This property is called "self-healing." Self-healing shall not cause the count of failed attempts to decrement below zero.

The count of failed attempts, the lockout interval, and self-healing interval are settable using TPM2_DictionaryAttackParameters(). The lockout parameters and the current value of the lockout counter can be read with TPM2_GetCapability().

Dictionary attack protection does not apply to an entity associated with a permanent handle (handle type == TPM_HT_PERMANENT) other than TPM_RH_LOCKOUT

25.2 TPM2_DictionaryAttackLockReset

25.2.1 General Description

This command cancels the effect of a TPM lockout due to a number of successive authorization failures. If this command is properly authorized, the lockout counter is set to zero.

Only one *lockoutAuth* authorization failure is allowed for this command during a *lockoutRecovery* interval (set using TPM2_DictionaryAttackParameters().

25.2.2 Command and Response

Table 180 — TPM2_DictionaryAttackLockReset Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_DictionaryAttackLockReset {NV}
TPMI_RH_LOCKOUT	@lockHandle	TPM_RH_LOCKOUT Auth Index: 1 Auth Role: USER

Table 181 — TPM2_DictionaryAttackLockReset Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

25.2.3 Detailed Actions

```
#include "Tpm.h"
#include "DictionaryAttackLockReset_fp.h"
#if CC_DictionaryAttackLockReset  // Conditional expansion of this file
TPM_RC
TPM2_DictionaryAttackLockReset(
    DictionaryAttackLockReset_In    *in             // IN: input parameter list
    )
{
    // Input parameter is not reference in command action
    NOT_REFERENCED(in);

    // The command needs NV update.
    RETURN_IF_NV_IS_NOT_AVAILABLE;

// Internal Data Update

    // Set failed tries to 0
    gp.failedTries = 0;

    // Record the changes to NV
    NV_SYNC_PERSISTENT(failedTries);

    return TPM_RC_SUCCESS;
}
#endif // CC_DictionaryAttackLockReset
```


25.3 TPM2_DictionaryAttackParameters

25.3.1 General Description

This command changes the lockout parameters.

The command requires Lockout Authorization.

The timeout parameters (*newRecoveryTime* and *lockoutRecovery*) indicate values that are measured with respect to the *Time* and not *Clock*.

NOTE Use of *Time* means that the TPM shall be continuously powered for the duration of a timeout.

If *newRecoveryTime* is zero, then DA protection is disabled. Authorizations are checked but authorization failures will not cause the TPM to enter lockout.

If *newMaxTries* is zero, the TPM will be in lockout and use of DA protected entities will be disabled.

If *lockoutRecovery* is zero, then the recovery interval is _TPM_Init followed by TPM2_Startup().

Only one *lockoutAuth* authorization failure is allowed for this command during a *lockoutRecovery* interval.

25.3.2 Command and Response

Table 182 — TPM2_DictionaryAttackParameters Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_DictionaryAttackParameters {NV}
TPMI_RH_LOCKOUT	@lockHandle	TPM_RH_LOCKOUT Auth Index: 1 Auth Role: USER
UINT32	newMaxTries	count of authorization failures before the lockout is imposed
UINT32	newRecoveryTime	time in seconds before the authorization failure count is automatically decremented A value of zero indicates that DA protection is disabled.
UINT32	lockoutRecovery	time in seconds after a *lockoutAuth* failure before use of *lockoutAuth* is allowed A value of zero indicates that a reboot is required.

Table 183 — TPM2_DictionaryAttackParameters Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

25.3.3 Detailed Actions

```
#include "Tpm.h"
#include "DictionaryAttackParameters_fp.h"
#if CC_DictionaryAttackParameters  // Conditional expansion of this file
TPM_RC
TPM2_DictionaryAttackParameters(
    DictionaryAttackParameters_In   *in             // IN: input parameter list
    )
{
    // The command needs NV update.
    RETURN_IF_NV_IS_NOT_AVAILABLE;

// Internal Data Update

    // Set dictionary attack parameters
    gp.maxTries = in->newMaxTries;
    gp.recoveryTime = in->newRecoveryTime;
    gp.lockoutRecovery = in->lockoutRecovery;

#if 0   // Errata eliminates this code
    // This functionality has been disabled. The preferred implementation is now
    // to leave failedTries unchanged when the parameters are changed. This could
    // have the effect of putting the TPM into DA lockout if in->newMaxTries is
    // not greater than the current value of gp.failedTries.
    // Set failed tries to 0
    gp.failedTries = 0;
#endif

    // Record the changes to NV
    NV_SYNC_PERSISTENT(failedTries);
    NV_SYNC_PERSISTENT(maxTries);
    NV_SYNC_PERSISTENT(recoveryTime);
    NV_SYNC_PERSISTENT(lockoutRecovery);

    return TPM_RC_SUCCESS;
}
#endif // CC_DictionaryAttackParameters
```

26 Miscellaneous Management Functions

26.1 Introduction

This clause contains commands that do not logically group with any other commands.

26.2 TPM2_PP_Commands

26.2.1 General Description

This command is used to determine which commands require assertion of Physical Presence (PP) in addition to *platformAuth*/*platformPolicy*.

This command requires that *auth* is TPM_RH_PLATFORM and that Physical Presence be asserted.

After this command executes successfully, the commands listed in *setList* will be added to the list of commands that require that Physical Presence be asserted when the handle associated with the authorization is TPM_RH_PLATFORM. The commands in *clearList* will no longer require assertion of Physical Presence in order to authorize a command.

If a command is not in either list, its state is not changed. If a command is in both lists, then it will no longer require Physical Presence (for example, *setList* is processed first).

Only commands with handle types of TPMI_RH_PLATFORM, TPMI_RH_PROVISION, TPMI_RH_CLEAR, or TPMI_RH_HIERARCHY can be gated with Physical Presence. If any other command is in either list, it is discarded.

When a command requires that Physical Presence be provided, then Physical Presence shall be asserted for either an HMAC or a Policy authorization.

NOTE 1 Physical Presence may be made a requirement of any policy.

NOTE 2 If the TPM does not implement this command, the command list is vendor specific. A platform-specific specification may require that the command list be initialized in a specific way.

TPM2_PP_Commands() always requires assertion of Physical Presence.

26.2.2 Command and Response

Table 184 — TPM2_PP_Commands Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_PP_Commands {NV}
TPMI_RH_PLATFORM	@auth	TPM_RH_PLATFORM+PP Auth Index: 1 Auth Role: USER + Physical Presence
TPML_CC	setList	list of commands to be added to those that will require that Physical Presence be asserted
TPML_CC	clearList	list of commands that will no longer require that Physical Presence be asserted

Table 185 — TPM2_PP_Commands Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

26.2.3 Detailed Actions

```
#include "Tpm.h"
#include "PP_Commands_fp.h"
#if CC_PP_Commands  // Conditional expansion of this file
TPM_RC
TPM2_PP_Commands(
    PP_Commands_In  *in             // IN: input parameter list
    )
{
    UINT32          i;

    // The command needs NV update.  Check if NV is available.
    // A TPM_RC_NV_UNAVAILABLE or TPM_RC_NV_RATE error may be returned at
    // this point
    RETURN_IF_NV_IS_NOT_AVAILABLE;

// Internal Data Update

    // Process set list
    for(i = 0; i < in->setList.count; i++)
        // If command is implemented, set it as PP required.  If the input
        // command is not a PP command, it will be ignored at
        // PhysicalPresenceCommandSet().
        // Note: PhysicalPresenceCommandSet() checks if the command is implemented.
        PhysicalPresenceCommandSet(in->setList.commandCodes[i]);

    // Process clear list
    for(i = 0; i < in->clearList.count; i++)
        // If command is implemented, clear it as PP required.  If the input
        // command is not a PP command, it will be ignored at
        // PhysicalPresenceCommandClear().  If the input command is
        // TPM2_PP_Commands, it will be ignored as well
        PhysicalPresenceCommandClear(in->clearList.commandCodes[i]);

    // Save the change of PP list
    NV_SYNC_PERSISTENT(ppList);

    return TPM_RC_SUCCESS;
}
#endif // CC_PP_Commands
```

26.3 TPM2_SetAlgorithmSet

26.3.1 General Description

This command allows the platform to change the set of algorithms that are used by the TPM. The *algorithmSet* setting is a vendor-dependent value.

If the changing of the algorithm set results in a change of the algorithms of PCR banks, then the TPM will need to be reset (_TPM_Init and TPM2_Startup(TPM_SU_CLEAR)) before the new PCR settings take effect. After this command executes successfully, if *startupType* in the next TPM2_Startup() is not TPM_SU_CLEAR, the TPM shall return TPM_RC_VALUE and may enter Failure mode.

Other than PCR, when an algorithm is no longer supported, the behavior of this command is vendor-dependent.

EXAMPLE Entities may remain resident. Persistent objects, transient objects, or sessions may be flushed. NV Indexes may be undefined. Policies may be erased.

NOTE The reference implementation does not have support for this command. In particular, it does not support use of this command to selectively disable algorithms. Proper support would require modification of the unmarshaling code so that each time an algorithm is unmarshaled, it would be verified as being enabled.

26.3.2 Command and Response

Table 186 — TPM2_SetAlgorithmSet Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_SetAlgorithmSet {NV}
TPMI_RH_PLATFORM	@authHandle	TPM_RH_PLATFORM Auth Index: 1 Auth Role: USER
UINT32	algorithmSet	a TPM vendor-dependent value indicating the algorithm set selection

Table 187 — TPM2_SetAlgorithmSet Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

26.3.3 Detailed Actions

```
#include "Tpm.h"
#include "SetAlgorithmSet_fp.h"
#if CC_SetAlgorithmSet  // Conditional expansion of this file
TPM_RC
TPM2_SetAlgorithmSet(
    SetAlgorithmSet_In  *in             // IN: input parameter list
    )
{
    // The command needs NV update.  Check if NV is available.
    // A TPM_RC_NV_UNAVAILABLE or TPM_RC_NV_RATE error may be returned at
    // this point
    RETURN_IF_NV_IS_NOT_AVAILABLE;

// Internal Data Update
    gp.algorithmSet = in->algorithmSet;

    // Write the algorithm set changes to NV
    NV_SYNC_PERSISTENT(algorithmSet);

    return TPM_RC_SUCCESS;
}
#endif // CC_SetAlgorithmSet
```

27 Field Upgrade

27.1 Introduction

This clause contains the commands for managing field upgrade of the firmware in the TPM. The field upgrade scheme may be used for replacement or augmentation of the firmware installed in the TPM.

EXAMPLE 1 If an algorithm is found to be flawed, a patch of that algorithm might be installed using the firmware upgrade process. The patch might be a replacement of a portion of the code or a complete replacement of the firmware.

EXAMPLE 2 If an additional set of ECC parameters is needed, the firmware process may be used to add the parameters to the TPM data set.

The field upgrade process uses two commands (TPM2_FieldUpgradeStart() and TPM2_FieldUpgradeData()). TPM2_FieldUpgradeStart() validates that a signature on the provided digest is from the TPM manufacturer and that proper authorization is provided using *platformPolicy*.

NOTE 1 The *platformPolicy* for field upgraded is defined by the PM and may include requirements that the upgrade be signed by the PM or the TPM owner and include any other constraints that are desired by the PM.

If the proper authorization is given, the TPM will retain the signed digest and enter the Field Upgrade mode (FUM). While in FUM, the TPM will accept TPM2_FieldUpgradeData() commands. It may accept other commands if it is able to complete them using the previously installed firmware. Otherwise, it will return TPM_RC_UPGRADE.

Each block of the field upgrade shall contain the digest of the next block of the field upgrade data. That digest shall be included in the digest of the previous block. The digest of the first block is signed by the TPM manufacturer. That signature and first block digest are the parameters for TPM2_FieldUpgradeStart(). The digest is saved in the TPM as the required digest for the next field upgrade data block and as the identifier of the field upgrade sequence.

For each field upgrade data block that is sent to the TPM by TPM2_FieldUpgradeData(), the TPM shall validate that the digest matches the required digest and if not, shall return TPM_RC_VALUE. The TPM shall extract the digest of the next expected block and return that value to the caller, along with the digest of the first data block of the update sequence.

The system may attempt to abandon the firmware upgrade by using a zero-length buffer in TPM2_FieldUpdateData(). If the TPM is able to resume operation using the firmware present when the upgrade started, then the TPM will indicate that it has abandon the update by setting the digest of the next block to the Empty Buffer. If the TPM cannot abandon the update, it will return the expected next digest.

The system may also attempt to abandon the update because of a power interruption. If the TPM is able to resume normal operations, then it will respond normally to TPM2_Startup(). If the TPM is not able to resume normal operations, then it will respond to any command but TPM2_FieldUpgradeData() with TPM_RC_UPGRADE.

After a _TPM_Init, system software may not be able to resume the field upgrade that was in process when the power interruption occurred. In such case, the TPM firmware may be reset to one of two other values:

- the original firmware that was installed at the factory (“initial firmware”); or
- the firmware that was in the TPM when the field upgrade process started (“previous firmware”).

The TPM retains the digest of the first block for these firmware images and checks to see if the first block after _TPM_Init matches either of those digests. If so, the firmware update process restarts and the original firmware may be loaded.

NOTE 2 The TPM is required to accept the previous firmware as either a vendor-provided update or as recovered from the TPM using TPM2_FirmwareRead().

When the last block of the firmware upgrade is loaded into the TPM (indicated to the TPM by data in the data block in a TPM vendor-specific manner), the TPM will complete the upgrade process. If the TPM is able to resume normal operations without a reboot, it will set the hash algorithm of the next block to TPM_ALG_NULL and return TPM_RC_SUCCESS. If a reboot is required, the TPM shall return TPM_RC_REBOOT in response to the last TPM2_FieldUpgradeData() and all subsequent TPM commands until a _TPM_Init is received.

NOTE 3 Because no additional data is allowed when the response code is not TPM_RC_SUCCESS, the TPM returns TPM_RC_SUCCESS for all calls to TPM2_FieldUpgradeData() except the last. In this manner, the TPM is able to indicate the digest of the next block. If a _TPM_Init occurs while the TPM is in FUM, the next block may be the digest for the first block of the original firmware. If it is not, then the TPM will not accept the original firmware until the next _TPM_Init when the TPM is in FUM.

During the field upgrade process, either the one specified in this clause or a vendor proprietary field upgrade process, the TPM should preserve:

- Primary Seeds;
- Hierarchy *authValue*, *authPolicy*, and *proof* values;
- Lockout *authValue* and authorization failure count values;
- PCR authValue and authPolicy values;
- NV Index allocations and contents;
- Persistent object allocations and contents; and
- Clock.

NOTE 4 A platform manufacturer may provide a means to change preserved data to accommodate a case where a field upgrade fixes a flaw that might have compromised TPM secrets.

27.2 TPM2_FieldUpgradeStart

27.2.1 General Description

This command uses *platformPolicy* and a TPM Vendor Authorization Key to authorize a Field Upgrade Manifest.

If the signature checks succeed, the authorization is valid and the TPM will accept TPM2_FieldUpgradeData().

This signature is checked against the loaded key referenced by *keyHandle*. This key will have a Name that is the same as a value that is part of the TPM firmware data. If the signature is not valid, the TPM shall return TPM_RC_SIGNATURE.

NOTE A loaded key is used rather than a hard-coded key to reduce the amount of memory needed for this key data in case more than one vendor key is needed.

27.2.2 Command and Response

Table 188 — TPM2_FieldUpgradeStart Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_FieldUpgradeStart
TPMI_RH_PLATFORM	@authorization	TPM_RH_PLATFORM+{PP} Auth Index:1 Auth Role: ADMIN
TPMI_DH_OBJECT	keyHandle	handle of a public area that contains the TPM Vendor Authorization Key that will be used to validate *manifestSignature* Auth Index: None
TPM2B_DIGEST	fuDigest	digest of the first block in the field upgrade sequence
TPMT_SIGNATURE	manifestSignature	signature over *fuDigest* using the key associated with *keyHandle* (not optional)

Table 189 — TPM2_FieldUpgradeStart Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

27.2.3 Detailed Actions

```
#include "Tpm.h"
#include "FieldUpgradeStart_fp.h"
#if CC_FieldUpgradeStart  // Conditional expansion of this file
TPM_RC
TPM2_FieldUpgradeStart(
    FieldUpgradeStart_In    *in             // IN: input parameter list
    )
{
    // Not implemented
    UNUSED_PARAMETER(in);
    return TPM_RC_SUCCESS;
}
#endif
```

27.3 TPM2_FieldUpgradeData

27.3.1 General Description

This command will take the actual field upgrade image to be installed on the TPM. The exact format of *fuData* is vendor-specific. This command is only possible following a successful TPM2_FieldUpgradeStart(). If the TPM has not received a properly authorized TPM2_FieldUpgradeStart(), then the TPM shall return TPM_RC_FIELDUPGRADE.

The TPM will validate that the digest of *fuData* matches an expected value. If so, the TPM may buffer or immediately apply the update. If the digest of *fuData* does not match an expected value, the TPM shall return TPM_RC_VALUE.

27.3.2 Command and Response

Table 190 — TPM2_FieldUpgradeData Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit or decrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_FieldUpgradeData {NV}
TPM2B_MAX_BUFFER	fuData	field upgrade image data

Table 191 — TPM2_FieldUpgradeData Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPMT_HA+	nextDigest	tagged digest of the next block TPM_ALG_NULL if field update is complete
TPMT_HA	firstDigest	tagged digest of the first block of the sequence

27.3.3 Detailed Actions

```
#include "Tpm.h"
#include "FieldUpgradeData_fp.h"
#if CC_FieldUpgradeData  // Conditional expansion of this file
TPM_RC
TPM2_FieldUpgradeData(
    FieldUpgradeData_In     *in,            // IN: input parameter list
    FieldUpgradeData_Out    *out            // OUT: output parameter list
    )
{
    // Not implemented
    UNUSED_PARAMETER(in);
    UNUSED_PARAMETER(out);
    return TPM_RC_SUCCESS;
}
#endif
```


27.4 TPM2_FirmwareRead

27.4.1 General Description

This command is used to read a copy of the current firmware installed in the TPM.

The presumption is that the data will be returned in reverse order so that the last block in the sequence would be the first block given to the TPM in case of a failure recovery. If the TPM2_FirmwareRead sequence completes successfully, then the data provided from the TPM will be sufficient to allow the TPM to recover from an abandoned upgrade of this firmware.

To start the sequence of retrieving the data, the caller sets *sequenceNumber* to zero. When the TPM has returned all the firmware data, the TPM will return the Empty Buffer as *fuData*.

The contents of *fuData* are opaque to the caller.

NOTE 1 The caller should retain the ordering of the update blocks so that the blocks sent to the TPM have the same size and inverse order as the blocks returned by a sequence of calls to this command.

NOTE 2 Support for this command is optional even if the TPM implements TPM2_FieldUpgradeStart() and TPM2_FieldUpgradeData().

27.4.2 Command and Response

Table 192 — TPM2_FirmwareRead Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit or encrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_FirmwareRead
UINT32	sequenceNumber	the number of previous calls to this command in this sequence set to 0 on the first call

Table 193 — TPM2_FirmwareRead Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_MAX_BUFFER	fuData	field upgrade image data

27.4.3 Detailed Actions

```
#include "Tpm.h"
#include "FirmwareRead_fp.h"
#if CC_FirmwareRead  // Conditional expansion of this file
TPM_RC
TPM2_FirmwareRead(
    FirmwareRead_In     *in,            // IN: input parameter list
    FirmwareRead_Out    *out            // OUT: output parameter list
    )
{
    // Not implemented
    UNUSED_PARAMETER(in);
    UNUSED_PARAMETER(out);
    return TPM_RC_SUCCESS;
}
#endif // CC_FirmwareRead
```


28 Context Management

28.1 Introduction

Three of the commands in this clause (TPM2_ContextSave(), TPM2_ContextLoad(), and TPM2_FlushContext()) implement the resource management described in the "Context Management" clause in TPM 2.0 Part 1.

The fourth command in this clause (TPM2_EvictControl()) is used to control the persistence of loadable objects in TPM memory. Background for this command may be found in the "Owner and Platform Evict Objects" clause in TPM 2.0 Part 1.

28.2 TPM2_ContextSave

28.2.1 General Description

This command saves a session context, object context, or sequence object context outside the TPM.

No authorization sessions of any type are allowed with this command and tag is required to be TPM_ST_NO_SESSIONS.

NOTE This preclusion avoids complex issues of dealing with the same session in *handle* and in the session area. While it might be possible to provide specificity, it would add unnecessary complexity to the TPM and, because this capability would provide no application benefit, use of authorization sessions for audit or encryption is prohibited.

The TPM shall encrypt and integrity protect the TPM2B_CONTEXT_SENSITIVE *context* as described in the "Context Protections" clause in TPM 2.0 Part 1.

See the “Context Data” clause in TPM 2.0 Part 2 for a description of the *context* structure in the response.

28.2.2 Command and Response

Table 194 — TPM2_ContextSave Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_ContextSave
TPMI_DH_CONTEXT	saveHandle	handle of the resource to save Auth Index: None

Table 195 — TPM2_ContextSave Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPMS_CONTEXT	context	

28.2.3 Detailed Actions

```
#include "Tpm.h"
#include "ContextSave_fp.h"
#if CC_ContextSave  // Conditional expansion of this file
#include "Context_spt_fp.h"
```

Error Returns	Meaning
TPM_RC_CONTEXT_GAP	a *contextID* could not be assigned for a session context save
TPM_RC_TOO_MANY_CONTEXTS	no more contexts can be saved as the counter has maxed out

```
TPM_RC
TPM2_ContextSave(
    ContextSave_In      *in,            // IN: input parameter list
    ContextSave_Out     *out            // OUT: output parameter list
    )
{
    TPM_RC          result = TPM_RC_SUCCESS;
    UINT16          fingerprintSize;    // The size of fingerprint in context
    // blob.
    UINT64          contextID = 0;      // session context ID
    TPM2B_SYM_KEY   symKey;
    TPM2B_IV        iv;

    TPM2B_DIGEST    integrity;
    UINT16          integritySize;
    BYTE            *buffer;

    // This command may cause the orderlyState to be cleared due to
    // the update of state reset data. If the state is orderly and
    // cannot be changed, exit early.
    RETURN_IF_ORDERLY;

// Internal Data Update

// This implementation does not do things in quite the same way as described in
// Part 2 of the specification. In Part 2, it indicates that the
// TPMS_CONTEXT_DATA contains two TPM2B values. That is not how this is
// implemented. Rather, the size field of the TPM2B_CONTEXT_DATA is used to
// determine the amount of data in the encrypted data. That part is not
// independently sized. This makes the actual size 2 bytes smaller than
// calculated using Part 2. Since this is opaque to the caller, it is not
// necessary to fix. The actual size is returned by TPM2_GetCapabilties().

    // Initialize output handle.  At the end of command action, the output
    // handle of an object will be replaced, while the output handle
    // for a session will be the same as input
    out->context.savedHandle = in->saveHandle;

    // Get the size of fingerprint in context blob.  The sequence value in
    // TPMS_CONTEXT structure is used as the fingerprint
    fingerprintSize = sizeof(out->context.sequence);

    // Compute the integrity size at the beginning of context blob
    integritySize = sizeof(integrity.t.size)
        + CryptHashGetDigestSize(CONTEXT_INTEGRITY_HASH_ALG);

// Perform object or session specific context save
    switch(HandleGetType(in->saveHandle))
    {
        case TPM_HT_TRANSIENT:
```

```
        {
            OBJECT              *object = HandleToObject(in->saveHandle);
            ANY_OBJECT_BUFFER   *outObject;
            UINT16               objectSize = ObjectIsSequence(object)
                ? sizeof(HASH_OBJECT) : sizeof(OBJECT);

            outObject = (ANY_OBJECT_BUFFER *)(out->context.contextBlob.t.buffer
                                              + integritySize + fingerprintSize);

            // Set size of the context data.  The contents of context blob is vendor
            // defined.  In this implementation, the size is size of integrity
            // plus fingerprint plus the whole internal OBJECT structure
            out->context.contextBlob.t.size = integritySize +
                fingerprintSize + objectSize;
#if ALG_RSA
            // For an RSA key, make sure that the key has had the private exponent
            // computed before saving.
            if(object->publicArea.type == TPM_ALG_RSA &&
               !(object->attributes.publicOnly))
                CryptRsaLoadPrivateExponent(&object->publicArea, &object->sensitive);
#endif
            // Make sure things fit
            pAssert(out->context.contextBlob.t.size
                    <= sizeof(out->context.contextBlob.t.buffer));
            // Copy the whole internal OBJECT structure to context blob
            MemoryCopy(outObject, object, objectSize);

            // Increment object context ID
            gr.objectContextID++;
            // If object context ID overflows, TPM should be put in failure mode
            if(gr.objectContextID == 0)
                FAIL(FATAL_ERROR_INTERNAL);

            // Fill in other return values for an object.
            out->context.sequence = gr.objectContextID;
            // For regular object, savedHandle is 0x80000000.  For sequence object,
            // savedHandle is 0x80000001.  For object with stClear, savedHandle
            // is 0x80000002
            if(ObjectIsSequence(object))
            {
                out->context.savedHandle = 0x80000001;
                SequenceDataExport((HASH_OBJECT *)object,
                                   (HASH_OBJECT_BUFFER *)outObject);
            }
            else
                out->context.savedHandle = (object->attributes.stClear == SET)
                ? 0x80000002 : 0x80000000;
// Get object hierarchy
            out->context.hierarchy = ObjectGetHierarchy(object);

            break;
        }
        case TPM_HT_HMAC_SESSION:
        case TPM_HT_POLICY_SESSION:
        {
            SESSION         *session = SessionGet(in->saveHandle);

            // Set size of the context data.  The contents of context blob is vendor
            // defined.  In this implementation, the size of context blob is the
            // size of a internal session structure plus the size of
            // fingerprint plus the size of integrity
            out->context.contextBlob.t.size = integritySize +
                fingerprintSize + sizeof(*session);

            // Make sure things fit
            pAssert(out->context.contextBlob.t.size
```

```
                    < sizeof(out->context.contextBlob.t.buffer));

            // Copy the whole internal SESSION structure to context blob.
            // Save space for fingerprint at the beginning of the buffer
            // This is done before anything else so that the actual context
            // can be reclaimed after this call
            pAssert(sizeof(*session) <= sizeof(out->context.contextBlob.t.buffer)
                    - integritySize - fingerprintSize);
            MemoryCopy(out->context.contextBlob.t.buffer + integritySize
                       + fingerprintSize, session, sizeof(*session));
           // Fill in the other return parameters for a session
           // Get a context ID and set the session tracking values appropriately
           // TPM_RC_CONTEXT_GAP is a possible error.
           // SessionContextSave() will flush the in-memory context
           // so no additional errors may occur after this call.
            result = SessionContextSave(out->context.savedHandle, &contextID);
            if(result != TPM_RC_SUCCESS)
                return result;
            // sequence number is the current session contextID
            out->context.sequence = contextID;

            // use TPM_RH_NULL as hierarchy for session context
            out->context.hierarchy = TPM_RH_NULL;

            break;
        }
        default:
            // SaveContext may only take an object handle or a session handle.
            // All the other handle type should be filtered out at unmarshal
            FAIL(FATAL_ERROR_INTERNAL);
            break;
    }

    // Save fingerprint at the beginning of encrypted area of context blob.
    // Reserve the integrity space
    pAssert(sizeof(out->context.sequence) <=
            sizeof(out->context.contextBlob.t.buffer) - integritySize);
    MemoryCopy(out->context.contextBlob.t.buffer + integritySize,
               &out->context.sequence, sizeof(out->context.sequence));

    // Compute context encryption key
    ComputeContextProtectionKey(&out->context, &symKey, &iv);

    // Encrypt context blob
    CryptSymmetricEncrypt(out->context.contextBlob.t.buffer + integritySize,
                          CONTEXT_ENCRYPT_ALG, CONTEXT_ENCRYPT_KEY_BITS,
                          symKey.t.buffer, &iv, ALG_CFB_VALUE,
                          out->context.contextBlob.t.size - integritySize,
                          out->context.contextBlob.t.buffer + integritySize);

    // Compute integrity hash for the object
    // In this implementation, the same routine is used for both sessions
    // and objects.
    ComputeContextIntegrity(&out->context, &integrity);

    // add integrity at the beginning of context blob
    buffer = out->context.contextBlob.t.buffer;
    TPM2B_DIGEST_Marshal(&integrity, &buffer, NULL);

    // orderly state should be cleared because of the update of state reset and
    // state clear data
    g_clearOrderly = TRUE;

    return result;
}
#endif // CC_ContextSave
```

28.3 TPM2_ContextLoad

28.3.1 General Description

This command is used to reload a context that has been saved by TPM2_ContextSave().

No authorization sessions of any type are allowed with this command and tag is required to be TPM_ST_NO_SESSIONS (see note in 28.2.1).

The TPM will return TPM_RC_HIERARCHY if the context is associated with a hierarchy that is disabled.

NOTE Contexts for authorization sessions and for sequence objects belong to the NULL hierarchy, which is never disabled.

See the "Context Data" clause in TPM 2.0 Part 2 for a description of the values in the *context* parameter.

If the integrity HMAC of the saved context is not valid, the TPM shall return TPM_RC_INTEGRITY.

The TPM shall perform a check on the decrypted context as described in the "Context Confidentiality Protection" clause of TPM 2.0 Part 1 and enter failure mode if the check fails.

28.3.2 Command and Response

Table 196 — TPM2_ContextLoad Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_ContextLoad
TPMS_CONTEXT	context	the context blob

Table 197 — TPM2_ContextLoad Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPMI_DH_CONTEXT	loadedHandle	the handle assigned to the resource after it has been successfully loaded

28.3.3 Detailed Actions

```
#include "Tpm.h"
#include "ContextLoad_fp.h"
#if CC_ContextLoad  // Conditional expansion of this file
#include "Context_spt_fp.h"
```

Error Returns	Meaning
TPM_RC_CONTEXT_GAP	there is only one available slot and this is not the oldest saved session context
TPM_RC_HANDLE	*context.savedHandle*' does not reference a saved session
TPM_RC_HIERARCHY	*context.hierarchy* is disabled
TPM_RC_INTEGRITY	*context* integrity check fail
TPM_RC_OBJECT_MEMORY	no free slot for an object
TPM_RC_SESSION_MEMORY	no free session slots
TPM_RC_SIZE	incorrect context blob size

```
TPM_RC
TPM2_ContextLoad(
    ContextLoad_In      *in,            // IN: input parameter list
    ContextLoad_Out     *out            // OUT: output parameter list
    )
{
    TPM_RC              result;
    TPM2B_DIGEST        integrityToCompare;
    TPM2B_DIGEST        integrity;
    BYTE                *buffer;    // defined to save some typing
    INT32               size;       // defined to save some typing
    TPM_HT              handleType;
    TPM2B_SYM_KEY       symKey;
    TPM2B_IV            iv;

// Input Validation

// See discussion about the context format in TPM2_ContextSave Detailed Actions

    // IF this is a session context, make sure that the sequence number is
    // consistent with the version in the slot

    // Check context blob size
    handleType = HandleGetType(in->context.savedHandle);

    // Get integrity from context blob
    buffer = in->context.contextBlob.t.buffer;
    size = (INT32)in->context.contextBlob.t.size;
    result = TPM2B_DIGEST_Unmarshal(&integrity, &buffer, &size);
    if(result != TPM_RC_SUCCESS)
        return result;

    // the size of the integrity value has to match the size of digest produced
    // by the integrity hash
    if(integrity.t.size != CryptHashGetDigestSize(CONTEXT_INTEGRITY_HASH_ALG))
        return TPM_RCS_SIZE + RC_ContextLoad_context;

    // Make sure that the context blob has enough space for the fingerprint. This
    // is elastic pants to go with the belt and suspenders we already have to make
    // sure that the context is complete and untampered.
```

```
    if((unsigned)size < sizeof(in->context.sequence))
        return TPM_RCS_SIZE + RC_ContextLoad_context;

    // After unmarshaling the integrity value, 'buffer' is pointing at the first
    // byte of the integrity protected and encrypted buffer and 'size' is the number
    // of integrity protected and encrypted bytes.

    // Compute context integrity
    ComputeContextIntegrity(&in->context, &integrityToCompare);

    // Compare integrity
    if(!MemoryEqual2B(&integrity.b, &integrityToCompare.b))
        return TPM_RCS_INTEGRITY + RC_ContextLoad_context;
    // Compute context encryption key
    ComputeContextProtectionKey(&in->context, &symKey, &iv);

    // Decrypt context data in place
    CryptSymmetricDecrypt(buffer, CONTEXT_ENCRYPT_ALG, CONTEXT_ENCRYPT_KEY_BITS,
                          symKey.t.buffer, &iv, ALG_CFB_VALUE, size, buffer);
    // See if the fingerprint value matches. If not, it is symptomatic of either
    // a broken TPM or that the TPM is under attack so go into failure mode.
    if(!MemoryEqual(buffer, &in->context.sequence, sizeof(in->context.sequence)))
        FAIL(FATAL_ERROR_INTERNAL);

    // step over fingerprint
    buffer += sizeof(in->context.sequence);

    // set the remaining size of the context
    size -= sizeof(in->context.sequence);

    // Perform object or session specific input check
    switch(handleType)
    {
        case TPM_HT_TRANSIENT:
        {
            OBJECT      *outObject;

            if(size > (INT32)sizeof(OBJECT))
                FAIL(FATAL_ERROR_INTERNAL);

            // Discard any changes to the handle that the TRM might have made
            in->context.savedHandle = TRANSIENT_FIRST;

            // If hierarchy is disabled, no object context can be loaded in this
            // hierarchy
            if(!HierarchyIsEnabled(in->context.hierarchy))
                return TPM_RCS_HIERARCHY + RC_ContextLoad_context;

            // Restore object. If there is no empty space, indicate as much
            outObject = ObjectContextLoad((ANY_OBJECT_BUFFER *)buffer,
                                          &out->loadedHandle);
            if(outObject == NULL)
                return TPM_RC_OBJECT_MEMORY;

            break;
        }
        case TPM_HT_POLICY_SESSION:
        case TPM_HT_HMAC_SESSION:
        {
            if(size != sizeof(SESSION))
                FAIL(FATAL_ERROR_INTERNAL);

            // This command may cause the orderlyState to be cleared due to
            // the update of state reset data.  If this is the case, check if NV is
            // available first
            RETURN_IF_ORDERLY;
```

```

            // Check if input handle points to a valid saved session and that the
            // sequence number makes sense
            if(!SequenceNumberForSavedContextIsValid(&in->context))
                return TPM_RCS_HANDLE + RC_ContextLoad_context;

            // Restore session.  A TPM_RC_SESSION_MEMORY, TPM_RC_CONTEXT_GAP error
            // may be returned at this point
            result = SessionContextLoad((SESSION_BUF *)buffer,
                                        &in->context.savedHandle);
            if(result != TPM_RC_SUCCESS)
                return result;

            out->loadedHandle = in->context.savedHandle;

            // orderly state should be cleared because of the update of state
            // reset and state clear data
            g_clearOrderly = TRUE;

            break;
        }
        default:
            // Context blob may only have an object handle or a session handle.
            // All the other handle type should be filtered out at unmarshal
            FAIL(FATAL_ERROR_INTERNAL);
            break;
    }

    return TPM_RC_SUCCESS;
}
#endif // CC_ContextLoad
```

28.4 TPM2_FlushContext

28.4.1 General Description

This command causes all context associated with a loaded object, sequence object, or session to be removed from TPM memory.

This command may not be used to remove a persistent object from the TPM. Use TPM2_EvictControl to remove a persistent object.

A session does not have to be loaded in TPM memory to have its context flushed. The saved session context associated with the indicated handle is invalidated. When flushing a session, the upper byte of the handle is ignored.

EXAMPLE A command to flush session handle 0x20000000 will flush session handle 0x03000000.

No sessions of any type are allowed with this command and tag is required to be TPM_ST_NO_SESSIONS (see note in 28.2.1).

If the handle is for a Transient Object and the handle is not associated with a loaded object, then the TPM shall return TPM_RC_HANDLE.

If the handle is for an authorization session and the handle does not reference a loaded or active session, then the TPM shall return TPM_RC_HANDLE.

NOTE *flushHandle* is a parameter and not a handle. If it were in the handle area, the TPM would validate that the context for the referenced entity is in the TPM. When a TPM2_FlushContext references a saved session context, it is not necessary for the context to be in the TPM. When the *flushHandle* is in the parameter area, the TPM does not validate that associated context is actually in the TPM.

28.4.2 Command and Response

Table 198 — TPM2_FlushContext Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_FlushContext
TPMI_DH_CONTEXT	flushHandle	the handle of the item to flush NOTE This is a use of a handle as a parameter.

Table 199 — TPM2_FlushContext Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

28.4.3 Detailed Actions

```
#include "Tpm.h"
#include "FlushContext_fp.h"
#if CC_FlushContext  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_HANDLE	*flushHandle* does not reference a loaded object or session

```
TPM_RC
TPM2_FlushContext(
    FlushContext_In     *in             // IN: input parameter list
    )
{
// Internal Data Update

    // Call object or session specific routine to flush
    switch(HandleGetType(in->flushHandle))
    {
        case TPM_HT_TRANSIENT:
            if(!IsObjectPresent(in->flushHandle))
                return TPM_RCS_HANDLE + RC_FlushContext_flushHandle;
           // Flush object
            FlushObject(in->flushHandle);
            break;
        case TPM_HT_HMAC_SESSION:
        case TPM_HT_POLICY_SESSION:
            if(!SessionIsLoaded(in->flushHandle)
               && !SessionIsSaved(in->flushHandle)
               )
                return TPM_RCS_HANDLE + RC_FlushContext_flushHandle;

            // If the session to be flushed is the exclusive audit session, then
            // indicate that there is no exclusive audit session any longer.
            if(in->flushHandle == g_exclusiveAuditSession)
                g_exclusiveAuditSession = TPM_RH_UNASSIGNED;

            // Flush session
            SessionFlush(in->flushHandle);
            break;
        default:
            // This command only takes object or session handle.  Other handles
            // should be filtered out at handle unmarshal
            FAIL(FATAL_ERROR_INTERNAL);
            break;
    }

    return TPM_RC_SUCCESS;
}
#endif // CC_FlushContext
```

28.5 TPM2_EvictControl

28.5.1 General Description

This command allows certain Transient Objects to be made persistent or a persistent object to be evicted.

NOTE 1 A transient object is one that may be removed from TPM memory using either TPM2_FlushContext or TPM2_Startup(). A persistent object is not removed from TPM memory by TPM2_FlushContext() or TPM2_Startup().

If *objectHandle* is a Transient Object, then this call makes a persistent copy of the object and assigns *persistentHandle* to the persistent version of the object. If *objectHandle* is a persistent object, then the call evicts the persistent object. The call does not affect the transient object.

Before execution of TPM2_EvictControl code below, the TPM verifies that *objectHandle* references an object that is resident on the TPM and that *persistentHandle* is a valid handle for a persistent object.

NOTE 2 This requirement simplifies the unmarshaling code so that it only need check that *persistentHandle* is always a persistent object.

If *objectHandle* references a Transient Object:

a) The TPM shall return TPM_RC_ATTRIBUTES if
   1) it is in the hierarchy of TPM_RH_NULL,
   2) only the public portion of the object is loaded, or

      NOTE 3 This is for NV space efficiency. Loading an object whose private part is empty would unnecessarily consume NV resources.

   3) the *stClear* is SET in the object or in an ancestor key.
b) The TPM shall return TPM_RC_HIERARCHY if the object is not in the proper hierarchy as determined by *auth*.
   1) If *auth* is TPM_RH_PLATFORM, the proper hierarchy is the Platform hierarchy.
   2) If *auth* is TPM_RH_OWNER, the proper hierarchy is either the Storage or the Endorsement hierarchy.
c) The TPM shall return TPM_RC_RANGE if *persistentHandle* is not in the proper range as determined by *auth*.
   1) If *auth* is TPM_RH_OWNER, then *persistentHandle* shall be in the inclusive range of 81 00 00 00_{16} to 81 7F FF FF_{16}.
   2) If *auth* is TPM_RH_PLATFORM, then *persistentHandle* shall be in the inclusive range of 81 80 00 00_{16} to 81 FF FF FF_{16}.

   NOTE 4 This separation permits the platform (the platform OEM) a range of indexes that will not interfere with indexes used by the TPM owner (the OS or applications).

d) The TPM shall return TPM_RC_NV_DEFINED if a persistent object exists with the same handle as *persistentHandle*.
e) The TPM shall return TPM_RC_NV_SPACE if insufficient space is available to make the object persistent.
f) The TPM shall return TPM_RC_NV_SPACE if execution of this command will prevent the TPM from being able to hold two transient objects of any kind.

   NOTE 5 This requirement anticipates that a TPM may be implemented such that all TPM memory is non-volatile and not subject to endurance issues. In such case, there is no movement of an object between memory of different types and it is necessary that the TPM ensure that it is always possible for the management software to move objects to/from TPM memory in order to ensure that the objects required for command execution can be context restored.

g) If the TPM returns TPM_RC_SUCCESS, the object referenced by *objectHandle* will not be flushed and both *objectHandle* and *persistentHandle* may be used to access the object.

If *objectHandle* references a persistent object:

a) The TPM shall return TPM_RC_RANGE if *objectHandle* is not in the proper range as determined by *auth*. If *auth* is TPM_RC_OWNER, *objectHandle* shall be in the inclusive range of 81 00 00 00_{16} to 81 7F FF FF_{16}. If *auth* is TPM_RC_PLATFORM, *objectHandle* may be any valid persistent object handle.

b) If objectHandle is not the same value as persistentHandle, return TPM_RC_HANDLE.

c) If the TPM returns TPM_RC_SUCCESS, *objectHandle* will be removed from persistent memory and no longer be accessible.

NOTE 5 The persistent object is not converted to a transient object, as this would prevent the immediate revocation of an object by removing it from persistent memory.

28.5.2 Command and Response

Table 200 — TPM2_EvictControl Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_EvictControl {NV}
TPMI_RH_PROVISION	@auth	TPM_RH_OWNER or TPM_RH_PLATFORM+{PP} Auth Handle: 1 Auth Role: USER
TPMI_DH_OBJECT	objectHandle	the handle of a loaded object Auth Index: None
TPMI_DH_PERSISTENT	persistentHandle	if *objectHandle* is a transient object handle, then this is the persistent handle for the object if *objectHandle* is a persistent object handle, then it shall be the same value as *persistentHandle*

Table 201 — TPM2_EvictControl Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

28.5.3 Detailed Actions

```
#include "Tpm.h"
#include "EvictControl_fp.h"
#if CC_EvictControl  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_ATTRIBUTES	an object with *temporary*, *stClear* or *publicOnly* attribute SET cannot be made persistent
TPM_RC_HIERARCHY	*auth* cannot authorize the operation in the hierarchy of *evictObject*
TPM_RC_HANDLE	*evictHandle* of the persistent object to be evicted is not the same as the *persistentHandle* argument
TPM_RC_NV_HANDLE	*persistentHandle* is unavailable
TPM_RC_NV_SPACE	no space in NV to make *evictHandle* persistent
TPM_RC_RANGE	*persistentHandle* is not in the range corresponding to the hierarchy of *evictObject*

```
TPM_RC
TPM2_EvictControl(
    EvictControl_In     *in             // IN: input parameter list
    )
{
    TPM_RC      result;
    OBJECT      *evictObject;

// Input Validation

    // Get internal object pointer
    evictObject = HandleToObject(in->objectHandle);

    // Temporary, stClear or public only objects can not be made persistent
    if(evictObject->attributes.temporary == SET
       || evictObject->attributes.stClear == SET
       || evictObject->attributes.publicOnly == SET)
        return TPM_RCS_ATTRIBUTES + RC_EvictControl_objectHandle;

    // If objectHandle refers to a persistent object, it should be the same as
    // input persistentHandle
    if(evictObject->attributes.evict == SET
       && evictObject->evictHandle != in->persistentHandle)
        return TPM_RCS_HANDLE + RC_EvictControl_objectHandle;

    // Additional authorization validation
    if(in->auth == TPM_RH_PLATFORM)
    {
        // To make persistent
        if(evictObject->attributes.evict == CLEAR)
        {
            // PlatformAuth can not set evict object in storage or endorsement
            // hierarchy
            if(evictObject->attributes.ppsHierarchy == CLEAR)
                return TPM_RCS_HIERARCHY + RC_EvictControl_objectHandle;
            // Platform cannot use a handle outside of platform persistent range.
            if(!NvIsPlatformPersistentHandle(in->persistentHandle))
                return TPM_RCS_RANGE + RC_EvictControl_persistentHandle;
        }
        // PlatformAuth can delete any persistent object
    }
```

```
    else if(in->auth == TPM_RH_OWNER)
    {
        // OwnerAuth can not set or clear evict object in platform hierarchy
        if(evictObject->attributes.ppsHierarchy == SET)
            return TPM_RCS_HIERARCHY + RC_EvictControl_objectHandle;

        // Owner cannot use a handle outside of owner persistent range.
        if(evictObject->attributes.evict == CLEAR
           && !NvIsOwnerPersistentHandle(in->persistentHandle))
            return TPM_RCS_RANGE + RC_EvictControl_persistentHandle;
    }
    else
    {
        // Other authorization is not allowed in this command and should have been
        // filtered out in unmarshal process
        FAIL(FATAL_ERROR_INTERNAL);
    }
// Internal Data Update
    // Change evict state
    if(evictObject->attributes.evict == CLEAR)
    {
        // Make object persistent
        if(NvFindHandle(in->persistentHandle) != 0)
            return TPM_RC_NV_DEFINED;
        // A TPM_RC_NV_HANDLE or TPM_RC_NV_SPACE error may be returned at this
        // point
        result = NvAddEvictObject(in->persistentHandle, evictObject);
    }
    else
    {
        // Delete the persistent object in NV
        result = NvDeleteEvict(evictObject->evictHandle);
    }
    return result;
}
#endif // CC_EvictControl
```


29 Clocks and Timers

29.1 TPM2_ReadClock

29.1.1 General Description

This command reads the current TPMS_TIME_INFO structure that contains the current setting of *Time*, *Clock*, *resetCount*, and *restartCount*.

29.1.2 Command and Response

Table 202 — TPM2_ReadClock Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_ReadClock

Table 203 — TPM2_ReadClock Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPMS_TIME_INFO	currentTime	

29.1.3 Detailed Actions

```
#include "Tpm.h"
#include "ReadClock_fp.h"
#if CC_ReadClock  // Conditional expansion of this file
TPM_RC
TPM2_ReadClock(
    ReadClock_Out   *out            // OUT: output parameter list
    )
{
// Command Output

    out->currentTime.time = g_time;
    TimeFillInfo(&out->currentTime.clockInfo);

    return TPM_RC_SUCCESS;
}
#endif // CC_ReadClock
```

29.2 TPM2_ClockSet

29.2.1 General Description

This command is used to advance the value of the TPM's *Clock*. The command will fail if *newTime* is less than the current value of *Clock* or if the new time is greater than FF FF 00 00 00 00 00 00_{16}. If both of these checks succeed, *Clock* is set to *newTime*. If either of these checks fails, the TPM shall return TPM_RC_VALUE and make no change to *Clock*.

NOTE This maximum setting would prevent *Clock* from rolling over to zero for approximately 8,000 years at the real time *Clock* update rate. If the *Clock* update rate was set so that TPM time was passing 33 percent faster than real time, it would still be more than 6,000 years before *Clock* would roll over to zero. Because *Clock* will not roll over in the lifetime of the TPM, there is no need for external software to deal with the possibility that *Clock* may wrap around.

If the value of *Clock* after the update makes the volatile and non-volatile versions of TPMS_CLOCK_INFO.*clock* differ by more than the reported update interval, then the TPM shall update the non-volatile version of TPMS_CLOCK_INFO.*clock* before returning.

This command requires Platform Authorization or Owner Authorization.

29.2.2 Command and Response

Table 204 — TPM2_ClockSet Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_ClockSet {NV}
TPMI_RH_PROVISION	@auth	TPM_RH_OWNER or TPM_RH_PLATFORM+{PP} Auth Handle: 1 Auth Role: USER
UINT64	newTime	new *Clock* setting in milliseconds

Table 205 — TPM2_ClockSet Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

29.2.3 Detailed Actions

```
#include "Tpm.h"
#include "ClockSet_fp.h"
#if CC_ClockSet  // Conditional expansion of this file
```

Read the current TPMS_TIMER_INFO structure settings

Error Returns	Meaning
TPM_RC_NV_RATE	NV is unavailable because of rate limit
TPM_RC_NV_UNAVAILABLE	NV is inaccessible
TPM_RC_VALUE	invalid new clock

```
TPM_RC
TPM2_ClockSet(
    ClockSet_In     *in             // IN: input parameter list
    )
{
// Input Validation
    // new time can not be bigger than 0xFFFF000000000000 or smaller than
    // current clock
    if(in->newTime > 0xFFFF000000000000ULL
       || in->newTime < go.clock)
        return TPM_RCS_VALUE + RC_ClockSet_newTime;

// Internal Data Update
    // Can't modify the clock if NV is not available.
    RETURN_IF_NV_IS_NOT_AVAILABLE;

    TimeClockUpdate(in->newTime);
    return TPM_RC_SUCCESS;
}
#endif // CC_ClockSet
```


29.3 TPM2_ClockRateAdjust

29.3.1 General Description

This command adjusts the rate of advance of *Clock* and *Time* to provide a better approximation to real time.

The *rateAdjust* value is relative to the current rate and not the nominal rate of advance.

EXAMPLE 1 If this command had been called three times with *rateAdjust* = TPM_CLOCK_COARSE_SLOWER and once with *rateAdjust* = TPM_CLOCK_COARSE_FASTER, the net effect will be as if the command had been called twice with *rateAdjust* = TPM_CLOCK_COARSE_SLOWER.

The range of adjustment shall be sufficient to allow *Clock* and *Time* to advance at real time but no more. If the requested adjustment would make the rate advance faster or slower than the nominal accuracy of the input frequency, the TPM shall return TPM_RC_VALUE.

EXAMPLE 2 If the frequency tolerance of the TPM's input clock is +/-10 percent, then the TPM will return TPM_RC_VALUE if the adjustment would make *Clock* run more than 10 percent faster or slower than nominal. That is, if the input oscillator were nominally 100 megahertz (MHz), then 1 millisecond (ms) would normally take 100,000 counts. The update *Clock* should be adjustable so that 1 ms is between 90,000 and 110,000 counts.

The interpretation of “fine” and “coarse” adjustments is implementation-specific.

The nominal rate of advance for *Clock* and *Time* shall be accurate to within 15 percent. That is, with no adjustment applied, *Clock* and *Time* shall be advanced at a rate within 15 percent of actual time.

NOTE If the adjustments are incorrect, it will be possible to make the difference between advance of *Clock/Time* and real time to be as much as 1.15^2 or ~1.33.

Changes to the current *Clock* update rate adjustment need not be persisted across TPM power cycles.

29.3.2 Command and Response

Table 206 — TPM2_ClockRateAdjust Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_ClockRateAdjust
TPMI_RH_PROVISION	@auth	TPM_RH_OWNER or TPM_RH_PLATFORM+{PP} Auth Handle: 1 Auth Role: USER
TPM_CLOCK_ADJUST	rateAdjust	Adjustment to current *Clock* update rate

Table 207 — TPM2_ClockRateAdjust Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

29.3.3 Detailed Actions

```
#include "Tpm.h"
#include "ClockRateAdjust_fp.h"
#if CC_ClockRateAdjust  // Conditional expansion of this file
TPM_RC
TPM2_ClockRateAdjust(
    ClockRateAdjust_In  *in             // IN: input parameter list
    )
{
// Internal Data Update
    TimeSetAdjustRate(in->rateAdjust);

    return TPM_RC_SUCCESS;
}
#endif // CC_ClockRateAdjust
```


30 Capability Commands

30.1 Introduction

The TPM has numerous values that indicate the state, capabilities, and properties of the TPM. These values are needed for proper management of the TPM. The TPM2_GetCapability() command is used to access these values.

TPM2_GetCapability() allows reporting of multiple values in a single call. The values are grouped according to type.

NOTE TPM2_TestParms()is used to determine if a TPM supports a particular combination of algorithm parameters

30.2 TPM2_GetCapability

30.2.1 General Description

This command returns various information regarding the TPM and its current state.

The *capability* parameter determines the category of data returned. The *property* parameter selects the first value of the selected category to be returned. If there is no property that corresponds to the value of *property*, the next higher value is returned, if it exists.

EXAMPLE 1 The list of handles of transient objects currently loaded in the TPM may be read one at a time. On the first read, set the property to TRANSIENT_FIRST and *propertyCount* to one. If a transient object is present, the lowest numbered handle is returned and *moreData* will be YES if transient objects with higher handles are loaded. On the subsequent call, use returned handle value plus 1 in order to access the next higher handle.

The *propertyCount* parameter indicates the number of capabilities in the indicated group that are requested. The TPM will return no more than the number of requested values (*propertyCount*) or until the last property of the requested type has been returned.

NOTE 1 The type of the capability is derived from a combination of *capability* and *property*.

NOTE 2 If the *property* selects an unimplemented property, the next higher implemented property is returned.

When all of the properties of the requested type have been returned, the *moreData* parameter in the response will be set to NO. Otherwise, it will be set to YES.

NOTE 3 The *moreData* parameter will be YES if there are more properties even if the requested number of capabilities has been returned.

The TPM is not required to return more than one value at a time. It is not required to provide the same number of values in response to subsequent requests.

EXAMPLE 2 A TPM may return 4 properties in response to a TPM2_GetCapability(*capability* = TPM_CAP_TPM_PROPERTY, *property* = TPM_PT_MANUFACTURER, *propertyCount* = 8) and for a latter request with the same parameters, the TPM may return as few as one and as many as 8 values.

When the TPM is in Failure mode, a TPM is required to allow use of this command for access of the following capabilities:

- TPM_PT_MANUFACTURER
- TPM_PT_VENDOR_STRING_1
- TPM_PT_VENDOR_STRING_2 (NOTE 4)
- TPM_PT_VENDOR_STRING_3 (NOTE 4)
- TPM_PT_VENDOR_STRING_4 (NOTE 4)
- TPM_PT_VENDOR_TPM_TYPE
- TPM_PT_FIRMWARE_VERSION_1
- TPM_PT_FIRMWARE_VERSION_2

NOTE 4 If the vendor string does not require one of these values, the property type does not need to exist.

A vendor may optionally allow the TPM to return other values.

If in Failure mode and a capability is requested that is not available in Failure mode, the TPM shall return no value.

EXAMPLE 3 Assume the TPM is in Failure mode and the TPM only supports reporting of the minimum required set of properties (the limited subset of TPML_TAGGED_TPM_PROPERTY values). If a TPM2_GetCapability is received requesting a capability that has a property type value greater than TPM_PT_FIRMWARE_VERSION_2, the TPM may return a zero length list with the moreData parameter set to NO or return the property TPM_PT_FIRMWARE_VERSION_2. If the property type is less than TPM_PT_MANUFACTURER, the TPM will return properties beginning with TPM_PT_MANUFACTURER.

In Failure mode, *tag* is required to be TPM_ST_NO_SESSIONS or the TPM shall return TPM_RC_FAILURE.

The capability categories and the types of the return values are:

capability	***property***	**Return Type**
TPM_CAP_ALGS	TPM_ALG_ID(1)	TPML_ALG_PROPERTY
TPM_CAP_HANDLES	TPM_HANDLE	TPML_HANDLE
TPM_CAP_COMMANDS	TPM_CC	TPML_CCA
TPM_CAP_PP_COMMANDS	TPM_CC	TPML_CC
TPM_CAP_AUDIT_COMMANDS	TPM_CC	TPML_CC
TPM_CAP_PCRS	Reserved	TPML_PCR_SELECTION
TPM_CAP_TPM_PROPERTIES	TPM_PT	TPML_TAGGED_TPM_PROPERTY
TPM_CAP_PCR_PROPERTIES	TPM_PT_PCR	TPML_TAGGED_PCR_PROPERTY
TPM_CAP_ECC_CURVES	TPM_ECC_CURVE(1)	TPML_ECC_CURVE
TPM_CAP_AUTH_POLICIES (3)	TPM_HANDLE(2)	TPML_TAGGED_POLICY
TPM_CAP_ACT(4)	TPM_HANDLE(2)	TPML_ACT_DATA
TPM_CAP_VENDOR_PROPERTY	manufacturer specific	manufacturer-specific values

NOTES:
(1) The TPM_ALG_ID or TPM_ECC_CURVE is cast to a UINT32
(2) The TPM will return TPM_RC_VALUE if the handle does not reference the range for permanent handles.
(3) TPM_CAP_AUTH_POLICIES was added in revision 01.32.
(4) TPM_CAP_ACT was added in revision 01.56.

- TPM_CAP_ALGS – Returns a list of TPMS_ALG_PROPERTIES. Each entry is an algorithm ID and a set of properties of the algorithm.
- TPM_CAP_HANDLES – Returns a list of all of the handles within the handle range of the *property* parameter. The range of the returned handles is determined by the handle type (the most-significant octet (MSO) of the *property)*. Any of the defined handle types is allowed

  EXAMPLE 4 If the MSO of *property* is TPM_HT_NV_INDEX, then the TPM will return a list of NV Index values.

  EXAMPLE 5 If the MSO of property is TPM_HT_PCR, then the TPM will return a list of PCR.

- For this capability, use of TPM_HT_LOADED_SESSION and TPM_HT_SAVED_SESSION is allowed. Requesting handles with a handle type of TPM_HT_LOADED_SESSION will return handles for loaded sessions. The returned handle values will have a handle type of either TPM_HT_HMAC_SESSION or TPM_HT_POLICY_SESSION. If saved sessions are requested, all returned values will have the TPM_HT_HMAC_SESSION handle type because the TPM does not track the session type of saved sessions.

  NOTE 5 TPM_HT_LOADED_SESSION and TPM_HT_HMAC_SESSION have the same value, as do TPM_HT_SAVED_SESSION and TPM_HT_POLICY_SESSION. It is not possible to request that the TPM return a list of loaded HMAC sessions without including the policy sessions.

- TPM_CAP_COMMANDS – Returns a list of the command attributes for all of the commands implemented in the TPM, starting with the TPM_CC indicated by the *property* parameter. If vendor specific commands are implemented, the vendor-specific command attribute with the lowest *commandIndex*, is returned after the non-vendor-specific (base) command.

  NOTE 6 The type of the property parameter is a TPM_CC while the type of the returned list is TPML_CCA.

- TPM_CAP_PP_COMMANDS – Returns a list of all of the commands currently requiring Physical Presence for confirmation of platform authorization. The list will start with the TPM_CC indicated by *property*.
- TPM_CAP_AUDIT_COMMANDS – Returns a list of all of the commands currently set for command audit.
- TPM_CAP_PCRS – Returns the current allocation of PCR in a TPML_PCR_SELECTION. The *property* parameter shall be zero. The TPM will always respond to this command with the full PCR allocation and *moreData* will be NO.

  The TPML_PCR_SELECTION must include a TPMS_PCR_SELECTION for each PCR bank in which there is at least one allocated PCR. The TPML_PCR_SELECTION may return a TPMS_PCR_SELECTION for each implemented PCR bank. The TPML_PCR_SELECTION may return a TPMS_PCR_SELECTION for each implemented hash algorithm.

- TPM_CAP_TPM_PROPERTIES – Returns a list of tagged properties. The tag is a TPM_PT and the property is a 32-bit value. The properties are returned in groups. Each property group is on a 256-value boundary (that is, the boundary occurs when the TPM_PT is evenly divisible by 256). The TPM will only return values in the same group as the *property* parameter in the command.
- TPM_CAP_PCR_PROPERTIES – Returns a list of tagged PCR properties. The tag is a TPM_PT_PCR and the property is a TPMS_PCR_SELECT.

  The input command property is a TPM_PT_PCR (see TPM 2.0 Part 2 for PCR properties to be requested) that specifies the first property to be returned. If propertyCount is greater than 1, the list of properties begins with that property and proceeds in TPM_PT_PCR sequence.

  Each item in the list is a TPMS_PCR_SELECT structure that contains a bitmap of all PCR.

  NOTE 7 A PCR index in all banks (all hash algorithms) has the same properties, so the hash algorithm is not specified here.

- TPM_CAP_TPM_ECC_CURVES – Returns a list of ECC curve identifiers currently available for use in the TPM.
- TPM_CAP_AUTH_POLICIES - Returns a list of tagged policies reporting the authorization policies for the permanent handles.
- TPM_CAP_ACT – Returns a list of TPMS_ACT_DATA, each of which contains the handle for the ACT, the remaining time before it expires, and the ACT attributes.

The *moreData* parameter will have a value of YES if there are more values of the requested type that were not returned.

If no next capability exists, the TPM will return a zero-length list and *moreData* will have a value of NO.

30.2.2 Command and Response

Table 208 — TPM2_GetCapability Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_GetCapability
TPM_CAP	capability	group selection; determines the format of the response
UINT32	property	further definition of information
UINT32	propertyCount	number of properties of the indicated type to return

Table 209 — TPM2_GetCapability Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPMI_YES_NO	moreData	flag to indicate if there are more values of this type
TPMS_CAPABILITY_DATA	capabilityData	the capability data

30.2.3 Detailed Actions

```
#include "Tpm.h"
#include "GetCapability_fp.h"
#if CC_GetCapability  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_HANDLE	value of *property* is in an unsupported handle range for the TPM_CAP_HANDLES *capability* value
TPM_RC_VALUE	invalid *capability*; or *property* is not 0 for the TPM_CAP_PCRS *capability* value

```
TPM_RC
TPM2_GetCapability(
    GetCapability_In    *in,            // IN: input parameter list
    GetCapability_Out   *out            // OUT: output parameter list
    )
{
    TPMU_CAPABILITIES   *data = &out->capabilityData.data;
// Command Output

    // Set output capability type the same as input type
    out->capabilityData.capability = in->capability;

    switch(in->capability)
    {
        case TPM_CAP_ALGS:
            out->moreData = AlgorithmCapGetImplemented((TPM_ALG_ID)in->property,
                                                       in->propertyCount,
                                                       &data->algorithms);
            break;
        case TPM_CAP_HANDLES:
            switch(HandleGetType((TPM_HANDLE)in->property))
            {
                case TPM_HT_TRANSIENT:
                    // Get list of handles of loaded transient objects
                    out->moreData = ObjectCapGetLoaded((TPM_HANDLE)in->property,
                                                       in->propertyCount,
                                                       &data->handles);
                    break;
                case TPM_HT_PERSISTENT:
                    // Get list of handles of persistent objects
                    out->moreData = NvCapGetPersistent((TPM_HANDLE)in->property,
                                                       in->propertyCount,
                                                       &data->handles);
                    break;
                case TPM_HT_NV_INDEX:
                    // Get list of defined NV index
                    out->moreData = NvCapGetIndex((TPM_HANDLE)in->property,
                                                  in->propertyCount,
                                                  &data->handles);
                    break;
                case TPM_HT_LOADED_SESSION:
                    // Get list of handles of loaded sessions
                    out->moreData = SessionCapGetLoaded((TPM_HANDLE)in->property,
                                                        in->propertyCount,
                                                        &data->handles);
                    break;
#ifdef TPM_HT_SAVED_SESSION
                case TPM_HT_SAVED_SESSION:
#else
```

```
                case TPM_HT_ACTIVE_SESSION:
#endif
            // Get list of handles of
                    out->moreData = SessionCapGetSaved((TPM_HANDLE)in->property,
                                                       in->propertyCount,
                                                       &data->handles);
                    break;
                case TPM_HT_PCR:
                    // Get list of handles of PCR
                    out->moreData = PCRCapGetHandles((TPM_HANDLE)in->property,
                                                     in->propertyCount,
                                                     &data->handles);
                    break;
                case TPM_HT_PERMANENT:
                    // Get list of permanent handles
                    out->moreData = PermanentCapGetHandles((TPM_HANDLE)in->property,
                                                           in->propertyCount,
                                                           &data->handles);
                    break;
                default:
                    // Unsupported input handle type
                    return TPM_RCS_HANDLE + RC_GetCapability_property;
                    break;
            }
            break;
        case TPM_CAP_COMMANDS:
            out->moreData = CommandCapGetCCList((TPM_CC)in->property,
                                                in->propertyCount,
                                                &data->command);
            break;
        case TPM_CAP_PP_COMMANDS:
            out->moreData = PhysicalPresenceCapGetCCList((TPM_CC)in->property,
                                                         in->propertyCount,
                                                         &data->ppCommands);
            break;
        case TPM_CAP_AUDIT_COMMANDS:
            out->moreData = CommandAuditCapGetCCList((TPM_CC)in->property,
                                                     in->propertyCount,
                                                     &data->auditCommands);
            break;
        case TPM_CAP_PCRS:
            // Input property must be 0
            if(in->property != 0)
                return TPM_RCS_VALUE + RC_GetCapability_property;
            out->moreData = PCRCapGetAllocation(in->propertyCount,
                                                &data->assignedPCR);
            break;
        case TPM_CAP_PCR_PROPERTIES:
            out->moreData = PCRCapGetProperties((TPM_PT_PCR)in->property,
                                                in->propertyCount,
                                                &data->pcrProperties);
            break;
        case TPM_CAP_TPM_PROPERTIES:
            out->moreData = TPMCapGetProperties((TPM_PT)in->property,
                                                in->propertyCount,
                                                &data->tpmProperties);
            break;
#if ALG_ECC
        case TPM_CAP_ECC_CURVES:
            out->moreData = CryptCapGetECCCurve((TPM_ECC_CURVE)in->property,
                                                in->propertyCount,
                                                &data->eccCurves);
            break;
#endif // ALG_ECC
        case TPM_CAP_AUTH_POLICIES:
            if(HandleGetType((TPM_HANDLE)in->property) != TPM_HT_PERMANENT)
```

```
                return TPM_RCS_VALUE + RC_GetCapability_property;
            out->moreData = PermanentHandleGetPolicy((TPM_HANDLE)in->property,
                                                     in->propertyCount,
                                                     &data->authPolicies);
            break;
        case TPM_CAP_ACT:
            if(((TPM_RH)in->property < TPM_RH_ACT_0)
               || ((TPM_RH)in->property > TPM_RH_ACT_F))
                return TPM_RCS_VALUE + RC_GetCapability_property;
            out->moreData = ActGetCapabilityData((TPM_HANDLE)in->property,
                                                 in->propertyCount,
                                                 &data->actData);
            break;
        case TPM_CAP_VENDOR_PROPERTY:
            // vendor property is not implemented
        default:
            // Unsupported TPM_CAP value
            return TPM_RCS_VALUE + RC_GetCapability_capability;
            break;
    }

    return TPM_RC_SUCCESS;
}
#endif // CC_GetCapability
```

30.3 TPM2_TestParms

30.3.1 General Description

This command is used to check to see if specific combinations of algorithm parameters are supported.

The TPM will unmarshal the provided TPMT_PUBLIC_PARMS. If the parameters unmarshal correctly, then the TPM will return TPM_RC_SUCCESS, indicating that the parameters are valid for the TPM. The TPM will return the appropriate unmarshaling error if a parameter is not valid.

30.3.2 Command and Response

Table 210 — TPM2_TestParms Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_TestParms
TPMT_PUBLIC_PARMS	parameters	algorithm parameters to be validated

Table 211 — TPM2_TestParms Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	TPM_RC

30.3.3 Detailed Actions

```
#include "Tpm.h"
#include "TestParms_fp.h"
#if CC_TestParms  // Conditional expansion of this file
TPM_RC
TPM2_TestParms(
    TestParms_In    *in             // IN: input parameter list
    )
{
    // Input parameter is not reference in command action
    NOT_REFERENCED(in);

    // The parameters are tested at unmarshal process.  We do nothing in command
    // action
    return TPM_RC_SUCCESS;
}
#endif // CC_TestParms
```

31 Non-volatile Storage

31.1 Introduction

The NV commands are used to create, update, read, and delete allocations of space in NV memory. Before an Index may be used, it must be defined (TPM2_NV_DefineSpace()).

An Index may be modified if the proper write authorization is provided or read if the proper read authorization is provided. Different controls are available for reading and writing.

An Index may have an Index-specific *authValue* and *authPolicy*. The *authValue* may be used to authorize reading if TPMA_NV_AUTHREAD is SET and writing if TPMA_NV_AUTHWRITE is SET. The *authPolicy* may be used to authorize reading if TPMA_NV_POLICYREAD is SET and writing if TPMA_NV_POLICYWRITE is SET.

For commands that have both *authHandle* and *nvIndex* parameters, *authHandle* can be an NV Index, Platform Authorization, or Owner Authorization. If *authHandle* is an NV Index, it must be the same as *nvIndex* (TPM_RC_NV_AUTHORIZATION).

TPMA_NV_PPREAD and TPMA_NV_PPWRITE indicate if reading or writing of the NV Index may be authorized by *platformAuth* or *platformPolicy*.

TPMA_NV_OWNERREAD and TPMA_NV_OWNERWRITE indicate if reading or writing of the NV Index may be authorized by *ownerAuth* or *ownerPolicy*.

If an operation on an NV index requires authorization, and the authHandle parameter is the handle of an NV Index, then the nvIndex parameter must have the same value or the TPM will return TPM_RC_NV_AUTHORIZATION.

NOTE 1 This check ensures that the authorization that was provided is associated with the NV Index being authorized.

For creating an Index, Owner Authorization may not be used if *shEnable* is CLEAR and Platform Authorization may not be used if *phEnableNV* is CLEAR.

If an Index was defined using Platform Authorization, then that Index is not accessible when *phEnableNV* is CLEAR. If an Index was defined using Owner Authorization, then that Index is not accessible when *shEnable* is CLEAR.

For read access control, any combination of TPMA_NV_PPREAD, TPMA_NV_OWNERREAD, TPMA_NV_AUTHREAD, or TPMA_NV_POLICYREAD is allowed as long as at least one is SET.

For write access control, any combination of TPMA_NV_PPWRITE, TPMA_NV_OWNERWRITE, TPMA_NV_AUTHWRITE, or TPMA_NV_POLICYWRITE is allowed as long as at least one is SET.

If an Index has been defined and not written, then any operation on the NV Index that requires read authorization will fail (TPM_RC_NV_INITIALIZED). This check may be made before or after other authorization checks but shall be performed before checking the NV Index *authValue*. An authorization failure due to the NV Index not having been written shall not be logged by the dictionary attack logic.

If TPMA_NV_CLEAR_STCLEAR is SET, then the TPMA_NV_WRITTEN will be CLEAR on each TPM2_Startup(TPM_SU_CLEAR). TPMA_NV_CLEAR_STCLEAR shall not be SET if the *nvIndexType* is TPM_NT_COUNTER.

The code in the “Detailed Actions” clause of each command is written to interface with an implementation-dependent library that allows access to NV memory. The actions assume no specific layout of the structure of the NV data.

Only one NV Index may be directly referenced in a command.

NOTE 2 This means that, if *authHandle* references an NV Index, then *nvIndex* will have the same value. However, this does not limit the number of changes that may occur as side effects. For example, any number of NV Indexes might be relocated as a result of deleting or adding a NV Index.

31.2 NV Counters

When an Index has the TPM_NT_COUNTER attribute, it behaves as a monotonic counter and may only be updated using TPM2_NV_Increment().

When an NV counter is created, the TPM shall initialize the 8-octet counter value with a number that is greater than any count value for any NV counter on the TPM since the time of TPM manufacture.

An NV counter may be defined with the TPMA_NV_ORDERLY attribute to indicate that the NV Index is expected to be modified at a high frequency and that the data is only required to persist when the TPM goes through an orderly shutdown process. The TPM may update the counter value in RAM and occasionally update the non-volatile version of the counter. An orderly shutdown is one occasion to update the non-volatile count. If the difference between the volatile and non-volatile version of the counter becomes as large as MAX_ORDERLY_COUNT, this shall be another occasion for updating the non-volatile count.

Before an NV counter can be used, the TPM shall validate that the count is not less than a previously reported value. If the TPMA_NV_ORDERLY attribute is not SET, or if the TPM experienced an orderly shutdown, then the count is assumed to be correct. If the TPMA_NV_ORDERLY attribute is SET, and the TPM shutdown was not orderly, then the TPM shall OR MAX_ORDERLY_COUNT to the contents of the non-volatile counter and set that as the current count.

NOTE 1 Because the TPM would have updated the NV Index if the difference between the count values was equal to MAX_ORDERLY_COUNT + 1, the highest value that could have been in the NV Index is MAX_ORDERLY_COUNT so it is safe to restore that value.

NOTE 2 The TPM may implement the RAM portion of the counter such that the effective value of the NV counter is the sum of both the volatile and non-volatile parts. If so, then the TPM may initialize the RAM version of the counter to MAX_ORDERLY_COUNT and no update of NV is necessary.

NOTE 3 When a new NV counter is created, the TPM may search all the counters to determine which has the highest value. In this search, the TPM would use the sum of the non-volatile and RAM portions of the counter. The RAM portion of the counter shall be properly initialized to reflect shutdown process (orderly or not) of the TPM.

31.3 TPM2_NV_DefineSpace

31.3.1 General Description

This command defines the attributes of an NV Index and causes the TPM to reserve space to hold the data associated with the NV Index. If a definition already exists at the NV Index, the TPM will return TPM_RC_NV_DEFINED.

The TPM will return TPM_RC_ATTRIBUTES if *nvIndexType* has a reserved value in *publicInfo.*

NOTE 1 It is not required that any of these three attributes be set.

The TPM shall return TPM_RC_ATTRIBUTES if TPMA_NV_WRITTEN, TPMA_NV_READLOCKED, or TPMA_NV_WRITELOCKED is SET.

If *nvIndexType* is TPM_NT_COUNTER, TPM_NT_BITS, TPM_NT_PIN_FAIL, or TPM_NT_PIN_PASS, then *publicInfo→dataSize* shall be set to eight (8) or the TPM shall return TPM_RC_SIZE.

If *nvIndexType* is TPM_NT_EXTEND, then *publicInfo→dataSize* shall match the digest size of the *publicInfo.nameAlg* or the TPM shall return TPM_RC_SIZE.

NOTE 2 TPM_RC_ATTRIBUTES could be returned by a TPM that is based on the reference code of older versions of the specification but the correct response for this error is TPM_RC_SIZE.

If the NV Index is an ordinary Index and *publicInfo→dataSize* is larger than supported by the TPM implementation then the TPM shall return TPM_RC_SIZE.

NOTE 3 The limit for the data size may vary according to the type of the index. For example, if the index has TPMA_NV_ORDERLY SET, then the maximum size of an ordinary NV Index may be less than the size of an ordinary NV Index that has TPMA_NV_ORDERLY CLEAR.

At least one of TPMA_NV_PPREAD, TPMA_NV_OWNERREAD, TPMA_NV_AUTHREAD, or TPMA_NV_POLICYREAD shall be SET or the TPM shall return TPM_RC_ATTRIBUTES.

At least one of TPMA_NV_PPWRITE, TPMA_NV_OWNERWRITE, TPMA_NV_AUTHWRITE, or TPMA_NV_POLICYWRITE shall be SET or the TPM shall return TPM_RC_ATTRIBUTES.

If TPMA_NV_CLEAR_STCLEAR is SET, then *nvIndexType* shall not be TPM_NT_COUNTER or the TPM shall return TPM_RC_ATTRIBUTES.

If *platformAuth/platformPolicy* is used for authorization, then TPMA_NV_PLATFORMCREATE shall be SET in *publicInfo*. If *ownerAuth/ownerPolicy* is used for authorization, TPMA_NV_PLATFORMCREATE shall be CLEAR in *publicInfo.* If TPMA_NV_PLATFORMCREATE is not set correctly for the authorization, the TPM shall return TPM_RC_ATTRIBUTES.

If TPMA_NV_POLICY_DELETE is SET, then the authorization shall be with Platform Authorization or the TPM shall return TPM_RC_ATTRIBUTES.

If *nvIndexType* is TPM_NT_PIN_FAIL, then TPMA_NV_NO_DA shall be SET. Otherwise, the TPM shall return TPM_RC_ATTRIBUTES.

NOTE 4 The intent of a PIN Fail index is that its DA protection is on a per-index basis, not based on the global DA protection. This avoids conflict over which type of dictionary attack protection is in use.

If *nvIndexType* is TPM_NT_PIN_FAIL or TPM_NT_PIN_PASS, then at least one of TPMA_NV_PPWRITE, TPMA_NV_OWNERWRITE, or TPMA_NV_POLICYWRITE shall be SET or the TPM shall return TPM_RC_ATTRIBUTES. TPMA_NV_AUTHWRITE shall be CLEAR. Otherwise, the TPM shall return TPM_RC_ATTRIBUTES.

NOTE 5 If TPMA_NV_AUTHWRITE was SET for a PIN Pass index, a user knowing the authorization value could decrease pinCount or increase pinLimit, defeating the purpose of a PIN Pass index. The requirement is also enforced for a PIN Fail index for consistency.

If the implementation does not support TPM2_NV_Increment(), the TPM shall return TPM_RC_ATTRIBUTES if *nvIndexType* is TPM_NT_COUNTER.

If the implementation does not support TPM2_NV_SetBits(), the TPM shall return TPM_RC_ATTRIBUTES if *nvIndexType* is TPM_NT_BITS.

If the implementation does not support TPM2_NV_Extend(), the TPM shall return TPM_RC_ATTRIBUTES if *nvIndexType* is TPM_NT_EXTEND.

If the implementation does not support TPM2_NV_UndefineSpaceSpecial(), the TPM shall return TPM_RC_ATTRIBUTES if TPMA_NV_POLICY_DELETE is SET.

After the successful completion of this command, the NV Index exists but TPMA_NV_WRITTEN will be CLEAR. Any access of the NV data will return TPM_RC_NV_UNINITIALIZED.

In some implementations, an NV Index with the TPM_NT_COUNTER attribute may require special TPM resources that provide higher endurance than regular NV. For those implementations, if this command fails because of lack of resources, the TPM will return TPM_RC_NV_SPACE.

The value of *auth* is saved in the created structure. The size of *auth* is limited to be no larger than the size of the digest produced by the NV Index's *nameAlg* (TPM_RC_SIZE).

31.3.2 Command and Response

Table 212 — TPM2_NV_DefineSpace Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_NV_DefineSpace {NV}
TPMI_RH_PROVISION	@authHandle	TPM_RH_OWNER or TPM_RH_PLATFORM+{PP} Auth Index: 1 Auth Role: USER
TPM2B_AUTH	auth	the authorization value
TPM2B_NV_PUBLIC	publicInfo	the public parameters of the NV area

Table 213 — TPM2_NV_DefineSpace Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

31.3.3 Detailed Actions

```
#include "Tpm.h"
#include "NV_DefineSpace_fp.h"
#if CC_NV_DefineSpace  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_HIERARCHY	for authorizations using TPM_RH_PLATFORM *phEnable_NV* is clear preventing access to NV data in the platform hierarchy.
TPM_RC_ATTRIBUTES	attributes of the index are not consistent
TPM_RC_NV_DEFINED	index already exists
TPM_RC_NV_SPACE	insufficient space for the index
TPM_RC_SIZE	'auth->size' or '*publicInfo->authPolicy*.size' is larger than the digest size of '*publicInfo->nameAlg*'; or '*publicInfo->dataSize*' is not consistent with '*publicInfo*->attributes' (this includes the case when the index is larger than a MAX_NV_BUFFER_SIZE but the TPMA_NV_WRITEALL attribute is SET)

```
TPM_RC
TPM2_NV_DefineSpace(
    NV_DefineSpace_In   *in             // IN: input parameter list
    )
{
    TPMA_NV         attributes = in->publicInfo.nvPublic.attributes;
    UINT16          nameSize;

    nameSize = CryptHashGetDigestSize(in->publicInfo.nvPublic.nameAlg);

// Input Validation

    // Checks not specific to type

    // If the UndefineSpaceSpecial command is not implemented, then can't have
    // an index that can only be deleted with policy
#if CC_NV_UndefineSpaceSpecial == NO
    if(IS_ATTRIBUTE(attributes, TPMA_NV, POLICY_DELETE))
        return TPM_RCS_ATTRIBUTES + RC_NV_DefineSpace_publicInfo;
#endif

    // check that the authPolicy consistent with hash algorithm

    if(in->publicInfo.nvPublic.authPolicy.t.size != 0
       && in->publicInfo.nvPublic.authPolicy.t.size != nameSize)
        return TPM_RCS_SIZE + RC_NV_DefineSpace_publicInfo;

   // make sure that the authValue is not too large
    if(MemoryRemoveTrailingZeros(&in->auth)
       > CryptHashGetDigestSize(in->publicInfo.nvPublic.nameAlg))
        return TPM_RCS_SIZE + RC_NV_DefineSpace_auth;

    // If an index is being created by the owner and shEnable is
    // clear, then we would not reach this point because ownerAuth
    // can't be given when shEnable is CLEAR. However, if phEnable
    // is SET but phEnableNV is CLEAR, we have to check here
    if(in->authHandle == TPM_RH_PLATFORM && gc.phEnableNV == CLEAR)
        return TPM_RCS_HIERARCHY + RC_NV_DefineSpace_authHandle;

    // Attribute checks
    // Eliminate the unsupported types
```



```
    switch(GET_TPM_NT(attributes))
    {
#if CC_NV_Increment == YES
        case TPM_NT_COUNTER:
#endif
#if CC_NV_SetBits == YES
        case TPM_NT_BITS:
#endif
#if CC_NV_Extend == YES
        case TPM_NT_EXTEND:
#endif
#if CC_PolicySecret == YES && defined TPM_NT_PIN_PASS
        case TPM_NT_PIN_PASS:
        case TPM_NT_PIN_FAIL:
#endif
        case TPM_NT_ORDINARY:
            break;
        default:
            return TPM_RCS_ATTRIBUTES + RC_NV_DefineSpace_publicInfo;
            break;
    }
    // Check that the sizes are OK based on the type
    switch(GET_TPM_NT(attributes))
    {
        case TPM_NT_ORDINARY:
            // Can't exceed the allowed size for the implementation
            if(in->publicInfo.nvPublic.dataSize > MAX_NV_INDEX_SIZE)
                return TPM_RCS_SIZE + RC_NV_DefineSpace_publicInfo;
            break;
        case TPM_NT_EXTEND:
            if(in->publicInfo.nvPublic.dataSize != nameSize)
                return TPM_RCS_SIZE + RC_NV_DefineSpace_publicInfo;
            break;
        default:
            // Everything else needs a size of 8
            if(in->publicInfo.nvPublic.dataSize != 8)
                return TPM_RCS_SIZE + RC_NV_DefineSpace_publicInfo;
            break;
    }
    // Handle other specifics
    switch(GET_TPM_NT(attributes))
    {
        case TPM_NT_COUNTER:
            // Counter can't have TPMA_NV_CLEAR_STCLEAR SET (don't clear counters)
            if(IS_ATTRIBUTE(attributes, TPMA_NV, CLEAR_STCLEAR))
                return TPM_RCS_ATTRIBUTES + RC_NV_DefineSpace_publicInfo;
            break;
#ifdef TPM_NT_PIN_FAIL
        case TPM_NT_PIN_FAIL:
            // NV_NO_DA must be SET and AUTHWRITE must be CLEAR
            // NOTE: As with a PIN_PASS index, the authValue of the index is not
            // available until the index is written. If AUTHWRITE is the only way to
            // write then index, it could never be written. Rather than go through
            // all of the other possible ways to write the Index, it is simply
            // prohibited to write the index with the authValue. Other checks
            // below will insure that there seems to be a way to write the index
            // (i.e., with platform authorization , owner authorization,
            // or with policyAuth.)
            // It is not allowed to create a PIN Index that can't be modified.
            if(!IS_ATTRIBUTE(attributes, TPMA_NV, NO_DA))
                return TPM_RCS_ATTRIBUTES + RC_NV_DefineSpace_publicInfo;
#endif
#ifdef TPM_NT_PIN_PASS
        case TPM_NT_PIN_PASS:
            // AUTHWRITE must be CLEAR (see note above to TPM_NT_PIN_FAIL)
            if(IS_ATTRIBUTE(attributes, TPMA_NV, AUTHWRITE)
```

```
               || IS_ATTRIBUTE(attributes, TPMA_NV, GLOBALLOCK)
               || IS_ATTRIBUTE(attributes, TPMA_NV, WRITEDEFINE))
                return TPM_RCS_ATTRIBUTES + RC_NV_DefineSpace_publicInfo;
#endif  // this comes before break because PIN_FAIL falls through
            break;
        default:
            break;
    }

    // Locks may not be SET and written cannot be SET
    if(IS_ATTRIBUTE(attributes, TPMA_NV, WRITTEN)
       || IS_ATTRIBUTE(attributes, TPMA_NV, WRITELOCKED)
       || IS_ATTRIBUTE(attributes, TPMA_NV, READLOCKED))
        return TPM_RCS_ATTRIBUTES + RC_NV_DefineSpace_publicInfo;

    // There must be a way to read the index.
    if(!IS_ATTRIBUTE(attributes, TPMA_NV, OWNERREAD)
       && !IS_ATTRIBUTE(attributes, TPMA_NV, PPREAD)
       && !IS_ATTRIBUTE(attributes, TPMA_NV, AUTHREAD)
       && !IS_ATTRIBUTE(attributes, TPMA_NV, POLICYREAD))
        return TPM_RCS_ATTRIBUTES + RC_NV_DefineSpace_publicInfo;

    // There must be a way to write the index
    if(!IS_ATTRIBUTE(attributes, TPMA_NV, OWNERWRITE)
       && !IS_ATTRIBUTE(attributes, TPMA_NV, PPWRITE)
       && !IS_ATTRIBUTE(attributes, TPMA_NV, AUTHWRITE)
       && !IS_ATTRIBUTE(attributes, TPMA_NV, POLICYWRITE))
        return TPM_RCS_ATTRIBUTES + RC_NV_DefineSpace_publicInfo;

    // An index with TPMA_NV_CLEAR_STCLEAR can't have TPMA_NV_WRITEDEFINE SET
    if(IS_ATTRIBUTE(attributes, TPMA_NV, CLEAR_STCLEAR)
       &&  IS_ATTRIBUTE(attributes, TPMA_NV, WRITEDEFINE))
        return TPM_RCS_ATTRIBUTES + RC_NV_DefineSpace_publicInfo;

    // Make sure that the creator of the index can delete the index
    if((IS_ATTRIBUTE(attributes, TPMA_NV, PLATFORMCREATE)
        && in->authHandle == TPM_RH_OWNER)
       || (!IS_ATTRIBUTE(attributes, TPMA_NV, PLATFORMCREATE)
           && in->authHandle == TPM_RH_PLATFORM))
        return TPM_RCS_ATTRIBUTES + RC_NV_DefineSpace_authHandle;

    // If TPMA_NV_POLICY_DELETE is SET, then the index must be defined by
    // the platform
    if(IS_ATTRIBUTE(attributes, TPMA_NV, POLICY_DELETE)
       &&  TPM_RH_PLATFORM != in->authHandle)
        return TPM_RCS_ATTRIBUTES + RC_NV_DefineSpace_publicInfo;

    // Make sure that the TPMA_NV_WRITEALL is not set if the index size is larger
    // than the allowed NV buffer size.
    if(in->publicInfo.nvPublic.dataSize > MAX_NV_BUFFER_SIZE
       &&  IS_ATTRIBUTE(attributes, TPMA_NV, WRITEALL))
        return TPM_RCS_SIZE + RC_NV_DefineSpace_publicInfo;

    // And finally, see if the index is already defined.
    if(NvIndexIsDefined(in->publicInfo.nvPublic.nvIndex))
        return TPM_RC_NV_DEFINED;

// Internal Data Update
    // define the space.  A TPM_RC_NV_SPACE error may be returned at this point
    return NvDefineIndex(&in->publicInfo.nvPublic, &in->auth);
}
#endif // CC_NV_DefineSpace
```


31.4 TPM2_NV_UndefineSpace

31.4.1 General Description

This command removes an Index from the TPM.

If *nvIndex* is not defined, the TPM shall return TPM_RC_HANDLE.

If *nvIndex* references an Index that has its TPMA_NV_PLATFORMCREATE attribute SET, the TPM shall return TPM_RC_NV_AUTHORIZATION unless Platform Authorization is provided.

If *nvIndex* references an Index that has its TPMA_NV_POLICY_DELETE attribute SET, the TPM shall return TPM_RC_ATTRIBUTES.

NOTE An Index with TPMA_NV_PLATFORMCREATE CLEAR may be deleted with Platform Authorization as long as shEnable is SET. If shEnable is CLEAR, indexes created using Owner Authorization are not accessible even for deletion by the platform.

31.4.2 Command and Response

Table 214 — TPM2_NV_UndefineSpace Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_NV_UndefineSpace {NV}
TPMI_RH_PROVISION	@authHandle	TPM_RH_OWNER or TPM_RH_PLATFORM+{PP} Auth Index: 1 Auth Role: USER
TPMI_RH_NV_INDEX	nvIndex	the NV Index to remove from NV space Auth Index: None

Table 215 — TPM2_NV_UndefineSpace Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

31.4.3 Detailed Actions

```
#include "Tpm.h"
#include "NV_UndefineSpace_fp.h"
#if CC_NV_UndefineSpace  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_ATTRIBUTES	TPMA_NV_POLICY_DELETE is SET in the Index referenced by *nvIndex* so this command may not be used to delete this Index (see TPM2_NV_UndefineSpaceSpecial())
TPM_RC_NV_AUTHORIZATION	attempt to use *ownerAuth* to delete an index created by the platform

```
TPM_RC
TPM2_NV_UndefineSpace(
    NV_UndefineSpace_In     *in             // IN: input parameter list
    )
{
    NV_REF           locator;
    NV_INDEX        *nvIndex = NvGetIndexInfo(in->nvIndex, &locator);

// Input Validation
    // This command can't be used to delete an index with TPMA_NV_POLICY_DELETE SET
    if(IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, POLICY_DELETE))
        return TPM_RCS_ATTRIBUTES + RC_NV_UndefineSpace_nvIndex;

    // The owner may only delete an index that was defined with ownerAuth. The
    // platform may delete an index that was created with either authorization.
    if(in->authHandle == TPM_RH_OWNER
       && IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, PLATFORMCREATE))
        return TPM_RC_NV_AUTHORIZATION;

// Internal Data Update

    // Call implementation dependent internal routine to delete NV index
    return NvDeleteIndex(nvIndex, locator);
}
#endif // CC_NV_UndefineSpace
```

31.5 TPM2_NV_UndefineSpaceSpecial

31.5.1 General Description

This command allows removal of a platform-created NV Index that has TPMA_NV_POLICY_DELETE SET.

This command requires that the policy of the NV Index be satisfied before the NV Index may be deleted. Because administrative role is required, the policy must contain a command that sets the policy command code to TPM_CC_NV_UndefineSpaceSpecial. This indicates that the policy that is being used is a policy that is for this command, and not a policy that would approve another use. That is, authority to use an entity does not grant authority to undefine the entity.

Since the index is deleted, the Empty Buffer is used as the authValue when generating the response HMAC.

If *nvIndex* is not defined, the TPM shall return TPM_RC_HANDLE.

If *nvIndex* references an Index that has its TPMA_NV_PLATFORMCREATE or TPMA_NV_POLICY_DELETE attribute CLEAR, the TPM shall return TPM_RC_ATTRIBUTES.

NOTE An Index with TPMA_NV_PLATFORMCREATE CLEAR may be deleted with TPM2_UndefineSpace()as long as shEnable is SET. If shEnable is CLEAR, indexes created using Owner Authorization are not accessible even for deletion by the platform.

31.5.2 Command and Response

Table 216 — TPM2_NV_UndefineSpaceSpecial Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_NV_UndefineSpaceSpecial {NV}
TPMI_RH_NV_INDEX	@nvIndex	Index to be deleted Auth Index: 1 Auth Role: ADMIN
TPMI_RH_PLATFORM	@platform	TPM_RH_PLATFORM + {PP} Auth Index: 2 Auth Role: USER

Table 217 — TPM2_NV_UndefineSpaceSpecial Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

31.5.3 Detailed Actions

```
#include "Tpm.h"
#include "NV_UndefineSpaceSpecial_fp.h"
#include "SessionProcess_fp.h"
#if CC_NV_UndefineSpaceSpecial  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_ATTRIBUTES	TPMA_NV_POLICY_DELETE is not SET in the Index referenced by *nvIndex*

```
TPM_RC
TPM2_NV_UndefineSpaceSpecial(
    NV_UndefineSpaceSpecial_In  *in             // IN: input parameter list
    )
{
    TPM_RC          result;
    NV_REF          locator;
    NV_INDEX       *nvIndex = NvGetIndexInfo(in->nvIndex, &locator);
// Input Validation
    // This operation only applies when the TPMA_NV_POLICY_DELETE attribute is SET
    if(!IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, POLICY_DELETE))
        return TPM_RCS_ATTRIBUTES + RC_NV_UndefineSpaceSpecial_nvIndex;
// Internal Data Update
    // Call implementation dependent internal routine to delete NV index
    result = NvDeleteIndex(nvIndex, locator);

    // If we just removed the index providing the authorization, make sure that the
    // authorization session computation is modified so that it doesn't try to
    // access the authValue of the just deleted index
    if(result == TPM_RC_SUCCESS)
        SessionRemoveAssociationToHandle(in->nvIndex);
    return result;
}
#endif // CC_NV_UndefineSpaceSpecial
```


31.6 TPM2_NV_ReadPublic

31.6.1 General Description

This command is used to read the public area and Name of an NV Index. The public area of an Index is not privacy-sensitive and no authorization is required to read this data.

31.6.2 Command and Response

Table 218 — TPM2_NV_ReadPublic Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit or encrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_NV_ReadPublic
TPMI_RH_NV_INDEX	nvIndex	the NV Index Auth Index: None

Table 219 — TPM2_NV_ReadPublic Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_NV_PUBLIC	nvPublic	the public area of the NV Index
TPM2B_NAME	nvName	the Name of the *nvIndex*

31.6.3 Detailed Actions

```
#include "Tpm.h"
#include "NV_ReadPublic_fp.h"
#if CC_NV_ReadPublic  // Conditional expansion of this file
TPM_RC
TPM2_NV_ReadPublic(
    NV_ReadPublic_In    *in,            // IN: input parameter list
    NV_ReadPublic_Out   *out            // OUT: output parameter list
    )
{
    NV_INDEX        *nvIndex = NvGetIndexInfo(in->nvIndex, NULL);

// Command Output

    // Copy index public data to output
    out->nvPublic.nvPublic = nvIndex->publicArea;

    // Compute NV name
    NvGetIndexName(nvIndex, &out->nvName);

    return TPM_RC_SUCCESS;
}
#endif // CC_NV_ReadPublic
```

31.7 TPM2_NV_Write

31.7.1 General Description

This command writes a value to an area in NV memory that was previously defined by TPM2_NV_DefineSpace().

Proper authorizations are required for this command as determined by TPMA_NV_PPWRITE; TPMA_NV_OWNERWRITE; TPMA_NV_AUTHWRITE; and, if TPMA_NV_POLICY_WRITE is SET, the *authPolicy* of the NV Index.

If the TPMA_NV_WRITELOCKED attribute of the NV Index is SET, then the TPM shall return TPM_RC_NV_LOCKED.

NOTE 1 If authorization sessions are present, they are checked before checks to see if writes to the NV Index are locked.

If *nvIndexType* is TPM_NT_COUNTER, TPM_NT_BITS or TPM_NT_EXTEND, then the TPM shall return TPM_RC_ATTRIBUTES.

If *offset* and the *size* field of *data* add to a value that is greater than the *dataSize* field of the NV Index referenced by *nvIndex*, the TPM shall return an error (TPM_RC_NV_RANGE). The implementation may return an error (TPM_RC_VALUE) if it performs an additional check and determines that *offset* is greater than the *dataSize* field of the NV Index.

If the TPMA_NV_WRITEALL attribute of the NV Index is SET, then the TPM shall return TPM_RC_NV_RANGE if the size of the *data* parameter of the command is not the same as the *data* field of the NV Index.

If all checks succeed, the TPM will merge the *data.size* octets of *data.buffer* value into the *nvIndex→data* starting at *nvIndex→data[offset].* If the NV memory is implemented with a technology that has endurance limitations, the TPM shall check that the merged data is different from the current contents of the NV Index and only perform a write to NV memory if they differ.

After successful completion of this command, TPMA_NV_WRITTEN for the NV Index will be SET.

NOTE 2 Once SET, TPMA_NV_WRITTEN remains SET until the NV Index is undefined or the NV Index is cleared.

31.7.2 Command and Response

Table 220 — TPM2_NV_Write Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_NV_Write {NV}
TPMI_RH_NV_AUTH	@authHandle	handle indicating the source of the authorization value Auth Index: 1 Auth Role: USER
TPMI_RH_NV_INDEX	nvIndex	the NV Index of the area to write Auth Index: None
TPM2B_MAX_NV_BUFFER	data	the data to write
UINT16	offset	the octet offset into the NV Area

Table 221 — TPM2_NV_Write Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

31.7.3 Detailed Actions

```
#include "Tpm.h"
#include "NV_Write_fp.h"
#if CC_NV_Write  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_ATTRIBUTES	Index referenced by *nvIndex* has either TPMA_NV_BITS, TPMA_NV_COUNTER, or TPMA_NV_EVENT attribute SET
TPM_RC_NV_AUTHORIZATION	the authorization was valid but the authorizing entity (*authHandle*) is not allowed to write to the Index referenced by *nvIndex*
TPM_RC_NV_LOCKED	Index referenced by *nvIndex* is write locked
TPM_RC_NV_RANGE	if TPMA_NV_WRITEALL is SET then the write is not the size of the Index referenced by *nvIndex*; otherwise, the write extends beyond the limits of the Index

```
TPM_RC
TPM2_NV_Write(
    NV_Write_In     *in             // IN: input parameter list
    )
{
    NV_INDEX        *nvIndex = NvGetIndexInfo(in->nvIndex, NULL);
    TPMA_NV          attributes = nvIndex->publicArea.attributes;
    TPM_RC           result;

// Input Validation

    // Common access checks, NvWriteAccessCheck() may return TPM_RC_NV_AUTHORIZATION
    // or TPM_RC_NV_LOCKED
    result = NvWriteAccessChecks(in->authHandle,
                                 in->nvIndex,
                                 attributes);
    if(result != TPM_RC_SUCCESS)
        return result;

    // Bits index, extend index or counter index may not be updated by
    // TPM2_NV_Write
    if(IsNvCounterIndex(attributes)
       || IsNvBitsIndex(attributes)
       || IsNvExtendIndex(attributes))
        return TPM_RC_ATTRIBUTES;

    // Make sure that the offset is not too large
    if(in->offset > nvIndex->publicArea.dataSize)
        return TPM_RCS_VALUE + RC_NV_Write_offset;

    // Make sure that the selection is within the range of the Index
    if(in->data.t.size > (nvIndex->publicArea.dataSize - in->offset))
        return TPM_RC_NV_RANGE;

    // If this index requires a full sized write, make sure that input range is
    // full sized.
    // Note: if the requested size is the same as the Index data size, then offset
    // will have to be zero. Otherwise, the range check above would have failed.
    if(IS_ATTRIBUTE(attributes, TPMA_NV, WRITEALL)
       && in->data.t.size < nvIndex->publicArea.dataSize)
        return TPM_RC_NV_RANGE;

// Internal Data Update

```

```
    // Perform the write.  This called routine will SET the TPMA_NV_WRITTEN
    // attribute if it has not already been SET. If NV isn't available, an error
    // will be returned.
    return NvWriteIndexData(nvIndex, in->offset, in->data.t.size,
                            in->data.t.buffer);
}
#endif // CC_NV_Write
```


31.8 TPM2_NV_Increment

31.8.1 General Description

This command is used to increment the value in an NV Index that has the TPM_NT_COUNTER attribute. The data value of the NV Index is incremented by one.

NOTE 1 The NV Index counter is an unsigned value.

If *nvIndexType* is not TPM_NT_COUNTER in the indicated NV Index, the TPM shall return TPM_RC_ATTRIBUTES.

If TPMA_NV_WRITELOCKED is SET, the TPM shall return TPM_RC_NV_LOCKED.

If TPMA_NV_WRITTEN is CLEAR, it will be SET.

If TPMA_NV_ORDERLY is SET, and the difference between the volatile and non-volatile versions of this field is greater than MAX_ORDERLY_COUNT, then the non-volatile version of the counter is updated.

NOTE 2 If a TPM implements TPMA_NV_ORDERLY and an Index is defined with TPMA_NV_ORDERLY and TPM_NT_COUNTER both SET, then in the Event of a non-orderly shutdown, the non-volatile value for the counter Index will be advanced by MAX_ORDERLY_COUNT at the next TPM2_Startup().

NOTE 3 An allowed implementation would keep a counter value in NV and a resettable counter in RAM. The reported value of the NV Index would be the sum of the two values. When the RAM count increments past the maximum allowed value (MAX_ORDERLY_COUNT), the non-volatile version of the count is updated with the sum of the values and the RAM count is reset to zero.

31.8.2 Command and Response

Table 222 — TPM2_NV_Increment Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_NV_Increment {NV}
TPMI_RH_NV_AUTH	@authHandle	handle indicating the source of the authorization value Auth Index: 1 Auth Role: USER
TPMI_RH_NV_INDEX	nvIndex	the NV Index to increment Auth Index: None

Table 223 — TPM2_NV_Increment Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

31.8.3 Detailed Actions

```
#include "Tpm.h"
#include "NV_Increment_fp.h"
#if CC_NV_Increment  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_ATTRIBUTES	NV index is not a counter
TPM_RC_NV_AUTHORIZATION	authorization failure
TPM_RC_NV_LOCKED	Index is write locked

```
TPM_RC
TPM2_NV_Increment(
    NV_Increment_In     *in             // IN: input parameter list
    )
{
    TPM_RC           result;
    NV_REF           locator;
    NV_INDEX        *nvIndex = NvGetIndexInfo(in->nvIndex, &locator);
    UINT64           countValue;

// Input Validation

    // Common access checks, NvWriteAccessCheck() may return TPM_RC_NV_AUTHORIZATION
    // or TPM_RC_NV_LOCKED
    result = NvWriteAccessChecks(in->authHandle,
                                 in->nvIndex,
                                 nvIndex->publicArea.attributes);
    if(result != TPM_RC_SUCCESS)
        return result;

    // Make sure that this is a counter
    if(!IsNvCounterIndex(nvIndex->publicArea.attributes))
        return TPM_RCS_ATTRIBUTES + RC_NV_Increment_nvIndex;

// Internal Data Update

    // If counter index is not been written, initialize it
    if(!IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, WRITTEN))
        countValue = NvReadMaxCount();
    else
        // Read NV data in native format for TPM CPU.
        countValue = NvGetUINT64Data(nvIndex, locator);

    // Do the increment
    countValue++;

    // Write NV data back. A TPM_RC_NV_UNAVAILABLE or TPM_RC_NV_RATE error may
    // be returned at this point. If necessary, this function will set the
    // TPMA_NV_WRITTEN attribute
    result = NvWriteUINT64Data(nvIndex, countValue);
    if(result == TPM_RC_SUCCESS)
    {
        // If a counter just rolled over, then force the NV update.
        // Note, if this is an orderly counter, then the write-back needs to be
        // forced, for other counters, the write-back will happen anyway
        if(IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, ORDERLY)
           && (countValue & MAX_ORDERLY_COUNT) == 0 )
        {
            // Need to force an NV update of orderly data
```



```
            SET_NV_UPDATE(UT_ORDERLY);
        }
    }
    return result;
}
#endif // CC_NV_Increment
```

31.9 TPM2_NV_Extend

31.9.1 General Description

This command extends a value to an area in NV memory that was previously defined by TPM2_NV_DefineSpace.

If *nvIndexType* is not TPM_NT_EXTEND, then the TPM shall return TPM_RC_ATTRIBUTES.

Proper write authorizations are required for this command as determined by TPMA_NV_PPWRITE, TPMA_NV_OWNERWRITE, TPMA_NV_AUTHWRITE, and the *authPolicy* of the NV Index.

After successful completion of this command, TPMA_NV_WRITTEN for the NV Index will be SET.

NOTE 1 Once SET, TPMA_NV_WRITTEN remains SET until the NV Index is undefined, unless the TPMA_NV_CLEAR_STCLEAR attribute is SET and a TPM Reset or TPM Restart occurs.

If the TPMA_NV_WRITELOCKED attribute of the NV Index is SET, then the TPM shall return TPM_RC_NV_LOCKED.

NOTE 2 If authorization sessions are present, they are checked before checks to see if writes to the NV Index are locked.

The *data.buffer* parameter may be larger than the defined size of the NV Index.

The Index will be updated by:

$$nvIndex{\rightarrow}data_{new} \coloneqq \mathbf{H}_{nameAkg}(nvIndex{\rightarrow}data_{old} \mid\mid data.buffer) \tag{41}$$

where

$nvIndex{\rightarrow}data_{new}$	the value of the data field in the NV Index after the command returns
$\mathbf{H}_{nameAkg}()$	the hash algorithm indicated in *nvIndex→nameAlg*
$nvIndex{\rightarrow}data_{old}$	the value of the data field in the NV Index before the command is called
$data.buffer$	the data buffer of the command parameter

NOTE 3 If TPMA_NV_WRITTEN is CLEAR, then $nvIndex{\rightarrow}data_{old}$ is a Zero Digest.

31.9.2 Command and Response

Table 224 — TPM2_NV_Extend Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_NV_Extend {NV}
TPMI_RH_NV_AUTH	@authHandle	handle indicating the source of the authorization value Auth Index: 1 Auth Role: USER
TPMI_RH_NV_INDEX	nvIndex	the NV Index to extend Auth Index: None
TPM2B_MAX_NV_BUFFER	data	the data to extend

Table 225 — TPM2_NV_Extend Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

31.9.3 Detailed Actions

```
#include "Tpm.h"
#include "NV_Extend_fp.h"
#if CC_NV_Extend  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_ATTRIBUTES	the TPMA_NV_EXTEND attribute is not SET in the Index referenced by *nvIndex*
TPM_RC_NV_AUTHORIZATION	the authorization was valid but the authorizing entity (*authHandle*) is not allowed to write to the Index referenced by *nvIndex*
TPM_RC_NV_LOCKED	the Index referenced by *nvIndex* is locked for writing

```
TPM_RC
TPM2_NV_Extend(
    NV_Extend_In    *in             // IN: input parameter list
    )
{
    TPM_RC                  result;
    NV_REF                  locator;
    NV_INDEX               *nvIndex = NvGetIndexInfo(in->nvIndex, &locator);

    TPM2B_DIGEST           oldDigest;
    TPM2B_DIGEST           newDigest;
    HASH_STATE             hashState;

// Input Validation

    // Common access checks, NvWriteAccessCheck() may return TPM_RC_NV_AUTHORIZATION
    // or TPM_RC_NV_LOCKED
    result = NvWriteAccessChecks(in->authHandle,
                                 in->nvIndex,
                                 nvIndex->publicArea.attributes);
    if(result != TPM_RC_SUCCESS)
        return result;

    // Make sure that this is an extend index
    if(!IsNvExtendIndex(nvIndex->publicArea.attributes))
        return TPM_RCS_ATTRIBUTES + RC_NV_Extend_nvIndex;

// Internal Data Update

    // Perform the write.
    oldDigest.t.size = CryptHashGetDigestSize(nvIndex->publicArea.nameAlg);
    pAssert(oldDigest.t.size <= sizeof(oldDigest.t.buffer));
    if(IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, WRITTEN))
    {
        NvGetIndexData(nvIndex, locator, 0, oldDigest.t.size, oldDigest.t.buffer);
    }
    else
    {
        MemorySet(oldDigest.t.buffer, 0, oldDigest.t.size);
    }
    // Start hash
    newDigest.t.size = CryptHashStart(&hashState, nvIndex->publicArea.nameAlg);

    // Adding old digest
    CryptDigestUpdate2B(&hashState, &oldDigest.b);

    // Adding new data
```

```
    CryptDigestUpdate2B(&hashState, &in->data.b);

    // Complete hash
    CryptHashEnd2B(&hashState, &newDigest.b);

    // Write extended hash back.
    // Note, this routine will SET the TPMA_NV_WRITTEN attribute if necessary
    return NvWriteIndexData(nvIndex, 0, newDigest.t.size, newDigest.t.buffer);
}
#endif // CC_NV_Extend
```

31.10 TPM2_NV_SetBits

31.10.1 General Description

This command is used to SET bits in an NV Index that was created as a bit field. Any number of bits from 0 to 64 may be SET. The contents of *bits* are ORed with the current contents of the NV Index.

If TPMA_NV_WRITTEN is not SET, then, for the purposes of this command, the NV Index is considered to contain all zero bits and *data* is ORed with that value.

If TPM_NT_BITS is not SET, then the TPM shall return TPM_RC_ATTRIBUTES.

After successful completion of this command, TPMA_NV_WRITTEN for the NV Index will be SET.

NOTE TPMA_NV_WRITTEN will be SET even if no bits were SET.

31.10.2 Command and Response

Table 226 — TPM2_NV_SetBits Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_NV_SetBits {NV}
TPMI_RH_NV_AUTH	@authHandle	handle indicating the source of the authorization value Auth Index: 1 Auth Role: USER
TPMI_RH_NV_INDEX	nvIndex	NV Index of the area in which the bit is to be set Auth Index: None
UINT64	bits	the data to OR with the current contents

Table 227 — TPM2_NV_SetBits Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

31.10.3 Detailed Actions

```
#include "Tpm.h"
#include "NV_SetBits_fp.h"
#if CC_NV_SetBits  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_ATTRIBUTES	the TPMA_NV_BITS attribute is not SET in the Index referenced by *nvIndex*
TPM_RC_NV_AUTHORIZATION	the authorization was valid but the authorizing entity (*authHandle*) is not allowed to write to the Index referenced by *nvIndex*
TPM_RC_NV_LOCKED	the Index referenced by *nvIndex* is locked for writing

```
TPM_RC
TPM2_NV_SetBits(
    NV_SetBits_In   *in             // IN: input parameter list
    )
{
    TPM_RC           result;
    NV_REF           locator;
    NV_INDEX        *nvIndex = NvGetIndexInfo(in->nvIndex, &locator);
    UINT64           oldValue;
    UINT64           newValue;

// Input Validation

    // Common access checks, NvWriteAccessCheck() may return TPM_RC_NV_AUTHORIZATION
    // or TPM_RC_NV_LOCKED
    result = NvWriteAccessChecks(in->authHandle,
                                 in->nvIndex,
                                 nvIndex->publicArea.attributes);
    if(result != TPM_RC_SUCCESS)
        return result;

    // Make sure that this is a bit field
    if(!IsNvBitsIndex(nvIndex->publicArea.attributes))
        return TPM_RCS_ATTRIBUTES + RC_NV_SetBits_nvIndex;

    // If index is not been written, initialize it
    if(!IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, WRITTEN))
        oldValue = 0;
    else
        // Read index data
        oldValue = NvGetUINT64Data(nvIndex, locator);

    // Figure out what the new value is going to be
    newValue = oldValue | in->bits;

// Internal Data Update
    return  NvWriteUINT64Data(nvIndex, newValue);
}
#endif // CC_NV_SetBits
```


31.11 TPM2_NV_WriteLock

31.11.1 General Description

If the TPMA_NV_WRITEDEFINE or TPMA_NV_WRITE_STCLEAR attributes of an NV location are SET, then this command may be used to inhibit further writes of the NV Index.

Proper write authorization is required for this command as determined by TPMA_NV_PPWRITE, TPMA_NV_OWNERWRITE, TPMA_NV_AUTHWRITE, and the *authPolicy* of the NV Index.

It is not an error if TPMA_NV_WRITELOCKED for the NV Index is already SET.

If neither TPMA_NV_WRITEDEFINE nor TPMA_NV_WRITE_STCLEAR of the NV Index is SET, then the TPM shall return TPM_RC_ATTRIBUTES.

If the command is properly authorized and TPMA_NV_WRITE_STCLEAR or TPMA_NV_WRITEDEFINE is SET, then the TPM shall SET TPMA_NV_WRITELOCKED for the NV Index. TPMA_NV_WRITELOCKED will be clear on the next TPM2_Startup(TPM_SU_CLEAR) if either TPMA_NV_WRITEDEFINE is CLEAR or TPMA_NV_WRITTEN is CLEAR.

31.11.2 Command and Response

Table 228 — TPM2_NV_WriteLock Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_NV_WriteLock {NV}
TPMI_RH_NV_AUTH	@authHandle	handle indicating the source of the authorization value Auth Index: 1 Auth Role: USER
TPMI_RH_NV_INDEX	nvIndex	the NV Index of the area to lock Auth Index: None

Table 229 — TPM2_NV_WriteLock Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

31.11.3 Detailed Actions

```
#include "Tpm.h"
#include "NV_WriteLock_fp.h"
#if CC_NV_WriteLock  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_ATTRIBUTES	neither TPMA_NV_WRITEDEFINE nor TPMA_NV_WRITE_STCLEAR is SET in Index referenced by *nvIndex*
TPM_RC_NV_AUTHORIZATION	the authorization was valid but the authorizing entity (*authHandle*) is not allowed to write to the Index referenced by *nvIndex*

```
TPM_RC
TPM2_NV_WriteLock(
    NV_WriteLock_In     *in              // IN: input parameter list
    )
{
    TPM_RC            result;
    NV_REF            locator;
    NV_INDEX         *nvIndex = NvGetIndexInfo(in->nvIndex, &locator);
    TPMA_NV           nvAttributes = nvIndex->publicArea.attributes;

// Input Validation:

    // Common access checks, NvWriteAccessCheck() may return TPM_RC_NV_AUTHORIZATION
    // or TPM_RC_NV_LOCKED
    result = NvWriteAccessChecks(in->authHandle, in->nvIndex, nvAttributes);
    if(result != TPM_RC_SUCCESS)
    {
        if(result == TPM_RC_NV_AUTHORIZATION)
            return result;
        // If write access failed because the index is already locked, then it is
        // no error.
        return TPM_RC_SUCCESS;
    }
    // if neither TPMA_NV_WRITEDEFINE nor TPMA_NV_WRITE_STCLEAR is set, the index
    // can not be write-locked
    if(!IS_ATTRIBUTE(nvAttributes, TPMA_NV, WRITEDEFINE)
       && !IS_ATTRIBUTE(nvAttributes, TPMA_NV, WRITE_STCLEAR))
        return TPM_RCS_ATTRIBUTES + RC_NV_WriteLock_nvIndex;
// Internal Data Update
    // Set the WRITELOCK attribute.
    // Note: if TPMA_NV_WRITELOCKED were already SET, then the write access check
    // above would have failed and this code isn't executed.
    SET_ATTRIBUTE(nvAttributes, TPMA_NV, WRITELOCKED);

    // Write index info back
    return NvWriteIndexAttributes(nvIndex->publicArea.nvIndex, locator,
                                  nvAttributes);
}
#endif // CC_NV_WriteLock
```

31.12 TPM2_NV_GlobalWriteLock

31.12.1 General Description

The command will SET TPMA_NV_WRITELOCKED for all indexes that have their TPMA_NV_GLOBALLOCK attribute SET.

If an Index has both TPMA_NV_GLOBALLOCK and TPMA_NV_WRITEDEFINE SET, then this command will permanently lock the NV Index for writing unless TPMA_NV_WRITTEN is CLEAR.

NOTE If an Index is defined with TPMA_NV_GLOBALLOCK SET, then the global lock does not apply until the next time this command is executed.

This command requires either platformAuth/platformPolicy or ownerAuth/ownerPolicy.

31.12.2 Command and Response

Table 230 — TPM2_NV_GlobalWriteLock Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_NV_GlobalWriteLock {NV}
TPMI_RH_PROVISION	@authHandle	TPM_RH_OWNER or TPM_RH_PLATFORM+{PP} Auth Index: 1 Auth Role: USER

Table 231 — TPM2_NV_GlobalWriteLock Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

31.12.3 Detailed Actions

```
#include "Tpm.h"
#include "NV_GlobalWriteLock_fp.h"
#if CC_NV_GlobalWriteLock  // Conditional expansion of this file
TPM_RC
TPM2_NV_GlobalWriteLock(
    NV_GlobalWriteLock_In   *in             // IN: input parameter list
    )
{
    // Input parameter (the authorization handle) is not reference in command action.
    NOT_REFERENCED(in);

// Internal Data Update

    // Implementation dependent method of setting the global lock
    return NvSetGlobalLock();
}
#endif // CC_NV_GlobalWriteLock
```


31.13 TPM2_NV_Read

31.13.1 General Description

This command reads a value from an area in NV memory previously defined by TPM2_NV_DefineSpace().

Proper authorizations are required for this command as determined by TPMA_NV_PPREAD, TPMA_NV_OWNERREAD, TPMA_NV_AUTHREAD, and the *authPolicy* of the NV Index.

If TPMA_NV_READLOCKED of the NV Index is SET, then the TPM shall return TPM_RC_NV_LOCKED.

If *offset* and the *size* field of *data* add to a value that is greater than the *dataSize* field of the NV Index referenced by *nvIndex*, the TPM shall return an error (TPM_RC_NV_RANGE). The implementation may return an error (TPM_RC_VALUE) if it performs an additional check and determines that *offset* is greater than the *dataSize* field of the NV Index.

For an NV Index with the TPM_NT_COUNTER or TPM_NT_BITS attribute SET, the TPM may ignore the *offset* parameter and use an offset of 0. Therefore, it is recommended that the caller set the *offset* parameter to 0 for interoperability.

NOTE 1 If authorization sessions are present, they are checked before the read-lock status of the NV Index is checked.

If the NV Index has been defined but the TPMA_NV_WRITTEN attribute is CLEAR, then this command shall return TPM_RC_NV_UNINITIALIZED even if *size* is zero.

The *data* parameter in the response may be encrypted using parameter encryption.

31.13.2 Command and Response

Table 232 — TPM2_NV_Read Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_NV_Read
TPMI_RH_NV_AUTH	@authHandle	the handle indicating the source of the authorization value Auth Index: 1 Auth Role: USER
TPMI_RH_NV_INDEX	nvIndex	the NV Index to be read Auth Index: None
UINT16	size	number of octets to read
UINT16	offset	octet offset into the NV area This value shall be less than or equal to the size of the *nvIndex* data.

Table 233 — TPM2_NV_Read Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_MAX_NV_BUFFER	data	the data read

31.13.3 Detailed Actions

```
#include "Tpm.h"
#include "NV_Read_fp.h"
#if CC_NV_Read  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_NV_AUTHORIZATION	the authorization was valid but the authorizing entity (*authHandle*) is not allowed to read from the Index referenced by *nvIndex*
TPM_RC_NV_LOCKED	the Index referenced by *nvIndex* is read locked
TPM_RC_NV_RANGE	read range defined by *size* and *offset* is outside the range of the Index referenced by *nvIndex*
TPM_RC_NV_UNINITIALIZED	the Index referenced by *nvIndex* has not been initialized (written)
TPM_RC_VALUE	the read size is larger than the MAX_NV_BUFFER_SIZE

```
TPM_RC
TPM2_NV_Read(
    NV_Read_In      *in,            // IN: input parameter list
    NV_Read_Out     *out            // OUT: output parameter list
    )
{
    NV_REF           locator;
    NV_INDEX        *nvIndex = NvGetIndexInfo(in->nvIndex, &locator);
    TPM_RC           result;

// Input Validation
    // Common read access checks. NvReadAccessChecks() may return
    // TPM_RC_NV_AUTHORIZATION, TPM_RC_NV_LOCKED, or TPM_RC_NV_UNINITIALIZED
    result = NvReadAccessChecks(in->authHandle, in->nvIndex,
                                nvIndex->publicArea.attributes);
    if(result != TPM_RC_SUCCESS)
        return result;

    // Make sure the data will fit the return buffer
    if(in->size > MAX_NV_BUFFER_SIZE)
        return TPM_RCS_VALUE + RC_NV_Read_size;

    // Verify that the offset is not too large
    if(in->offset > nvIndex->publicArea.dataSize)
        return TPM_RCS_VALUE + RC_NV_Read_offset;

    // Make sure that the selection is within the range of the Index
    if(in->size > (nvIndex->publicArea.dataSize - in->offset))
        return TPM_RC_NV_RANGE;

// Command Output
    // Set the return size
    out->data.t.size = in->size;

    // Perform the read
    NvGetIndexData(nvIndex, locator, in->offset, in->size, out->data.t.buffer);

    return TPM_RC_SUCCESS;
}
#endif // CC_NV_Read
```


31.14 TPM2_NV_ReadLock

31.14.1 General Description

If TPMA_NV_READ_STCLEAR is SET in an Index, then this command may be used to prevent further reads of the NV Index until the next TPM2_Startup (TPM_SU_CLEAR).

Proper authorizations are required for this command as determined by TPMA_NV_PPREAD, TPMA_NV_OWNERREAD, TPMA_NV_AUTHREAD, and the *authPolicy* of the NV Index.

NOTE Only an entity that may read an Index is allowed to lock the NV Index for read.

If the command is properly authorized and TPMA_NV_READ_STCLEAR of the NV Index is SET, then the TPM shall SET TPMA_NV_READLOCKED for the NV Index. If TPMA_NV_READ_STCLEAR of the NV Index is CLEAR, then the TPM shall return TPM_RC_ATTRIBUTES. TPMA_NV_READLOCKED will be CLEAR by the next TPM2_Startup(TPM_SU_CLEAR).

It is not an error to use this command for an Index that is already locked for reading.

An Index that had not been written may be locked for reading.

31.14.2 Command and Response

Table 234 — TPM2_NV_ReadLock Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_NV_ReadLock {NV}
TPMI_RH_NV_AUTH	@authHandle	the handle indicating the source of the authorization value Auth Index: 1 Auth Role: USER
TPMI_RH_NV_INDEX	nvIndex	the NV Index to be locked Auth Index: None

Table 235 — TPM2_NV_ReadLock Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

31.14.3 Detailed Actions

```
#include "Tpm.h"
#include "NV_ReadLock_fp.h"
#if CC_NV_ReadLock  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_ATTRIBUTES	TPMA_NV_READ_STCLEAR is not SET so Index referenced by *nvIndex* may not be write locked
TPM_RC_NV_AUTHORIZATION	the authorization was valid but the authorizing entity (*authHandle*) is not allowed to read from the Index referenced by *nvIndex*

```
TPM_RC
TPM2_NV_ReadLock(
    NV_ReadLock_In  *in             // IN: input parameter list
    )
{
    TPM_RC           result;
    NV_REF           locator;
    // The referenced index has been checked multiple times before this is called
    // so it must be present and will be loaded into cache
    NV_INDEX        *nvIndex = NvGetIndexInfo(in->nvIndex, &locator);
    TPMA_NV          nvAttributes = nvIndex->publicArea.attributes;

// Input Validation
    // Common read access checks. NvReadAccessChecks() may return
    // TPM_RC_NV_AUTHORIZATION, TPM_RC_NV_LOCKED, or TPM_RC_NV_UNINITIALIZED
    result = NvReadAccessChecks(in->authHandle,
                                in->nvIndex,
                                nvAttributes);
    if(result == TPM_RC_NV_AUTHORIZATION)
        return TPM_RC_NV_AUTHORIZATION;
    // Index is already locked for write
    else if(result == TPM_RC_NV_LOCKED)
            return TPM_RC_SUCCESS;

    // If NvReadAccessChecks return TPM_RC_NV_UNINITALIZED, then continue.
    // It is not an error to read lock an uninitialized Index.

    // if TPMA_NV_READ_STCLEAR is not set, the index can not be read-locked
    if(!IS_ATTRIBUTE(nvAttributes, TPMA_NV, READ_STCLEAR))
        return TPM_RCS_ATTRIBUTES + RC_NV_ReadLock_nvIndex;

// Internal Data Update

    // Set the READLOCK attribute
    SET_ATTRIBUTE(nvAttributes, TPMA_NV, READLOCKED);

    // Write NV info back
    return NvWriteIndexAttributes(nvIndex->publicArea.nvIndex,
                                  locator,
                                  nvAttributes);
}
#endif // CC_NV_ReadLock
```


31.15 TPM2_NV_ChangeAuth

31.15.1 General Description

This command allows the authorization secret for an NV Index to be changed.

If successful, the authorization secret (*authValue*) of the NV Index associated with *nvIndex* is changed.

This command requires that a policy session be used for authorization of *nvIndex* so that the ADMIN role may be asserted and that *commandCode* in the policy session context shall be TPM_CC_NV_ChangeAuth. That is, the policy must contain a specific authorization for changing the authorization value of the referenced entity.

NOTE The reason for this restriction is to ensure that the administrative actions on *nvIndex* require explicit approval while other commands may use policy that is not command-dependent.

The size of the *newAuth* value may be no larger than the size of the digest produced by the *nameAlg* of the NV Index.

Since the NV Index authorization is changed before the response HMAC is calculated, the newAuth value is used when generating the response HMAC key if required. See TPM 2.0 Part 4 ComputeResponseHMAC().

31.15.2 Command and Response

Table 236 — TPM2_NV_ChangeAuth Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_NV_ChangeAuth {NV}
TPMI_RH_NV_INDEX	@nvIndex	handle of the entity Auth Index: 1 Auth Role: ADMIN
TPM2B_AUTH	newAuth	new authorization value

Table 237 — TPM2_NV_ChangeAuth Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

31.15.3 Detailed Actions

```
#include "Tpm.h"
#include "NV_ChangeAuth_fp.h"
#if CC_NV_ChangeAuth  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_SIZE	*newAuth* size is larger than the digest size of the Name algorithm for the Index referenced by '*nvIndex*

```
TPM_RC
TPM2_NV_ChangeAuth(
    NV_ChangeAuth_In    *in             // IN: input parameter list
    )
{
    NV_REF           locator;
    NV_INDEX        *nvIndex = NvGetIndexInfo(in->nvIndex, &locator);

// Input Validation

    // Remove trailing zeros and make sure that the result is not larger than the
    // digest of the nameAlg.
    if(MemoryRemoveTrailingZeros(&in->newAuth)
       > CryptHashGetDigestSize(nvIndex->publicArea.nameAlg))
        return TPM_RCS_SIZE + RC_NV_ChangeAuth_newAuth;

// Internal Data Update
    // Change authValue
    return NvWriteIndexAuth(locator, &in->newAuth);
}
#endif // CC_NV_ChangeAuth
```


31.16 TPM2_NV_Certify

31.16.1 General Description

The purpose of this command is to certify the contents of an NV Index or portion of an NV Index.

If the *sign* attribute is not SET in the key referenced by *signHandle* then the TPM shall return TPM_RC_KEY.

If the NV Index has been defined but the TPMA_NV_WRITTEN attribute is CLEAR, then this command shall return TPM_RC_NV_UNINITIALIZED even if *size* is zero.

If proper authorization for reading the NV Index is provided, the portion of the NV Index selected by *size* and *offset* are included in an attestation block and signed using the key indicated by *signHandle*. The attestation includes *size* and *offset* so that the range of the data can be determined. It also includes the NV index Name.

For an NV Index with the TPM_NT_COUNTER or TPM_NT_BITS attribute SET, the TPM may ignore the *offset* parameter and use an offset of 0. Therefore, it is recommended that the caller set the *offset* parameter to 0 for interoperability.

If *offset* and *size* add to a value that is greater than the *dataSize* field of the NV Index referenced by *nvIndex*, the TPM shall return an error (TPM_RC_NV_RANGE). The implementation may return an error (TPM_RC_VALUE) if it performs an additional check and determines that *offset* is greater than the *dataSize* field of the NV Index, or if *size* is greater than MAX_NV_BUFFER_SIZE.

NOTE 1 See 18.1 for description of how the signing scheme is selected.

NOTE 2 If *signHandle* is TPM_RH_NULL, the TPMS_ATTEST structure is returned and *signature* is a NULL Signature.

If *size* and *offset* are both zero (0), then *certifyInfo* in the response will contain a TPMS_NV_DIGEST_CERTIFY_INFO, otherwise, it will contain a TPMS_NV_CERTIFY_INFO. The digest in the TPMS_NV_DIGEST_CERTIFY_INFO is created using the digest of the selected signing scheme.

NOTE 3 TPMS_NV_DIGEST_CERTIFY_INFO was added in revision 01.53. It permits TPM2_NV_Certify() to certify NV Index contents that are larger than MAX_NV_BUFFER_SIZE.

31.16.2 Command and Response

Table 238 — TPM2_NV_Certify Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_NV_Certify
TPMI_DH_OBJECT+	@signHandle	handle of the key used to sign the attestation structure Auth Index: 1 Auth Role: USER
TPMI_RH_NV_AUTH	@authHandle	handle indicating the source of the authorization value for the NV Index Auth Index: 2 Auth Role: USER
TPMI_RH_NV_INDEX	nvIndex	Index for the area to be certified Auth Index: None
TPM2B_DATA	qualifyingData	user-provided qualifying data
TPMT_SIG_SCHEME+	inScheme	signing scheme to use if the *scheme* for *signHandle* is TPM_ALG_NULL
UINT16	size	number of octets to certify
UINT16	offset	octet offset into the NV area This value shall be less than or equal to the size of the *nvIndex* data.

Table 239 — TPM2_NV_Certify Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	.
TPM2B_ATTEST	certifyInfo	the structure that was signed
TPMT_SIGNATURE	signature	the asymmetric signature over *certifyInfo* using the key referenced by *signHandle*

31.16.3 Detailed Actions

```
#include "Tpm.h"
#include "Attest_spt_fp.h"
#include "NV_Certify_fp.h"
#if CC_NV_Certify  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_NV_AUTHORIZATION	the authorization was valid but the authorizing entity (*authHandle*) is not allowed to read from the Index referenced by *nvIndex*
TPM_RC_KEY	*signHandle* does not reference a signing key
TPM_RC_NV_LOCKED	Index referenced by *nvIndex* is locked for reading
TPM_RC_NV_RANGE	*offset* plus *size* extends outside of the data range of the Index referenced by *nvIndex*
TPM_RC_NV_UNINITIALIZED	Index referenced by *nvIndex* has not been written
TPM_RC_SCHEME	*inScheme* is not an allowed value for the key definition

```
TPM_RC
TPM2_NV_Certify(
    NV_Certify_In   *in,            // IN: input parameter list
    NV_Certify_Out  *out            // OUT: output parameter list
    )
{
    TPM_RC                  result;
    NV_REF                  locator;
    NV_INDEX               *nvIndex = NvGetIndexInfo(in->nvIndex, &locator);
    TPMS_ATTEST             certifyInfo;
    OBJECT                 *signObject = HandleToObject(in->signHandle);
// Input Validation
    if(!IsSigningObject(signObject))
        return TPM_RCS_KEY + RC_NV_Certify_signHandle;
    if(!CryptSelectSignScheme(signObject, &in->inScheme))
        return TPM_RCS_SCHEME + RC_NV_Certify_inScheme;

    // Common access checks, NvWriteAccessCheck() may return TPM_RC_NV_AUTHORIZATION
    // or TPM_RC_NV_LOCKED
    result = NvReadAccessChecks(in->authHandle, in->nvIndex,
                                nvIndex->publicArea.attributes);
    if(result != TPM_RC_SUCCESS)
        return result;

    // make sure that the selection is within the range of the Index (cast to avoid
    // any wrap issues with addition)
    if((UINT32)in->size + (UINT32)in->offset > (UINT32)nvIndex->publicArea.dataSize)
        return TPM_RC_NV_RANGE;
    // Make sure the data will fit the return buffer.
    // NOTE: This check may be modified if the output buffer will not hold the
    // maximum sized NV buffer as part of the certified data. The difference in
    // size could be substantial if the signature scheme was produced a large
    // signature (e.g., RSA 4096).
    if(in->size > MAX_NV_BUFFER_SIZE)
        return TPM_RCS_VALUE + RC_NV_Certify_size;

// Command Output

    // Fill in attest information common fields
    FillInAttestInfo(in->signHandle, &in->inScheme, &in->qualifyingData,
                     &certifyInfo);
```



```

    // Get the name of the index
    NvGetIndexName(nvIndex, &certifyInfo.attested.nv.indexName);

    // See if this is old format or new format
    if ((in->size != 0) || (in->offset != 0))
    {
        // NV certify specific fields
        // Attestation type
        certifyInfo.type = TPM_ST_ATTEST_NV;

        // Set the return size
        certifyInfo.attested.nv.nvContents.t.size = in->size;

        // Set the offset
        certifyInfo.attested.nv.offset = in->offset;

        // Perform the read
        NvGetIndexData(nvIndex, locator, in->offset, in->size,
            certifyInfo.attested.nv.nvContents.t.buffer);
    }
    else
    {
        HASH_STATE                  hashState;
        // This is to sign a digest of the data
        certifyInfo.type = TPM_ST_ATTEST_NV_DIGEST;
        // Initialize the hash before calling the function to add the Index data to
        // the hash.
        certifyInfo.attested.nvDigest.nvDigest.t.size =
            CryptHashStart(&hashState, in->inScheme.details.any.hashAlg);
        NvHashIndexData(&hashState, nvIndex, locator, 0,
            nvIndex->publicArea.dataSize);
        CryptHashEnd2B(&hashState, &certifyInfo.attested.nvDigest.nvDigest.b);
    }
    // Sign attestation structure.  A NULL signature will be returned if
    // signObject is NULL.
    return SignAttestInfo(signObject, &in->inScheme, &certifyInfo,
                          &in->qualifyingData, &out->certifyInfo, &out->signature);
}
#endif // CC_NV_Certify
```

32 Attached Components

32.1 Introduction

This section contains commands that allow interaction with an Attached Component (AC).

NOTE The Attached Component feature was added in revision 01.40.

32.2 TPM2_AC_GetCapability

32.2.1 General Description

The purpose of this command is to obtain information about an Attached Component referenced by an AC handle.

The returned list contains 0 or more values starting at the first tagged value that is equal to or greater than *capability*.

The list returned in *capabilitiesData* contains tagged values that indicate the type of the value.

The TPM will return the lesser of a) the available values, b) the number requested in *count,* or c) the number that will fit within the available response buffer. If additional values with higher *capability* numbers are available, *moreData* will be YES.

NOTE TPM2_AC_GetCapability() was added in revision 01.40.

32.2.2 Command and Response

Table 240 — TPM2_AC_GetCapability Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_AC_GetCapability
TPMI_RH_AC	ac	handle indicating the Attached Component Auth Index: None
TPM_AT	capability	starting info type
UINT32	count	maximum number of values to return

Table 241 — TPM2_AC_GetCapability Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	.
TPMI_YES_NO	moreData	flag to indicate whether there are more values
TPML_AC_CAPABILITIES	capabilitiesData	list of capabilities

32.2.3 Detailed Actions

```
#include "Tpm.h"
#include "AC_GetCapability_fp.h"
#include "AC_spt_fp.h"
#if CC_AC_GetCapability  // Conditional expansion of this file
TPM_RC
TPM2_AC_GetCapability(
    AC_GetCapability_In    *in,            // IN: input parameter list
    AC_GetCapability_Out   *out            // OUT: output parameter list
    )
{
// Command Output
    out->moreData = AcCapabilitiesGet(in->ac, in->count, &out->capabilitiesData);

    return TPM_RC_SUCCESS;
}
#endif // CC_AC_GetCapability
```

32.3 TPM2_AC_Send

32.3.1 General Description

The purpose of this command is to send (copy) a loaded object from the TPM to an Attached Component.

The Object referenced by *sendObject* is required to have *fixedTpm, fixedParent,* and *encryptedDuplication* attributes CLEAR (TPM_RC_ATTRIBUTES). Authorization for *sendObject* is required to be a policy session. The *policySession→commandCode* of the policy session context is required to be TPM_CC_AC_Send (TPM_RC_POLICY_FAIL) to demonstrate that the policy is specific for this command.

Authorization to send to the *ac* is provided by the session associated with *authHandle*.

If an NV Alias is not defined for *ac,* then *authHandle* is required to be either TPM_RH_OWNER or TPM_RH_PLATFORM (TPM_RC_HANDLE).

If an NV Alias is defined for *ac*, then the authorization for *authHandle* is required to be compatible with the write authorization attributes (TPMA_NV_PPWRITE, TPMA_NV_OWNERWRITE TPMA_NV_AUTHWRITE, and TPMA_NV_POLICYWRITE) in the NV Alias (TPM_RC_NV_AUTHORIZATION).

NOTE 1 If authorization for *authHandle* is the handle of an NV Index, then it is required to be the NV Alias value for *ac* (TPM_RC_NV_AUTHORIZATION).

If authorization succeeds, the TPM will attempt to send *acDataIn* and relevant portions of *sendObject* to the AC referenced by *ac*.

The TPM will return TPM_RC_SUCCESS if it succeeds in performing all the required authorizations and validations. If problems occur in the process of sending the object from the TPM to the AC, the response code will be TPM_RC_SUCCESS with the AC-dependent error reported in acDataOut.

NOTE 2 TPM2_AC_Send() was added in revision 01.40.

32.3.2 Command and Response

Table 242 — TPM2_AC_Send Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	Tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_AC_Send
TPMI_DH_OBJECT	@sendObject	handle of the object being sent to ac Auth Index: 1 Auth Role: DUP
TPMI_RH_NV_AUTH	@authHandle	the handle indicating the source of the authorization value Auth Index: 2 Auth Role: USER
TPMI_RH_AC	ac	handle indicating the Attached Component to which the object will be sent Auth Index: None
TPM2B_MAX_BUFFER	acDataIn	Optional non sensitive information related to the object

Table 243 — TPM2_AC_Send Response

Type	Name	Description
TPM_ST	Tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	
TPMS_AC_OUTPUT	acDataOut	May include AC specific data or information about an error.

32.3.3 Detailed Actions

```
#include "Tpm.h"
#include "AC_Send_fp.h"
#include "AC_spt_fp.h"
#if CC_AC_Send  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_ATTRIBUTES	key to duplicate has *fixedParent* SET
TPM_RC_HASH	for an RSA key, the *nameAlg* digest size for the *newParent* is not compatible with the key size
TPM_RC_HIERARCHY	*encryptedDuplication* is SET and *newParentHandle* specifies Null Hierarchy
TPM_RC_KEY	*newParentHandle* references invalid ECC key (public point not on the curve)
TPM_RC_SIZE	input encryption key size does not match the size specified in symmetric algorithm
TPM_RC_SYMMETRIC	*encryptedDuplication* is SET but no symmetric algorithm is provided
TPM_RC_TYPE	*newParentHandle* is neither a storage key nor TPM_RH_NULL; or the object has a NULL *nameAlg*
TPM_RC_VALUE	for an RSA *newParent*, the sizes of the digest and the encryption key are too large to be OAEP encoded

```
TPM_RC
TPM2_AC_Send(
    AC_Send_In    *in,              // IN: input parameter list
    AC_Send_Out   *out              // OUT: output parameter list
)
{
    NV_REF           locator;
    TPM_HANDLE       nvAlias = ((in->ac - AC_FIRST) + NV_AC_FIRST);
    NV_INDEX        *nvIndex = NvGetIndexInfo(nvAlias, &locator);
    OBJECT          *object = HandleToObject(in->sendObject);
    TPM_RC           result;
// Input validation
    // If there is an NV alias, then the index must allow the authorization provided
    if(nvIndex != NULL)
    {
        // Common access checks, NvWriteAccessCheck() may return
        // TPM_RC_NV_AUTHORIZATION or TPM_RC_NV_LOCKED
        result = NvWriteAccessChecks(in->authHandle, nvAlias,
                                     nvIndex->publicArea.attributes);
        if(result != TPM_RC_SUCCESS)
            return result;
    }
    // If 'ac' did not have an alias then the authorization had to be with either
    // platform or owner authorization. The type of TPMI_RH_NV_AUTH only allows
    // owner or platform or an NV index. If it was a valid index, it would have had
    // an alias and be processed above, so only success here is if this is a
    // permanent handle.
    else if(HandleGetType(in->authHandle) != TPM_HT_PERMANENT)
        return TPM_RCS_HANDLE + RC_AC_Send_authHandle;
    // Make sure that the object to be duplicated has the right attributes
    if(IS_ATTRIBUTE(object->publicArea.objectAttributes,
                    TPMA_OBJECT, encryptedDuplication)
       || IS_ATTRIBUTE(object->publicArea.objectAttributes, TPMA_OBJECT,
                       fixedParent)
```

```
        || IS_ATTRIBUTE(object->publicArea.objectAttributes, TPMA_OBJECT, fixedTPM))
         return TPM_RCS_ATTRIBUTES + RC_AC_Send_sendObject;
// Command output
    // Do the implementation dependent send
    return AcSendObject(in->ac, object, &out->acDataOut);
}
#endif // TPM_CC_AC_Send
```


32.4 TPM2_Policy_AC_SendSelect

32.4.1 General Description

This command allows qualification of the sending (copying) of an Object to an Attached Component (AC). Qualification includes selection of the receiving AC and the method of authentication for the AC, and, in certain circumstances, the Object to be sent may be specified.

If this command is not used in conjunction with TPM2_PolicyAuthorize(), then only the *authHandleName* and *acName* are selected and *includeObject* should be CLEAR.

NOTE 1 In the absence of TPM2_PolicyAuthorize(), a policy session cannot create a *policyDigest* that simultaneously equals the *authPolicy* in an Object and names that Object. This is because the *authPolicy* recorded in an Object is unable to include the Name of the Object as the Name of an Object depends on the Object's *authPolicy*.

NOTE 2 An object's *authPolicy* can incorporate the use of TPM2_PolicyAuthorize(). If the authorizing entity for the TPM2_PolicyAuthorize() command specifies only the *ac* and the *authHandle*, then the resultant *policyDigest* may be applied to the sending of any number of Objects. If the authorizing entity for the TPM2_PolicyAuthorize() specifies also the Name of the Object to be sent, then the resultant *policyDigest* applies only to that specific Object.

If either policySession→cpHash or policySession→nameHash has been previously set, the TPM shall return TPM_RC_CPHASH. Otherwise, policySession→nameHash will be set to:

$$nameHash \coloneqq \mathbf{H}_{policyAlg}(objectName \,||\, authHandleName \,||\, acName) \quad (42)$$

NOTE 3 A policy cannot specify both *cpHash* and *nameHash* because *policySession→nameHash* and *policySession→cpHash* may share the same memory space.

If the command succeeds, *policySession→policyDigest* will be updated according to the setting of the input parameter *includeObject*. If *includeObject* is SET, *policySession→policyDigest* is updated by:

$$policyDigest_{new} \coloneqq \mathbf{H}_{policyAlg}(policyDigest_{old} \,||\, \text{TPM\_CC\_Policy\_AC\_SendSelect} \,||\, objectName \,||\, authHandleName \,||\, acName \,||\, includeObject) \quad (43)$$

but if includeObject is CLEAR, *policySession→policyDigest* is updated by:

$$policyDigest_{new} \coloneqq \mathbf{H}_{policyAlg}(policyDigest_{old} \,||\, \text{TPM\_CC\_Policy\_AC\_SendSelect} \,||\, authHandleName \,||\, acName \,||\, includeObject) \quad (44)$$

NOTE 4 *policySession→nameHash* receives the digest of all Names so that the check performed in TPM2_AC_Send() may be the same regardless of which Names are included in *policySession→policyDigest*. This means that, when TPM2_Policy_AC_SendSelect() is executed, it is only valid for a specific triple of *objectName*, *authHandleName*, and *acName*.

If the command succeeds, *policySession→commandCode* is set to TPM_CC_AC_Send.

NOTE 5 The normal use of TPM2_Policy_AC_SendSelect() is before a TPM2_PolicyAuthorize(). An authorized entity would approve a *policyDigest* that allows sending to a specific Attached Component. The authorizing entity may want to limit the authorization so that the approval allows only a specific Object to be sent to the Attached Component. In that case, the authorizing entity would approve the *policyDigest* of *equation* (44).

NOTE 6 TPM2_Policy_AC_SendSelect() was added in revision 01.40.

32.4.2 Command and Response

Table 244 — TPM2_Policy_AC_SendSelect Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	Tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_Policy_AC_SendSelect
TPMI_SH_POLICY	policySession	handle for the policy session being extended Auth Index: None
TPM2B_NAME	objectName	the Name of the Object to be sent
TPM2B_NAME	authHandleName	the Name associated with *authHandle* used in the TPM2_AC_Send() command
TPM2B_NAME	acName	the Name of the Attached Component to which the Object will be sent
TPMI_YES_NO	includeObject	if SET, *objectName* will be included in the value in *policySession→policyDigest*

Table 245 — TPM2_Policy_AC_SendSelect Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

32.4.3 Detailed Actions

```c
#include "Tpm.h"
#include "Policy_AC_SendSelect_fp.h"
#if CC_Policy_AC_SendSelect      // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_COMMAND_CODE	*commandCode* of '*policySession*; is not empty
TPM_RC_CPHASH	*cpHash* of *policySession* is not empty

```c
TPM_RC
TPM2_Policy_AC_SendSelect(
    Policy_AC_SendSelect_In  *in                  // IN: input parameter list
    )
{
    SESSION         *session;
    HASH_STATE      hashState;
    TPM_CC          commandCode = TPM_CC_Policy_AC_SendSelect;

// Input Validation

    // Get pointer to the session structure
    session = SessionGet(in->policySession);

    // cpHash in session context must be empty
    if(session->u1.cpHash.t.size != 0)
        return TPM_RC_CPHASH;
    // commandCode in session context must be empty
    if(session->commandCode != 0)
        return TPM_RC_COMMAND_CODE;
// Internal Data Update
    // Update name hash
    session->u1.cpHash.t.size = CryptHashStart(&hashState, session->authHashAlg);

    //  add objectName
    CryptDigestUpdate2B(&hashState, &in->objectName.b);

    // add authHandleName
    CryptDigestUpdate2B(&hashState, &in->authHandleName.b);

    //  add ac name
    CryptDigestUpdate2B(&hashState, &in->acName.b);

    //  complete hash
    CryptHashEnd2B(&hashState, &session->u1.cpHash.b);

    // update policy hash
    // Old policyDigest size should be the same as the new policyDigest size since
    // they are using the same hash algorithm
    session->u2.policyDigest.t.size
        = CryptHashStart(&hashState, session->authHashAlg);
//  add old policy
    CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);

    //  add command code
    CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), commandCode);

    //  add objectName
    if(in->includeObject == YES)
        CryptDigestUpdate2B(&hashState, &in->objectName.b);

```

```
    //  add authHandleName
    CryptDigestUpdate2B(&hashState, &in->authHandleName.b);

    // add acName
    CryptDigestUpdate2B(&hashState, &in->acName.b);

    //  add includeObject
    CryptDigestUpdateInt(&hashState, sizeof(TPMI_YES_NO), in->includeObject);

    //  complete digest
    CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);

    // set commandCode in session context
    session->commandCode = TPM_CC_AC_Send;

    return TPM_RC_SUCCESS;
}
#endif // CC_Policy_AC_SendSelect
```


33 Authenticated Countdown Timer

33.1 Introduction

This section contains commands that allow interaction with an Authenticated Countdown Timer (ACT).

NOTE The Authenticated Countdown Timer was added in revision 01.56.

33.2 TPM2_ACT_SetTimeout

33.2.1 General Description

This command is used to set the time remaining before an Authenticated Countdown Timer (ACT) expires.

This command sets TPMS_ACT_DATA.*timeout* (ACT Timeout) to *startTimeout*. The *startTimeout* value is an integer number of seconds and may be zero. The *startTimeout* parameter may be greater, equal, or less than the current value of ACT Timeout.

When ACT Timeout is non-zero, it will count down, once per second until it reaches zero, at which time the *signaled* attribute of the TPMA_ACT associated with *actHandle* is SET.

When ACT Timeout is zero and the *signaled* attribute is SET, writing a *startTimeout* of FF FF FF FF_{16} will clear *signaled* and stop the counting.

There are four states for ACT Timeout and *startTimeout*. The *signaled* attribute will be set as follows:

1) If ACT Timeout is zero and *startTimeout* is non-zero, then *signaled* will be CLEAR.

2) If ACT Timeout is non-zero and *startTimeout* is non-zero, then *signaled* will be CLEAR.

3) If ACT Timeout is zero and *startTimeout* is zero, then *signaled* will be unchanged.

4) If ACT Timeout is non-zero and *startTimeout* is zero, then *signaled* will be SET.

NOTE 1 The ACT signals on a transition from non-zero to zero. The transition can occur either due to TPM2_ACT_SetTimeout() or a decrement. The effect of *signaled* is platform dependent.

NOTE 2 It may take up to one second until ACT Timeout will be set and *signaled* will be CLEAR or SET by TPM2_ACT_SetTimeout() or TPM2_Startup(STATE). This allows the counting and signaling to take place synchronously with the hardware clock tick.

NOTE 3 TPM2_ACT_SetTimeout() was added in revision 01.56.

33.2.2 Command and Response

Table 246 — TPM2_ACT_SetTimeout Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_ACT_SetTimeout
TPMI_RH_ACT	@actHandle	Handle of the selected ACT Auth Index: 1 Auth Role: USER
UINT32	startTimeout	the start timeout value for the ACT in seconds

Table 247 — TPM2_ACT_SetTimeout Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	

33.2.3 Detailed Actions

```
#include "Tpm.h"
#include "ACT_SetTimeout_fp.h"
#if CC_ACT_SetTimeout  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_RETRY	returned when an update for the selected ACT is already pending
TPM_RC_VALUE	attempt to disable signaling from an ACT that has not expired

```
TPM_RC
TPM2_ACT_SetTimeout(
    ACT_SetTimeout_In      *in             // IN: input parameter list
    )
{
    // If 'startTimeout' is UINT32_MAX, then this is an attempt to disable the ACT
    // and turn off the signaling for the ACT. This is only valid if the ACT
    // is signaling.
    if((in->startTimeout == UINT32_MAX) && !ActGetSignaled(in->actHandle))
        return TPM_RC_VALUE + RC_ACT_SetTimeout_startTimeout;
    return ActCounterUpdate(in->actHandle, in->startTimeout);
}
#endif // CC_ACT_SetTimeout
```


34 Vendor Specific

34.1 Introduction

This section contains commands that are vendor specific but made public in order to prevent proliferation.

This specification does define TPM2_Vendor_TCG_Test() in order to have at least one command that can be used to ensure the proper operation of the command dispatch code when processing a vendor-specific command.

34.2 TPM2_Vendor_TCG_Test

34.2.1 General Description

This is a placeholder to allow testing of the dispatch code.

34.2.2 Command and Response

Table 248 — TPM2_Vendor_TCG_Test Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_Vendor_TCG_Test
TPM2B_DATA	inputData	dummy data

Table 249 — TPM2_Vendor_TCG_Test Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	TPM_RC_SUCCESS
TPM2B_DATA	outputData	dummy data

34.2.3 Detailed Actions

```
#include "Tpm.h"
#if CC_Vendor_TCG_Test      // Conditional expansion of this file
#include "Vendor_TCG_Test_fp.h"
TPM_RC
TPM2_Vendor_TCG_Test(
    Vendor_TCG_Test_In        *in,            // IN: input parameter list
    Vendor_TCG_Test_Out       *out            // OUT: output parameter list
    )
{
    out->outputData = in->inputData;
    return TPM_RC_SUCCESS;
}
#endif // CC_Vendor_TCG_Test
```

