Trusted Platform Module Library

Part 3: Commands

Family "2.0"

Level 00 Revision 01.36

September 16, 2016

Committee Draft

Contact: admin@trustedcomputinggroup.org

Work in Progress: This document is an intermediate draft for comment only and is subject to change without notice. Readers should not design products based on this document.

TCG Public Review

TCG

Licenses and Notices

Copyright Licenses:

- Trusted Computing Group (TCG) grants to the user of the source code in this specification (the "Source Code") a worldwide, irrevocable, nonexclusive, royalty free, copyright license to reproduce, create derivative works, distribute, display and perform the Source Code and derivative works thereof, and to grant others the rights granted herein.

- The TCG grants to the user of the other parts of the specification (other than the Source Code) the rights to reproduce, distribute, display, and perform the specification solely for the purpose of developing products based on such documents.

Source Code Distribution Conditions:

- Redistributions of Source Code must retain the above copyright licenses, this list of conditions and the following disclaimers.

- Redistributions in binary form must reproduce the above copyright licenses, this list of conditions and the following disclaimers in the documentation and/or other materials provided with the distribution.

Disclaimers:

- THE COPYRIGHT LICENSES SET FORTH ABOVE DO NOT REPRESENT ANY FORM OF LICENSE OR WAIVER, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, WITH RESPECT TO PATENT RIGHTS HELD BY TCG MEMBERS (OR OTHER THIRD PARTIES) THAT MAY BE NECESSARY TO IMPLEMENT THIS SPECIFICATION OR OTHERWISE. Contact TCG Administration (admin@trustedcomputinggroup.org) for information on specification licensing rights available through TCG membership agreements.

- THIS SPECIFICATION IS PROVIDED "AS IS" WITH NO EXPRESS OR IMPLIED WARRANTIES WHATSOEVER, INCLUDING ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE, ACCURACY, COMPLETENESS, OR NONINFRINGEMENT OF INTELLECTUAL PROPERTY RIGHTS, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION OR SAMPLE.

- Without limitation, TCG and its members and licensors disclaim all liability, including liability for infringement of any proprietary rights, relating to use of information in this specification and to the implementation of this specification, and TCG disclaims all liability for cost of procurement of substitute goods or services, lost profits, loss of use, loss of data or any incidental, consequential, direct, indirect, or special damages, whether under contract, tort, warranty or otherwise, arising in any way out of use or reliance upon this specification or any information herein.

Any marks and brands contained herein are the property of their respective owners.

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 "Conditional Types" in TPM 2.0 Part 2). The null value is usually TPM_RH_NULL for a handle or TPM_ALG_NULL for an algorithm selector.
@	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().

4.2.1 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

4.3 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.

4.4 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_LOCKED)

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

```
1   #include "Tpm.h"
2   #include "_TPM_Init_fp.h"
3    // This function is used to process a _TPM_Init indication.
4   LIB_EXPORT void
5   _TPM_Init(
6       void
7       )
8   {
9       g_powerWasLost = g_powerWasLost | _plat__WasPowerLost();
10  #if defined SIMULATION && !defined NDEBUG
11      // If power was lost and this was a simulation, put canary in RAM used by NV
12      // so that uninitialized memory can be detected more easily
13      if(g_powerWasLost)
14      {
15          memset(&gc, 0xbb, sizeof(gc));
16          memset(&gr, 0xbb, sizeof(gr));
17          memset(&gp, 0xbb, sizeof(gp));
18          memset(&go, 0xbb, sizeof(go));
19      }
20  #endif
21  #ifdef SIMULATION
22      // Clear the flag that forces failure on self-test
23      g_forceFailureMode = FALSE;
24  #endif
25      // Set initialization state
26      TPMInit();
27      // Set g_DRTMHandle as unassigned
28      g_DRTMHandle = TPM_RH_UNASSIGNED;
29      // No H-CRTM, yet.
30      g_DrtmPreStartup = FALSE;
31      // Initialize the NvEnvironment.
32      g_nvOk = NvPowerOn();
33      // Initialize cryptographic functions
34      g_inFailureMode = (CryptInit() == FALSE);
35      if(!g_inFailureMode)
36      {
37          // Load the persistent data
38          NvReadPersistent();
39          // Load the orderly data (clock and DRBG state).
40          // If this is not done here, things break
41          NvRead(&go, NV_ORDERLY_DATA, sizeof(go));
42          // Start clock. Need to do this after NV has been restored.
43          TimePowerOn();
44      }
45      return;
46  }
```

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;

- TPMS_TIME_INFO.*time* shall be reset to zero; and

NOTE 4   If time is reset to zero at _TPM_Init, it need not be reset again here.

- 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_WRITE_DEFINE 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,

- 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.

- *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*)

    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,

- *shEnable* and *ehEnable* shall be SET,

- *platformAuth* and *platformPolicy* shall be set to the Empty Buffer,

- For each NV index with TPMA_NV_WRITE_DEFINE 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.

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.*

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

```
1   #include "Tpm.h"
2   #include "Startup_fp.h"
3   #ifdef TPM_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

```
4   TPM_RC
5   TPM2_Startup(
6       Startup_In      *in               // IN: input parameter list
7       )
8   {
9       STARTUP_TYPE         startup;
10      BYTE                 locality = _plat__LocalityGet();
11      // The command needs NV update.
12      RETURN_IF_NV_IS_NOT_AVAILABLE;
13      // Get the flags for the current startup locality and the H-CRTM.
14      // Rather than generalizing the locality setting, this code takes advantage
15      // of the fact that the PC Client specification only allows Startup()
16      // from locality 0 and 3. To generalize this probably would require a
17      // redo of the NV space and since this is a feature that is hardly ever used
18      // outside of the PC Client, this code just support the PC Client needs.
19  // Input Validation
20      // Check that the locality is a supported value
21      if(locality != 0 && locality != 3)
22          return TPM_RC_LOCALITY;
23      // If there was a H-CRTM, then treat the locality as being 3
24      // regardless of what the Startup() was. This is done to preserve the
25      // H-CRTM PCR so that they don't get overwritten with the normal
26      // PCR startup initialization. This basically means that g_StartupLocality3
27      // and g_DrtmPreStartup can't both be SET at the same time.
28      if(g_DrtmPreStartup)
29          locality = 0;
30      g_StartupLocality3 = (locality == 3);
31  #ifdef USE_DA_USED
32      // If there was no orderly shutdown, then their might have been a write to
33      // failedTries that didn't get recorded but only if DA_USES was SET in the
34      // shutdown state
35      g_daUsed = (gp.orderlyState == SU_DA_USED_VALUE);
36      if(g_daUsed)
37          gp.orderlyState = SU_NONE_VALUE;
38  #endif
39      g_prevOrderlyState = gp.orderlyState;
40      // If this is a Resume,
41      if(in->startupType == TPM_SU_STATE)
42      {
43          // Turn of the startup modifiers in the recovered state. This will modify
44          // the SU_NONE_VALUE but not make it anything that would be recognized as
45          // a valid shutdown
46          g_prevOrderlyState &= ~(PRE_STARTUP_FLAG | STARTUP_LOCALITY_3);
47          // then there must have been a prior TPM2_ShutdownState(STATE)
48          if(g_prevOrderlyState != TPM_SU_STATE)
49              return TPM_RC_VALUE + RC_Startup_startupType;
50          // and the part of NV used for state save must have been recovered
51          // correctly.
52          // NOTE: if this fails, then the caller will need to do Startup(CLEAR). The
```

```
53              // code for Startup(Clear) cannot fail if the NV can't be read correctly
54              // because that would prevent the TPM from ever getting unstuck.
55              if(g_nvOk == FALSE)
56                  return TPM_RC_NV_UNINITIALIZED;
57              // For Resume, the H-CRTM has to be the same as the previous boot
58              if(g_DrtmPreStartup != ((gp.orderlyState & PRE_STARTUP_FLAG) != 0))
59                  return TPM_RC_VALUE + RC_Startup_startupType;
60              if(g_StartupLocality3 != ((gp.orderlyState & STARTUP_LOCALITY_3) != 0))
61                  return TPM_RC_LOCALITY;
62              gp.orderlyState = g_prevOrderlyState;
63          }
64      // Internal Date Update
65          if((gp.orderlyState == TPM_SU_STATE) && (g_nvOk == TRUE))
66          {
67              // Always read the data that is only cleared on a Reset because this is not
68              // a reset
69              NvRead(&gr, NV_STATE_RESET_DATA, sizeof(gr));
70              if(in->startupType == TPM_SU_STATE)
71              {
72                  // If this is a startup STATE (a Resume) need to read the data
73                  // that is cleared on a startup CLEAR because this is not a Reset
74                  // or Restart.
75                  NvRead(&gc, NV_STATE_CLEAR_DATA, sizeof(gc));
76                  startup = SU_RESUME;
77              }
78              else
79                  startup = SU_RESTART;
80          }
81      else
82              // Will do a TPM reset if Shutdown(CLEAR) and Startup(CLEAR) or no shutdown
83              // or there was a failure reading the NV data.
84              startup = SU_RESET;
85      // Startup for cryptographic library. Don't do this until after the orderly
86      // state has been read in from NV.
87      CryptStartup(startup);
88      // Read the platform unique value that is used as VENDOR_PERMANENT
89      // authorization value
90      g_platformUniqueDetails.t.size
91          = (UINT16)_plat__GetUnique(1, sizeof(g_platformUniqueDetails.t.buffer),
92                                  g_platformUniqueDetails.t.buffer);
93  // Start up subsystems
94      // Start set the safe flag
95      TimeStartup(startup);
96      // Start dictionary attack subsystem
97      DAStartup(startup);
98      // Enable hierarchies
99      HierarchyStartup(startup);
100     // Restore/Initialize PCR
101     PCRStartup(startup, locality);
102     // Restore/Initialize command audit information
103     CommandAuditStartup(startup);
104 //// The following code was moved from Time.c where it made no sense
105     switch (startup)
106     {
107         case SU_RESUME:
108             // Resume sequence
109             gr.restartCount++;
110             break;
111         case SU_RESTART:
112             // Hibernate sequence
113             gr.clearCount++;
114             gr.restartCount++;
115             break;
116         default:
117             // Reset object context ID to 0
118             gr.objectContextID = 0;
```

```
119             // Reset clearCount to 0
120             gr.clearCount = 0;
121             // Reset sequence
122             // Increase resetCount
123             gp.resetCount++;
124             // Write resetCount to NV
125             NV_SYNC_PERSISTENT(resetCount);
126             gp.totalResetCount++;
127             // We do not expect the total reset counter overflow during the life
128             // time of TPM.  if it ever happens, TPM will be put to failure mode
129             // and there is no way to recover it.
130             // The reason that there is no recovery is that we don't increment
131             // the NV totalResetCount when incrementing would make it 0. When the
132             // TPM starts up again, the old value of totalResetCount will be read
133             // and we will get right back to here with the increment failing.
134             if(gp.totalResetCount == 0)
135                 FAIL(FATAL_ERROR_INTERNAL);
136             // Write total reset counter to NV
137             NV_SYNC_PERSISTENT(totalResetCount);
138             // Reset restartCount
139             gr.restartCount = 0;
140             break;
141         }
142 ////
143     // Initialize session table
144     SessionStartup(startup);
145     // Initialize object table
146     ObjectStartup();
147     // Initialize index/evict data.  This function clears read/write locks
148     // in NV index
149     NvEntityStartup(startup);
150     // Initialize the orderly shut down flag for this cycle to SU_NONE_VALUE.
151     gp.orderlyState = SU_NONE_VALUE;
152     NV_SYNC_PERSISTENT(orderlyState);
153     // Update TPM internal states if command succeeded.
154     // Record a TPM2_Startup command has been received.
155     TPMRegisterStartup();
156     // This can be reset after the first completion of a TPM2_Startup() after
157     // a power loss. It can probably be reset earlier but this is an OK place.
158     g_powerWasLost = FALSE;
159     return TPM_RC_SUCCESS;
160 }
161 #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; and

- 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

```
1    #include "Tpm.h"
2    #include "Shutdown_fp.h"
3    #ifdef TPM_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

```
4    TPM_RC
5    TPM2_Shutdown(
6        Shutdown_In      *in              // IN: input parameter list
7        )
8    {
9        // The command needs NV update.  Check if NV is available.
10       // A TPM_RC_NV_UNAVAILABLE or TPM_RC_NV_RATE error may be returned at
11       // this point
12       RETURN_IF_NV_IS_NOT_AVAILABLE;
13   // Input Validation
14       // If PCR bank has been reconfigured, a CLEAR state save is required
15       if(g_pcrReConfig && in->shutdownType == TPM_SU_STATE)
16           return TPM_RCS_TYPE + RC_Shutdown_shutdownType;
17   // Internal Data Update
18       gp.orderlyState = in->shutdownType;
19       // PCR private date state save
20       PCRStateSave(in->shutdownType);
21       // Save RAM backed NV index data
22       NvUpdateIndexOrderlyData();
23       // Save all orderly data
24       NvWrite(NV_ORDERLY_DATA, sizeof(ORDERLY_DATA), &go);
25       if(in->shutdownType == TPM_SU_STATE)
26       {
27           // Save STATE_RESET and STATE_CLEAR data
28           NvWrite(NV_STATE_CLEAR_DATA, sizeof(STATE_CLEAR_DATA), &gc);
29           NvWrite(NV_STATE_RESET_DATA, sizeof(STATE_RESET_DATA), &gr);
30           // Save the startup flags for resume
31           if(g_DrtmPreStartup)
32               gp.orderlyState = TPM_SU_STATE | PRE_STARTUP_FLAG;
33           else if(g_StartupLocality3)
34               gp.orderlyState = TPM_SU_STATE | STARTUP_LOCALITY_3;
35       }
36       else if(in->shutdownType == TPM_SU_CLEAR)
37       {
38           // Save STATE_RESET data
39           NvWrite(NV_STATE_RESET_DATA, sizeof(STATE_RESET_DATA), &gr);
40       }
41       else
42           FAIL(FATAL_ERROR_INTERNAL);
43       NV_SYNC_PERSISTENT(orderlyState);
44       return TPM_RC_SUCCESS;
45   }
46   #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_Event() 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_Event() 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.

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).

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

```
1    #include "Tpm.h"
2    #include "SelfTest_fp.h"
3    #ifdef TPM_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

```
4    TPM_RC
5    TPM2_SelfTest(
6        SelfTest_In     *in              // IN: input parameter list
7        )
8    {
9    // Command Output
10       // Call self test function in crypt module
11       return CryptSelfTest(in->fullTest);
12   }
13   #endif // CC_SelfTest
```
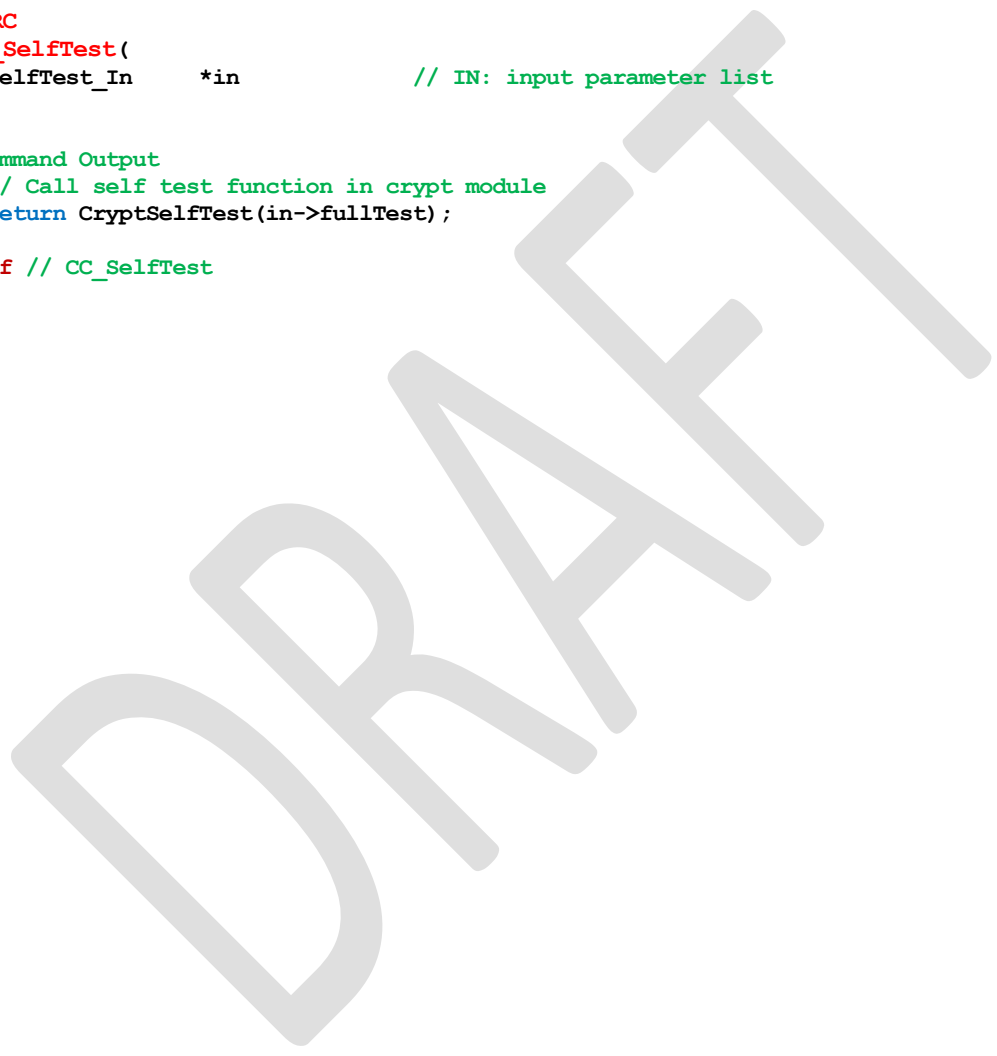
10.3   TPM2_IncrementalSelfTest

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

```
1    #include "Tpm.h"
2    #include "IncrementalSelfTest_fp.h"
3    #ifdef TPM_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

```
4    TPM_RC
5    TPM2_IncrementalSelfTest(
6        IncrementalSelfTest_In      *in,            // IN: input parameter list
7        IncrementalSelfTest_Out     *out            // OUT: output parameter list
8        )
9    {
10       TPM_RC                      result;
11   // Command Output
12       // Call incremental self test function in crypt module. If this function
13       // returns TPM_RC_VALUE, it means that an algorithm on the 'toTest' list is
14       // not implemented.
15       result = CryptIncrementalSelfTest(&in->toTest, &out->toDoList);
16       if(result == TPM_RC_VALUE)
17           return TPM_RCS_VALUE + RC_IncrementalSelfTest_toTest;
18       return result;
19   }
20   #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.

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	
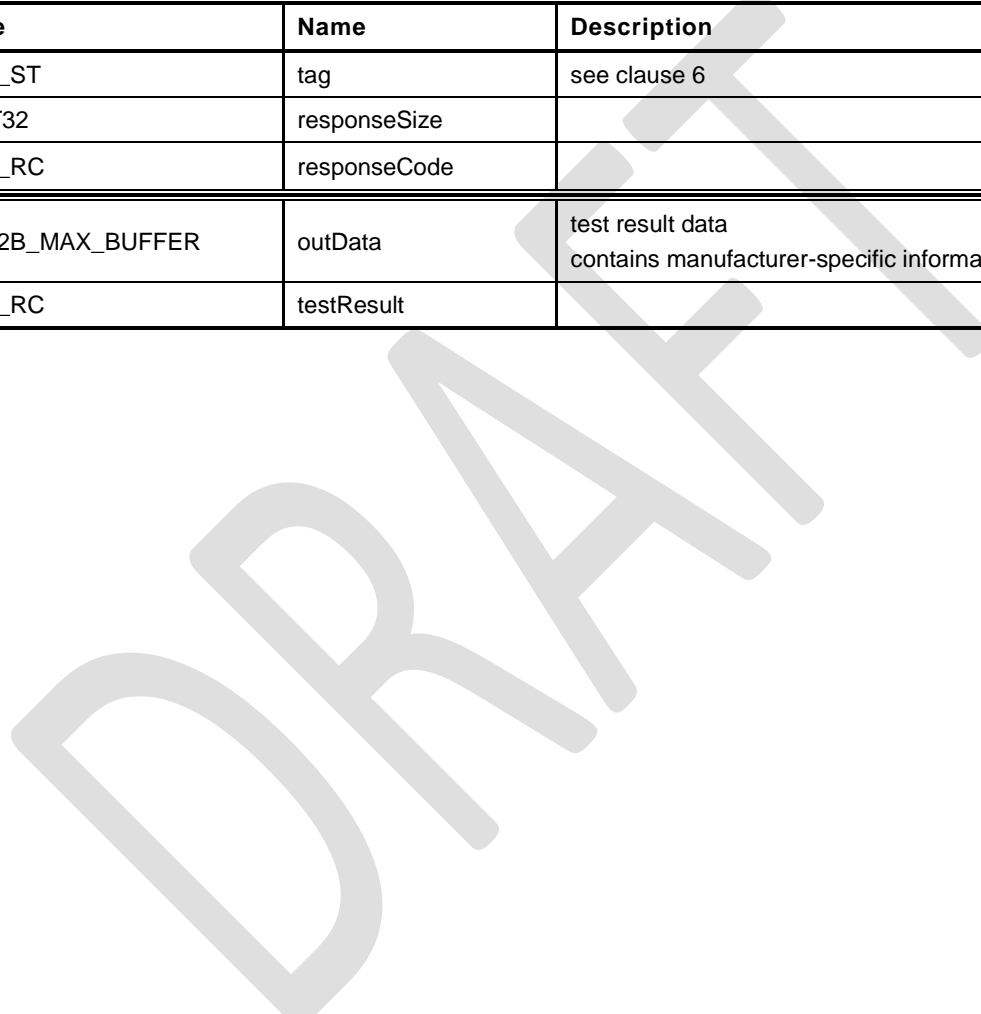
10.4.3   Detailed Actions

```
1    #include "Tpm.h"
2    #include "GetTestResult_fp.h"
3    #ifdef TPM_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.

```
4    TPM_RC
5    TPM2_GetTestResult(
6        GetTestResult_Out   *out            // OUT: output parameter list
7        )
8    {
9    // Command Output
10       // Call incremental self test function in crypt module
11       out->testResult = CryptGetTestResult(&out->outData);
12       return TPM_RC_SUCCESS;
13   }
14   #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_SCHEME if the scheme of the key is not TPM_ALG_OAEP or TPM_ALG_NULL.

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

```
1    #include "Tpm.h"
2    #include "StartAuthSession_fp.h"
3    #ifdef TPM_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 exponent of *tpmKey*.

```
4    TPM_RC
5    TPM2_StartAuthSession(
6        StartAuthSession_In     *in,            // IN: input parameter buffer
7        StartAuthSession_Out    *out            // OUT: output parameter buffer
8        )
9    {
10       TPM_RC                   result = TPM_RC_SUCCESS;
11       OBJECT                  *tpmKey;                // TPM key for decrypt salt
12       TPM2B_DATA               salt;
13   // Input Validation
14       // Check input nonce size.  IT should be at least 16 bytes but not larger
15       // than the digest size of session hash.
16       if(in->nonceCaller.t.size < 16
17          || in->nonceCaller.t.size > CryptHashGetDigestSize(in->authHash))
18           return TPM_RCS_SIZE + RC_StartAuthSession_nonceCaller;
19       // If an decrypt key is passed in, check its validation
20       if(in->tpmKey != TPM_RH_NULL)
21       {
22           // Get pointer to loaded decrypt key
23           tpmKey = HandleToObject(in->tpmKey);
24           // key must be asymmetric with its sensitive area loaded. Since this
25           // command does not require authorization, the presence of the sensitive
26           // area was not already checked as it is with most other commands that
27           // use the sensitive are so check it here
28           if(!CryptIsAsymAlgorithm(tpmKey->publicArea.type))
29               return TPM_RCS_KEY + RC_StartAuthSession_tpmKey;
30           // secret size cannot be 0
31           if(in->encryptedSalt.t.size == 0)
32               return TPM_RCS_VALUE + RC_StartAuthSession_encryptedSalt;
33           // Decrypting salt requires accessing the private portion of a key.
34           // Therefore, tmpKey can not be a key with only public portion loaded
35           if(tpmKey->attributes.publicOnly)
36               return TPM_RCS_HANDLE + RC_StartAuthSession_tpmKey;
37           // HMAC session input handle check.
38           // tpmKey should be a decryption key
39           if(tpmKey->publicArea.objectAttributes.decrypt != SET)
```

```
40                return TPM_RCS_ATTRIBUTES + RC_StartAuthSession_tpmKey;
41            // Secret Decryption.  A TPM_RC_VALUE, TPM_RC_KEY or Unmarshal errors
42            // may be returned at this point
43            result = CryptSecretDecrypt(tpmKey, &in->nonceCaller, SECRET_KEY,
44                                    &in->encryptedSalt, &salt);
45            if(result != TPM_RC_SUCCESS)
46                return TPM_RCS_VALUE + RC_StartAuthSession_encryptedSalt;
47        }
48        else
49        {
50            // secret size must be 0
51            if(in->encryptedSalt.t.size != 0)
52                return TPM_RCS_VALUE + RC_StartAuthSession_encryptedSalt;
53            salt.t.size = 0;
54        }
55        switch(HandleGetType(in->bind))
56        {
57            case TPM_HT_TRANSIENT:
58            {
59                OBJECT       *object = HandleToObject(in->bind);
60                // If the bind handle references a transient object, make sure that we
61                // can get to the authorization value. Also, make sure that the object
62                // has a proper Name (nameAlg != TPM_ALG_NULL). If it doesn't, then
63                // it might be possible to bind to an object where the authValue is
64                // known. This does not create a real issue in that, if you know the
65                // authorization value, you can actually bind to the object. However,
66                // there is a potential
67                if(object->attributes.publicOnly == SET)
68                    return TPM_RCS_HANDLE + RC_StartAuthSession_bind;
69                break;
70            }
71            case TPM_HT_NV_INDEX:
72            // a PIN index can't be a bind object
73            {
74                NV_INDEX        *nvIndex = NvGetIndexInfo(in->bind, NULL);
75                if(IsNvPinPassIndex(nvIndex->publicArea.attributes)
76                   || IsNvPinFailIndex(nvIndex->publicArea.attributes))
77                    return TPM_RCS_HANDLE + RC_StartAuthSession_bind;
78                break;
79            }
80            default:
81                break;
82        }
83        // If 'symmetric' is a symmetric block cipher (not TPM_ALG_NULL or TPM_ALG_XOR)
84        // then the mode must be CFB.
85        if(in->symmetric.algorithm != TPM_ALG_NULL
86           && in->symmetric.algorithm != TPM_ALG_XOR
87           && in->symmetric.mode.sym != TPM_ALG_CFB)
88            return TPM_RCS_MODE + RC_StartAuthSession_symmetric;
89    // Internal Data Update and command output
90        // Create internal session structure.  TPM_RC_CONTEXT_GAP, TPM_RC_NO_HANDLES
91        // or TPM_RC_SESSION_MEMORY errors may be returned at this point.
92        //
93        // The detailed actions for creating the session context are not shown here
94        // as the details are implementation dependent
95        // SessionCreate sets the output handle and nonceTPM
96        result = SessionCreate(in->sessionType, in->authHash, &in->nonceCaller,
97                            &in->symmetric, in->bind, &salt, &out->sessionHandle,
98                            &out->nonceTPM);
99        return result;
100    }
101    #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

```
1    #include "Tpm.h"
2    #include "PolicyRestart_fp.h"
3    #ifdef TPM_CC_PolicyRestart  // Conditional expansion of this file
4    TPM_RC
5    TPM2_PolicyRestart(
6        PolicyRestart_In    *in               // IN: input parameter list
7        )
8    {
9        // Initialize policy session data
10       SessionResetPolicyData(SessionGet(in->sessionHandle));
11       return TPM_RC_SUCCESS;
12   }
13   #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 objects 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 this 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.

$$unique := \mathbf{H}_{nameAlg}(sensitive.seedValue.buffer \,||\, sensitive.any.buffer) \tag{1}$$

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.

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.

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 5   The size of inSensitive.sensitive.data is limited to be no larger than the largest value of TPMT_SENSITIVE.sensitive.bits by 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 6   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 7   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.

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_TEMPLATE	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

```
1    #include "Tpm.h"
2    #include "Object_spt_fp.h"
3    #include "Create_fp.h"
4    #ifdef TPM_CC_Create  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_ASYMMETRIC	non-duplicable storage key and its parent have different public parameters
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*, *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; non-duplicable storage key and its parent have different types; *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

```
5    TPM_RC
6    TPM2_Create(
7        Create_In        *in,            // IN: input parameter list
8        Create_Out       *out            // OUT: output parameter list
9        )
10   {
11       TPM_RC                  result = TPM_RC_SUCCESS;
12       OBJECT                  *parentObject;
13       OBJECT                  *newObject;
14       TPMT_PUBLIC             *publicArea;
15   // Input Validation
```

```
16        parentObject = HandleToObject(in->parentHandle);
17        pAssert(parentObject != NULL);
18        // Does parent have the proper attributes?
19        if(!ObjectIsParent(parentObject))
20            return TPM_RCS_TYPE + RC_Create_parentHandle;
21        // Get a slot for the creation
22        newObject = FindEmptyObjectSlot(NULL);
23        if(newObject == NULL)
24            return TPM_RC_OBJECT_MEMORY;
25        // to save typing.
26        publicArea = &newObject->publicArea;
27        // Unmarshal the public area into the allocated object
28        result = UnmarshalToPublic(publicArea, &in->inPublic, FALSE);
29        if(result != TPM_RC_SUCCESS)
30            return result + RC_Create_inPublic;
31        // The sensitiveDataOrigin attribute must be consistent with the setting of
32        // the size of the data object in inSensitive.
33        if((publicArea->objectAttributes.sensitiveDataOrigin == SET)
34           != (in->inSensitive.sensitive.data.t.size == 0))
35            // Mismatch between the object attributes and the parameter.
36            return TPM_RCS_ATTRIBUTES + RC_Create_inSensitive;
37        // Check attributes in input public area. CreateChecks() checks the things that
38        // are unique to creation and then validates the attributes and values that are
39        // common to create and load.
40        result = CreateChecks(parentObject, publicArea);
41        if(result != TPM_RC_SUCCESS)
42            return RcSafeAddToResult(result, RC_Create_inPublic);
43        // Clean up the authValue if necessary
44        if(!AdjustAuthSize(&in->inSensitive.sensitive.userAuth, publicArea->nameAlg))
45            return TPM_RCS_SIZE + RC_Create_inSensitive;
46 // Command Output
47        // Create the object using the default TPM random-number generator
48        result = CryptCreateObject(newObject, &in->inSensitive.sensitive, NULL);
49        if(result != TPM_RC_SUCCESS)
50            return result;
51        // Fill in creation data
52        FillInCreationData(in->parentHandle, publicArea->nameAlg,
53                           &in->creationPCR, &in->outsideInfo,
54                           &out->creationData, &out->creationHash);
55        // Compute creation ticket
56        TicketComputeCreation(EntityGetHierarchy(in->parentHandle), &newObject->name,
57                           &out->creationHash, &out->creationTicket);
58        // Prepare output private data from sensitive
59        SensitiveToPrivate(&newObject->sensitive, &newObject->name.b, parentObject,
60                           publicArea->nameAlg,
61                           &out->outPrivate);
62        // Finish by copying the remaining return values
63        out->outPublic.publicArea = newObject->publicArea;
64        return TPM_RC_SUCCESS;
65    }
66 #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

```
1   #include "Tpm.h"
2   #include "Load_fp.h"
3   #ifdef TPM_CC_Load  // Conditional expansion of this file
4   #include "Object_spt_fp.h"
```

Error Returns	Meaning
TPM_RC_ASYMMETRIC	storage key with different asymmetric type than parent
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

```
5   TPM_RC
6   TPM2_Load(
7       Load_In          *in,                // IN: input parameter list
8       Load_Out         *out                // OUT: output parameter list
9       )
10  {
11      TPM_RC                    result = TPM_RC_SUCCESS;
12      TPMT_SENSITIVE            sensitive;
13      OBJECT                    *parentObject;
14      OBJECT                    *newObject;
15  // Input Validation
16      // Don't get invested in loading if there is no place to put it.
17      newObject = FindEmptyObjectSlot(&out->objectHandle);
18      if(newObject == NULL)
19          return TPM_RC_OBJECT_MEMORY;
20      if(in->inPrivate.t.size == 0)
21          return TPM_RCS_SIZE + RC_Load_inPrivate;
22      parentObject = HandleToObject(in->parentHandle);
23      pAssert(parentObject != NULL);
24      // Is the object that is being used as the parent actually a parent.
25      if(!ObjectIsParent(parentObject))
26          return TPM_RCS_TYPE + RC_Load_parentHandle;
27      // Compute the name of object. If there isn't one, it is because the nameAlg is
28      // not valid.
29      PublicMarshalAndComputeName(&in->inPublic.publicArea, &out->name);
30      if(out->name.t.size == 0)
31          return TPM_RCS_HASH + RC_Load_inPublic;
32      // Retrieve sensitive data.
```

```
33          result = PrivateToSensitive(&in->inPrivate.b, &out->name.b, parentObject,
34                                      in->inPublic.publicArea.nameAlg,
35                                      &sensitive);
36      if(result != TPM_RC_SUCCESS)
37          return RcSafeAddToResult(result, RC_Load_inPrivate);
38  // Internal Data Update
39      // Load and validate object
40      result = ObjectLoad(newObject, parentObject,
41                          &in->inPublic.publicArea, &sensitive,
42                          RC_Load_inPublic, RC_Load_inPrivate,
43                          &out->name);
44      if(result == TPM_RC_SUCCESS)
45      {
46          // Set the common OBJECT attributes for a loaded object.
47          ObjectSetLoadedAttributes(newObject, in->parentHandle);
48      }
49      return result;
50  }
51  #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 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

```
1    #include "Tpm.h"
2    #include "LoadExternal_fp.h"
3    #ifdef TPM_CC_LoadExternal  // Conditional expansion of this file
4    #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

```
5    TPM_RC
6    TPM2_LoadExternal(
7        LoadExternal_In      *in,              // IN: input parameter list
8        LoadExternal_Out     *out              // OUT: output parameter list
9        )
10   {
11       TPM_RC                result;
12       OBJECT                *object;
13       TPMT_SENSITIVE        *sensitive = NULL;
14   // Input Validation
15       // Don't get invested in loading if there is no place to put it.
16       object = FindEmptyObjectSlot(&out->objectHandle);
17       if(object == NULL)
18           return TPM_RC_OBJECT_MEMORY;
19       // If the hierarchy to be associated with this object is turned off, the object
20       // cannot be loaded.
21       if(!HierarchyIsEnabled(in->hierarchy))
22           return TPM_RCS_HIERARCHY + RC_LoadExternal_hierarchy;
23       // For loading an object with both public and sensitive
24       if(in->inPrivate.size != 0)
25       {
26           // An external object with a sensitive area can only be loaded in the
27           // NULL hierarchy
28           if(in->hierarchy != TPM_RH_NULL)
29               return TPM_RCS_HIERARCHY + RC_LoadExternal_hierarchy;
30           // An external object with a sensitive area must have fixedTPM == CLEAR
31           // fixedParent == CLEAR so that it does not appear to be a key created by
32           // this TPM.
33           if(in->inPublic.publicArea.objectAttributes.fixedTPM != CLEAR
34               || in->inPublic.publicArea.objectAttributes.fixedParent != CLEAR
35               || in->inPublic.publicArea.objectAttributes.restricted != CLEAR)
```

```
36                  return TPM_RCS_ATTRIBUTES + RC_LoadExternal_inPublic;
37              // Have sensitive point to something other than NULL so that object
38              // initialization will load the sensitive part too
39              sensitive = &in->inPrivate.sensitiveArea;
40          }
41      // Need the name to initialize the object structure
42      PublicMarshalAndComputeName(&in->inPublic.publicArea, &out->name);
43      // Load and validate key
44      result = ObjectLoad(object, NULL,
45                              &in->inPublic.publicArea, sensitive,
46                              RC_LoadExternal_inPublic, RC_LoadExternal_inPrivate,
47                              &out->name);
48      if(result == TPM_RC_SUCCESS)
49      {
50          object->attributes.external = SET;
51          // Set the common OBJECT attributes for a loaded object.
52          ObjectSetLoadedAttributes(object, in->hierarchy);
53      }
54      return result;
55  }
56  #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

```
1    #include "Tpm.h"
2    #include "ReadPublic_fp.h"
3    #ifdef TPM_CC_ReadPublic  // Conditional expansion of this file
```

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

```
4    TPM_RC
5    TPM2_ReadPublic(
6        ReadPublic_In   *in,              // IN: input parameter list
7        ReadPublic_Out  *out              // OUT: output parameter list
8        )
9    {
10       OBJECT                  *object = HandleToObject(in->objectHandle);
11   // Input Validation
12       // Can not read public area of a sequence object
13       if(ObjectIsSequence(object))
14           return TPM_RC_SEQUENCE;
15   // Command Output
16       out->outPublic.publicArea = object->publicArea;
17       out->name = object->name;
18       out->qualifiedName = object->qualifiedName;
19       return TPM_RC_SUCCESS;
20   }
21   #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

```
1   #include "Tpm.h"
2   #include "ActivateCredential_fp.h"
3   #ifdef TPM_CC_ActivateCredential  // Conditional expansion of this file
4   #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)

```
5   TPM_RC
6   TPM2_ActivateCredential(
7       ActivateCredential_In   *in,              // IN: input parameter list
8       ActivateCredential_Out  *out              // OUT: output parameter list
9       )
10  {
11      TPM_RC                   result = TPM_RC_SUCCESS;
12      OBJECT                   *object;               // decrypt key
13      OBJECT                   *activateObject;       // key associated with credential
14      TPM2B_DATA               data;                 // credential data
15  // Input Validation
16      // Get decrypt key pointer
17      object = HandleToObject(in->keyHandle);
18      // Get certificated object pointer
19      activateObject = HandleToObject(in->activateHandle);
20      // input decrypt key must be an asymmetric, restricted decryption key
21      if(!CryptIsAsymAlgorithm(object->publicArea.type)
22          || object->publicArea.objectAttributes.decrypt == CLEAR
23          || object->publicArea.objectAttributes.restricted == CLEAR)
24          return TPM_RCS_TYPE + RC_ActivateCredential_keyHandle;
25  // Command output
26      // Decrypt input credential data via asymmetric decryption.  A
27      // TPM_RC_VALUE, TPM_RC_KEY or unmarshal errors may be returned at this
28      // point
29      result = CryptSecretDecrypt(object, NULL, IDENTITY_STRING, &in->secret, &data);
30      if(result != TPM_RC_SUCCESS)
31      {
32          if(result == TPM_RC_KEY)
33              return TPM_RC_FAILURE;
34          return RcSafeAddToResult(result, RC_ActivateCredential_secret);
35      }
36      // Retrieve secret data.  A TPM_RC_INTEGRITY error or unmarshal
37      // errors may be returned at this point
38      result = CredentialToSecret(&in->credentialBlob.b,
39                                  &activateObject->name.b,
40                                  &data.b,
41                                  object,
42                                  &out->certInfo);
43      if(result != TPM_RC_SUCCESS)
44          return RcSafeAddToResult(result, RC_ActivateCredential_credentialBlob);
45      return TPM_RC_SUCCESS;
```

```
46     }
47     #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

```
1   #include "Tpm.h"
2   #include "MakeCredential_fp.h"
3   #ifdef TPM_CC_MakeCredential  // Conditional expansion of this file
4   #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

```
5   TPM_RC
6   TPM2_MakeCredential(
7       MakeCredential_In    *in,              // IN: input parameter list
8       MakeCredential_Out   *out              // OUT: output parameter list
9       )
10  {
11      TPM_RC                  result = TPM_RC_SUCCESS;
12      OBJECT               *object;
13      TPM2B_DATA            data;
14  // Input Validation
15      // Get object pointer
16      object = HandleToObject(in->handle);
17      // input key must be an asymmetric, restricted decryption key
18      // NOTE: Needs to be restricted to have a symmetric value.
19      if(!CryptIsAsymAlgorithm(object->publicArea.type)
20          || object->publicArea.objectAttributes.decrypt == CLEAR
21          || object->publicArea.objectAttributes.restricted == CLEAR)
22          return TPM_RCS_TYPE + RC_MakeCredential_handle;
23      // The credential information may not be larger than the digest size used for
24      // the Name of the key associated with handle.
25      if(in->credential.t.size > CryptHashGetDigestSize(object->publicArea.nameAlg))
26          return TPM_RCS_SIZE + RC_MakeCredential_credential;
27  // Command Output
28      // Make encrypt key and its associated secret structure.
29      out->secret.t.size = sizeof(out->secret.t.secret);
30      result = CryptSecretEncrypt(object, IDENTITY_STRING, &data, &out->secret);
31      if(result != TPM_RC_SUCCESS)
32          return result;
33      // Prepare output credential data from secret
34      SecretToCredential(&in->credential, &in->objectName.b, &data.b,
35                  object, &out->credentialBlob);
36      return TPM_RC_SUCCESS;
37  }
38  #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

```
1   #include "Tpm.h"
2   #include "Unseal_fp.h"
3   #ifdef TPM_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

```
4   TPM_RC
5   TPM2_Unseal(
6       Unseal_In           *in,
7       Unseal_Out          *out
8       )
9   {
10      OBJECT                  *object;
11  // Input Validation
12      // Get pointer to loaded object
13      object = HandleToObject(in->itemHandle);
14      // Input handle must be a data object
15      if(object->publicArea.type != TPM_ALG_KEYEDHASH)
16          return TPM_RCS_TYPE + RC_Unseal_itemHandle;
17      if(object->publicArea.objectAttributes.decrypt == SET
18         || object->publicArea.objectAttributes.sign == SET
19         || object->publicArea.objectAttributes.restricted == SET)
20          return TPM_RCS_ATTRIBUTES + RC_Unseal_itemHandle;
21  // Command Output
22      // Copy data
23      out->outData = object->sensitive.sensitive.bits;
24      return TPM_RC_SUCCESS;
25  }
26  #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

```
1   #include "Tpm.h"
2   #include "ObjectChangeAuth_fp.h"
3   #ifdef TPM_CC_ObjectChangeAuth   // Conditional expansion of this file
4   #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.

```
5    TPM_RC
6    TPM2_ObjectChangeAuth(
7        ObjectChangeAuth_In      *in,              // IN: input parameter list
8        ObjectChangeAuth_Out     *out              // OUT: output parameter list
9        )
10   {
11       TPMT_SENSITIVE           sensitive;
12       OBJECT                   *object = HandleToObject(in->objectHandle);
13       TPM2B_NAME               QNCompare;
14   // Input Validation
15       // Can not change authorization on sequence object
16       if(ObjectIsSequence(object))
17           return TPM_RCS_TYPE + RC_ObjectChangeAuth_objectHandle;
18       // Make sure that the authorization value is consistent with the nameAlg
19       if(!AdjustAuthSize(&in->newAuth, object->publicArea.nameAlg))
20           return TPM_RCS_SIZE + RC_ObjectChangeAuth_newAuth;
21       // Parent handle should be the parent of object handle.  In this
22       // implementation we verify this by checking the QN of object.  Other
23       // implementation may choose different method to verify this attribute.
24       ComputeQualifiedName(in->parentHandle,
25                       object->publicArea.nameAlg,
26                       &object->name, &QNCompare);
27       if(!MemoryEqual2B(&object->qualifiedName.b, &QNCompare.b))
28           return TPM_RCS_TYPE + RC_ObjectChangeAuth_parentHandle;
29   // Command Output
30       // Prepare the sensitive area with the new authorization value
31       sensitive = object->sensitive;
32       sensitive.authValue = in->newAuth;
33       // Protect the sensitive area
34       SensitiveToPrivate(&sensitive, &object->name.b, HandleToObject(in->parentHandle),
35                       object->publicArea.nameAlg,
36                       &out->outPrivate);
37       return TPM_RC_SUCCESS;
38   }
39   #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 exeption: when *parentHandle* references a Derivation Parent, then *sensitiveDataOrigin* in *inPublic* is required to be CLEAR.

For RSA keys, the TPM may return TPM_RC_VALUE.

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

NOTE   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

```
1    #include "Tpm.h"
2    #include "CreateLoaded_fp.h"
3    #ifdef TPM_CC_CreateLoaded  // 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; 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).

```
4    TPM_RC
5    TPM2_CreateLoaded(
6        CreateLoaded_In     *in,                // IN: input parameter list
7        CreateLoaded_Out    *out                // OUT: output parameter list
8        )
9    {
10   // Local variables
11       TPM_RC                      result = TPM_RC_SUCCESS;
12       // These are the values used in object creation
13       TPM2B_NAME                  name;
14       OBJECT                      *parent = HandleToObject(in->parentHandle);
15       OBJECT                      *newObject;
16       BOOL                        derivation;
17       TPMT_PUBLIC                 *publicArea;
18       RAND_STATE                  randState;
19       RAND_STATE                  *rand = &randState;
20   // Input Validation
21       // How the public area is unmarshaled is determined by the parent, so
22       // see if parent is a derivation parent
23       derivation = (parent != NULL && parent->attributes.derivation);
24       // If the parent is an object, then make sure that it is either a parent or
25       // derivation parent
26       if(parent != NULL && !parent->attributes.isParent && !derivation)
27           return TPM_RCS_TYPE + RC_CreateLoaded_parentHandle;
28       // Get a spot in which to create the newObject
29       newObject = FindEmptyObjectSlot(&out->objectHandle);
30       if(newObject == NULL)
31           return TPM_RC_OBJECT_MEMORY;
32       // Do this to save typing
33       publicArea = &newObject->publicArea;
```

```
34          // Unmarshal the template into the object space. TPM2_Create() and
35          // TPM2_CreatePrimary() have the publicArea unmarshaled by CommandDispatcher.
36          // This command is different because of an unfortunate property of the
37          // unique field of an ECC key. It is a structure rather than a single TPM2B. If
38          // if had been a TPM2B, then the label and context could be within a TPM2B and
39          // unmarshaled like other public areas. Since it is not, this command needs its
40          // on template that is a TPM2B that is unmarshaled as a BYTE array with a
41          // its own unmarshal function.
42          result = UnmarshalToPublic(publicArea, &in->inPublic, derivation);
43          if(result != TPM_RC_SUCCESS)
44              return result + RC_CreateLoaded_inPublic;
45          // Validate that the authorization size is appropriate
46          if(!AdjustAuthSize(&in->inSensitive.sensitive.userAuth, publicArea->nameAlg))
47              return TPM_RCS_SIZE + RC_CreateLoaded_inSensitive;
48          // Command output
49          if(derivation)
50          {
51              TPMT_KEYEDHASH_SCHEME        *scheme;
52              scheme = &parent->publicArea.parameters.keyedHashDetail.scheme;
53              // SP800-108 is the only KDF supported by this implementation and there is
54              // no default hash algorithm.
55              pAssert(scheme->details.xor.hashAlg != TPM_ALG_NULL
56                      && scheme->details.xor.kdf == TPM_ALG_KDF1_SP800_108);
57              // Don't derive RSA keys
58              if(publicArea->type == ALG_RSA_VALUE)
59                  return TPM_RCS_TYPE + RC_CreateLoaded_inPublic;
60              // sensitiveDataOrigin has to be CLEAR in a derived object. Since this
61              // is specific to a derived object, it is checked here.
62              if(publicArea->objectAttributes.sensitiveDataOrigin)
63                  return TPM_RCS_ATTRIBUTES;
64              // Check the reset of the attributes
65              result = PublicAttributesValidation(parent, publicArea);
66              if(result != TPM_RC_SUCCESS)
67                  return RcSafeAddToResult(result, RC_CreateLoaded_inPublic);
68              // Process the template and sensitive areas to get the actual 'label' and
69              // 'context' values to be used for this derivation.
70              result = SetLabelAndContext(publicArea, &in->inSensitive.sensitive.data);
71              if(result != TPM_RC_SUCCESS)
72                  return result;
73              // Set up the KDF for object generation
74              DRBG_InstantiateSeededKdf((KDF_STATE *)rand,
75                                        scheme->details.xor.hashAlg,
76                                        scheme->details.xor.kdf,
77                                        &parent->sensitive.seedValue.b,
78                                        &publicArea->unique.derive.label.b,
79                                        &publicArea->unique.derive.context.b);
80              // Clear the sensitive size so that the creation functions will not try
81              // to use this value.
82              in->inSensitive.sensitive.data.t.size = 0;
83          }
84          else
85          {
86              // Check attributes in input public area. CreateChecks() checks the things
87              // that are unique to creation and then validates the attributes and values
88              // that are common to create and load.
89              result = CreateChecks(parent, publicArea);
90              if(result != TPM_RC_SUCCESS)
91                  return RcSafeAddToResult(result, RC_CreateLoaded_inPublic);
92              // Creating a primary object
93              if(parent == NULL)
94              {
95                  newObject->attributes.primary = SET;
96                  if(in->parentHandle == TPM_RH_ENDORSEMENT)
97                      newObject->attributes.epsHierarchy = SET;
98                  // If so, use the primary seed and the digest of the template
99                  // to seed the DRBG
```

```
100                     DRBG_InstantiateSeeded((DRBG_STATE *)rand,
101                                 &HierarchyGetPrimarySeed(in->parentHandle)->b,
102                                 PRIMARY_OBJECT_CREATION,
103                                 &ObjectComputeName(&in->inPublic.b,
104                                             publicArea->nameAlg,
105                                             &name)->b,
106                                 &in->inSensitive.sensitive.data.b);
107             }
108         else
109             // This is an ordinary object so use the normal random number generator
110             rand = NULL;
111     }
112 // Internal data update
113     // Create the object
114     result = CryptCreateObject(newObject, &in->inSensitive.sensitive, rand);
115     if(result != TPM_RC_SUCCESS)
116         return result;
117     // if this is not a Primary key and not a derived key, then return the sensitive
118     // area
119     if(parent != NULL && !derivation)
120         // Prepare output private data from sensitive
121         SensitiveToPrivate(&newObject->sensitive, &newObject->name.b,
122                             parent, newObject->publicArea.nameAlg,
123                             &out->outPrivate);
124     else
125         out->outPrivate.t.size = 0;
126     // Set the remaining return values
127     out->outPublic.publicArea = newObject->publicArea;
128     out->name = newObject->name;
129     // Set the remaining attributes for a loaded object
130     ObjectSetLoadedAttributes(newObject, in->parentHandle);
131     return result;
132 }
133 #endif // CC_CreatePrimary
```

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

$$\mathbf{H}_{policyAlg}(objectHandle{\to}Name \parallel newParentHandle{\to}Name) \tag{2}$$

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

```
1   #include "Tpm.h"
2   #include "Duplicate_fp.h"
3   #ifdef TPM_CC_Duplicate  // Conditional expansion of this file
4   #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

```
5   TPM_RC
6   TPM2_Duplicate(
7       Duplicate_In     *in,            // IN: input parameter list
8       Duplicate_Out    *out            // OUT: output parameter list
9       )
10  {
11      TPM_RC                  result = TPM_RC_SUCCESS;
12      TPMT_SENSITIVE          sensitive;
13      UINT16                  innerKeySize = 0; // encrypt key size for inner wrap
14      OBJECT                  *object;
15      OBJECT                  *newParent;
16      TPM2B_DATA              data;
17  // Input Validation
18      // Get duplicate object pointer
19      object = HandleToObject(in->objectHandle);
20      // Get new parent
21      newParent = HandleToObject(in->newParentHandle);
22      // duplicate key must have fixParent bit CLEAR.
23      if(object->publicArea.objectAttributes.fixedParent == SET)
24          return TPM_RCS_ATTRIBUTES + RC_Duplicate_objectHandle;
25      // Do not duplicate object with NULL nameAlg
26      if(object->publicArea.nameAlg == TPM_ALG_NULL)
27          return TPM_RCS_TYPE + RC_Duplicate_objectHandle;
28      // new parent key must be a storage object or TPM_RH_NULL
29      if(in->newParentHandle != TPM_RH_NULL
30          && !ObjectIsStorage(in->newParentHandle))
31          return TPM_RCS_TYPE + RC_Duplicate_newParentHandle;
32      // If the duplicated object has encryptedDuplication SET, then there must be
33      // an inner wrapper and the new parent may not be TPM_RH_NULL
34      if(object->publicArea.objectAttributes.encryptedDuplication == SET)
35      {
36          if(in->symmetricAlg.algorithm == TPM_ALG_NULL)
37              return TPM_RCS_SYMMETRIC + RC_Duplicate_symmetricAlg;
38          if(in->newParentHandle == TPM_RH_NULL)
39              return TPM_RCS_HIERARCHY + RC_Duplicate_newParentHandle;
```

```
40          }
41          if(in->symmetricAlg.algorithm == TPM_ALG_NULL)
42          {
43              // if algorithm is TPM_ALG_NULL, input key size must be 0
44              if(in->encryptionKeyIn.t.size != 0)
45                  return TPM_RCS_SIZE + RC_Duplicate_encryptionKeyIn;
46          }
47          else
48          {
49              // Get inner wrap key size
50              innerKeySize = in->symmetricAlg.keyBits.sym;
51              // If provided the input symmetric key must match the size of the algorithm
52              if(in->encryptionKeyIn.t.size != 0
53                  && in->encryptionKeyIn.t.size != (innerKeySize + 7) / 8)
54                  return TPM_RCS_SIZE + RC_Duplicate_encryptionKeyIn;
55          }
56  // Command Output
57          if(in->newParentHandle != TPM_RH_NULL)
58          {
59              // Make encrypt key and its associated secret structure.  A TPM_RC_KEY
60              // error may be returned at this point
61              out->outSymSeed.t.size = sizeof(out->outSymSeed.t.secret);
62              result = CryptSecretEncrypt(newParent, DUPLICATE_STRING, &data,
63                                          &out->outSymSeed);
64              if(result != TPM_RC_SUCCESS)
65                  return result;
66          }
67          else
68          {
69              // Do not apply outer wrapper
70              data.t.size = 0;
71              out->outSymSeed.t.size = 0;
72          }
73          // Copy sensitive area
74          sensitive = object->sensitive;
75          // Prepare output private data from sensitive.
76          // Note: If there is no encryption key, one will be provided by
77          // SensitiveToDuplicate(). This is why the assignment of encryptionKeyIn to
78          // encryptionKeyOut will work properly and is not conditional.
79          SensitiveToDuplicate(&sensitive, &object->name.b, newParent,
80                               object->publicArea.nameAlg, &data.b,
81                               &in->symmetricAlg, &in->encryptionKeyIn,
82                               &out->duplicate);
83          out->encryptionKeyOut = in->encryptionKeyIn;
84          return TPM_RC_SUCCESS;
85  }
86  #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 "Terms and Definitions" in TPM 2.0 Part 1).

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

```
1   #include "Tpm.h"
2   #include "Rewrap_fp.h"
3   #ifdef TPM_CC_Rewrap   // Conditional expansion of this file
4   #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
Unmarshal errors	errors during unmarshaling the input encrypted buffer to a ECC public key, or unmarshal the private buffer to sensitive

```
5    TPM_RC
6    TPM2_Rewrap(
7        Rewrap_In        *in,              // IN: input parameter list
8        Rewrap_Out       *out              // OUT: output parameter list
9        )
10   {
11       TPM_RC                    result = TPM_RC_SUCCESS;
12       TPM2B_DATA                data;               // symmetric key
13       UINT16                    hashSize = 0;
14       TPM2B_PRIVATE             privateBlob;        // A temporary private blob
15                                                     // to transit between old
16                                                     // and new wrappers
17   // Input Validation
18       if((in->inSymSeed.t.size == 0 && in->oldParent != TPM_RH_NULL)
19          || (in->inSymSeed.t.size != 0 && in->oldParent == TPM_RH_NULL))
20           return TPM_RCS_HANDLE + RC_Rewrap_oldParent;
21       if(in->oldParent != TPM_RH_NULL)
22       {
23           OBJECT                *oldParent = HandleToObject(in->oldParent);;
24           // old parent key must be a storage object
25           if(!ObjectIsStorage(in->oldParent))
26               return TPM_RCS_TYPE + RC_Rewrap_oldParent;
27           // Decrypt input secret data via asymmetric decryption.  A
28           // TPM_RC_VALUE, TPM_RC_KEY or unmarshal errors may be returned at this
29           // point
30           result = CryptSecretDecrypt(oldParent, NULL, DUPLICATE_STRING,
31                                       &in->inSymSeed, &data);
32           if(result != TPM_RC_SUCCESS)
33               return TPM_RCS_VALUE + RC_Rewrap_inSymSeed;
34           // Unwrap Outer
35           result = UnwrapOuter(oldParent, &in->name.b,
36                                oldParent->publicArea.nameAlg, &data.b,
37                                FALSE,
38                                in->inDuplicate.t.size, in->inDuplicate.t.buffer);
39           if(result != TPM_RC_SUCCESS)
40               return RcSafeAddToResult(result, RC_Rewrap_inDuplicate);
41           // Copy unwrapped data to temporary variable, remove the integrity field
42           hashSize = sizeof(UINT16) +
43               CryptHashGetDigestSize(oldParent->publicArea.nameAlg);
```

```
44              privateBlob.t.size = in->inDuplicate.t.size - hashSize;
45              pAssert(privateBlob.t.size <= sizeof(privateBlob.t.buffer));
46              MemoryCopy(privateBlob.t.buffer, in->inDuplicate.t.buffer + hashSize,
47                         privateBlob.t.size);
48          }
49      else
50          {
51              // No outer wrap from input blob.  Direct copy.
52              privateBlob = in->inDuplicate;
53          }
54      if(in->newParent != TPM_RH_NULL)
55          {
56              OBJECT          *newParent;
57              newParent = HandleToObject(in->newParent);
58              // New parent must be a storage object
59              if(!ObjectIsStorage(in->newParent))
60                  return TPM_RCS_TYPE + RC_Rewrap_newParent;
61              // Make new encrypt key and its associated secret structure.  A
62              // TPM_RC_VALUE error may be returned at this point if RSA algorithm is
63              // enabled in TPM
64              out->outSymSeed.t.size = sizeof(out->outSymSeed.t.secret);
65              result = CryptSecretEncrypt(newParent, DUPLICATE_STRING, &data,
66                                          &out->outSymSeed);
67              if(result != TPM_RC_SUCCESS)
68                  return result;
69              // Copy temporary variable to output, reserve the space for integrity
70              hashSize = sizeof(UINT16) +
71                  CryptHashGetDigestSize(newParent->publicArea.nameAlg);
72       // Make sure that everything fits into the output buffer
73       // Note: this is mostly only an issue if there was no outer wrapper on
74       // 'inDuplicate'. It could be as large as a TPM2B_PRIVATE buffer. If we add
75       // a digest for an outer wrapper, it won't fit anymore.
76              if((privateBlob.t.size + hashSize) > sizeof(out->outDuplicate.t.buffer))
77                  return TPM_RCS_VALUE + RC_Rewrap_inDuplicate;
78   // Command output
79              out->outDuplicate.t.size = privateBlob.t.size;
80              pAssert(privateBlob.t.size
81                      <= sizeof(out->outDuplicate.t.buffer) - hashSize);
82              MemoryCopy(out->outDuplicate.t.buffer + hashSize, privateBlob.t.buffer,
83                         privateBlob.t.size);
84              // Produce outer wrapper for output
85              out->outDuplicate.t.size = ProduceOuterWrap(newParent, &in->name.b,
86                                                          newParent->publicArea.nameAlg,
87                                                          &data.b,
88                                                          FALSE,
89                                                          out->outDuplicate.t.size,
90                                                          out->outDuplicate.t.buffer);
91          }
92      else  // New parent is a null key so there is no seed
93          {
94              out->outSymSeed.t.size = 0;
95              // Copy privateBlob directly
96              out->outDuplicate = privateBlob;
97          }
98      return TPM_RC_SUCCESS;
99  }
100  #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*, then *encryptedDuplication* shall be SET in *objectPublic* (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.

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

```
1    #include "Tpm.h"
2    #include "Import_fp.h"
3    #ifdef TPM_CC_Import  // Conditional expansion of this file
4    #include "Object_spt_fp.h"
```

Error Returns	Meaning
TPM_RC_ASYMMETRIC	non-duplicable storage key represented by *objectPublic* and its parent referenced by *parentHandle* have different public parameters
TPM_RC_ATTRIBUTES	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: 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 non-duplicable storage key represented by *objectPublic* and its parent referenced by *parentHandle* are of different types; 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*

```
 5    TPM_RC
 6    TPM2_Import(
 7        Import_In        *in,             // IN: input parameter list
 8        Import_Out       *out             // OUT: output parameter list
 9        )
10    {
11        TPM_RC                  result = TPM_RC_SUCCESS;
12        OBJECT                 *parentObject;
13        TPM2B_DATA              data;                    // symmetric key
14        TPMT_SENSITIVE          sensitive;
15        TPM2B_NAME              name;
16        UINT16                  innerKeySize = 0;        // encrypt key size for inner
17                                                         // wrapper
18    // Input Validation
19        // FixedTPM and fixedParent must be CLEAR
20        if(in->objectPublic.publicArea.objectAttributes.fixedTPM == SET
21           || in->objectPublic.publicArea.objectAttributes.fixedParent == SET)
22            return TPM_RCS_ATTRIBUTES + RC_Import_objectPublic;
23        // Get parent pointer
24        parentObject = HandleToObject(in->parentHandle);
25        if(!ObjectIsParent(parentObject))
26            return TPM_RCS_TYPE + RC_Import_parentHandle;
27        if(in->symmetricAlg.algorithm != TPM_ALG_NULL)
28        {
29            // Get inner wrap key size
30            innerKeySize = in->symmetricAlg.keyBits.sym;
31            // Input symmetric key must match the size of algorithm.
32            if(in->encryptionKey.t.size != (innerKeySize + 7) / 8)
33                return TPM_RCS_SIZE + RC_Import_encryptionKey;
34        }
35        else
36        {
37            // If input symmetric algorithm is NULL, input symmetric key size must
38            // be 0 as well
39            if(in->encryptionKey.t.size != 0)
40                return TPM_RCS_SIZE + RC_Import_encryptionKey;
41            // If encryptedDuplication is SET, then the object must have an inner
42            // wrapper
43            if(in->objectPublic.publicArea.objectAttributes.encryptedDuplication)
44                return TPM_RCS_ATTRIBUTES + RC_Import_encryptionKey;
45        }
46        // See if there is an outer wrapper
47        if(in->inSymSeed.t.size != 0)
48        {
49            // in->inParentHandle is a parent, but in order to decrypt an outer wrapper,
50            // it must be able to do key exchange and a symmetric key can't do that.
51            if(parentObject->publicArea.type == TPM_ALG_SYMCIPHER)
52                return TPM_RCS_TYPE + RC_Import_parentHandle;
53            // Decrypt input secret data via asymmetric decryption. TPM_RC_ATTRIBUTES,
54            // TPM_RC_ECC_POINT, TPM_RC_INSUFFICIENT, TPM_RC_KEY, TPM_RC_NO_RESULT,
55            // TPM_RC_SIZE, TPM_RC_VALUE may be returned at this point
56            result = CryptSecretDecrypt(parentObject, NULL, DUPLICATE_STRING,
57                                        &in->inSymSeed, &data);
58            pAssert(result != TPM_RC_BINDING);
59            if(result != TPM_RC_SUCCESS)
60                return RcSafeAddToResult(result, RC_Import_inSymSeed);
61        }
62        else
63        {
64            // If encrytpedDuplication is set, then the object must have an outer
```

```
65              // wrapper
66              if(in->objectPublic.publicArea.objectAttributes.encryptedDuplication)
67                  return TPM_RCS_ATTRIBUTES + RC_Import_inSymSeed;
68              data.t.size = 0;
69          }
70      // Compute name of object
71      PublicMarshalAndComputeName(&(in->objectPublic.publicArea), &name);
72      if(name.t.size == 0)
73          return TPM_RCS_HASH + RC_Import_objectPublic;
74      // Retrieve sensitive from private.
75      // TPM_RC_INSUFFICIENT, TPM_RC_INTEGRITY, TPM_RC_SIZE may be returned here.
76      result = DuplicateToSensitive(&in->duplicate.b, &name.b, parentObject,
77                                    in->objectPublic.publicArea.nameAlg,
78                                    &data.b, &in->symmetricAlg,
79                                    &in->encryptionKey.b, &sensitive);
80      if(result != TPM_RC_SUCCESS)
81          return RcSafeAddToResult(result, RC_Import_duplicate);
82      // If the parent of this object has fixedTPM SET, then validate this
83      // object as if it were being loaded so that validation can be skipped
84      // when it is actually loaded.
85      if(parentObject->publicArea.objectAttributes.fixedTPM == SET)
86      {
87          result = ObjectLoad(NULL, NULL, &in->objectPublic.publicArea,
88                              &sensitive, RC_Import_objectPublic, RC_Import_duplicate,
89                              NULL);
90      }
91  // Command output
92      if(result == TPM_RC_SUCCESS)
93      {
94          // Prepare output private data from sensitive
95          SensitiveToPrivate(&sensitive, &name.b, parentObject,
96                             in->objectPublic.publicArea.nameAlg,
97                             &out->outPrivate);
98      }
99      return result;
100  }
101  #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 3447. 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 3447 (PKCS#1). The only supported mask generation is MGF1.

2) TPM_ALG_RSAES – Data is padded as described in 7.2 of IETF RFC 3447 (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 3447 (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. If a zero octet occurs before *label.buffer*[*label.size-1*], the TPM shall truncate the label at that point. 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.

The function returns padded and encrypted value *outData*.

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

NOTE 3   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

```
1   #include "Tpm.h"
2   #include "RSA_Encrypt_fp.h"
3   #ifdef TPM_CC_RSA_Encrypt  // Conditional expansion of this file
4   #ifdef TPM_ALG_RSA
```

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

```
5   TPM_RC
6   TPM2_RSA_Encrypt(
7       RSA_Encrypt_In      *in,                // IN: input parameter list
8       RSA_Encrypt_Out     *out                // OUT: output parameter list
9       )
10  {
11      TPM_RC                  result;
12      OBJECT                  *rsaKey;
13      TPMT_RSA_DECRYPT        *scheme;
14  // Input Validation
15      rsaKey = HandleToObject(in->keyHandle);
16      // selected key must be an RSA key
17      if(rsaKey->publicArea.type != TPM_ALG_RSA)
18          return TPM_RCS_KEY + RC_RSA_Encrypt_keyHandle;
19      // selected key must have the decryption attribute
20      if(rsaKey->publicArea.objectAttributes.decrypt != SET)
21          return TPM_RCS_ATTRIBUTES + RC_RSA_Encrypt_keyHandle;
22      // Is there a label?
23      if(!IsLabelProperlyFormatted(&in->label.b))
24          return TPM_RCS_VALUE + RC_RSA_Encrypt_label;
25  // Command Output
26      // Select a scheme for encryption
27      scheme = CryptRsaSelectScheme(in->keyHandle, &in->inScheme);
28      if(scheme == NULL)
29          return TPM_RCS_SCHEME + RC_RSA_Encrypt_inScheme;
30      // Encryption.  TPM_RC_VALUE, or TPM_RC_SCHEME errors my be returned buy
31      // CryptEncyptRSA.
32      out->outData.t.size = sizeof(out->outData.t.buffer);
33      result = CryptRsaEncrypt(&out->outData, &in->message.b, rsaKey, scheme,
34                      &in->label.b, NULL);
35      return result;
36  }
37  #endif
38  #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 3447 ((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 3447 (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 NULL-terminated string 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

```
1    #include "Tpm.h"
2    #include "RSA_Decrypt_fp.h"
3    #ifdef TPM_CC_RSA_Decrypt  // Conditional expansion of this file
4    #ifdef TPM_ALG_RSA
```

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 an 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

```
5    TPM_RC
6    TPM2_RSA_Decrypt(
7        RSA_Decrypt_In        *in,                // IN: input parameter list
8        RSA_Decrypt_Out       *out                // OUT: output parameter list
9        )
10   {
11       TPM_RC                        result;
12       OBJECT                        *rsaKey;
13       TPMT_RSA_DECRYPT              *scheme;
14   // Input Validation
15       rsaKey = HandleToObject(in->keyHandle);
16       // The selected key must be an RSA key
17       if(rsaKey->publicArea.type != TPM_ALG_RSA)
18           return TPM_RCS_KEY + RC_RSA_Decrypt_keyHandle;
19       // The selected key must be an unrestricted decryption key
20       if(rsaKey->publicArea.objectAttributes.restricted == SET
21          || rsaKey->publicArea.objectAttributes.decrypt == CLEAR)
22           return TPM_RCS_ATTRIBUTES + RC_RSA_Decrypt_keyHandle;
23       // NOTE: Proper operation of this command requires that the sensitive area
24       // of the key is loaded. This is assured because authorization is required
25       // to use the sensitive area of the key. In order to check the authorization,
26       // the sensitive area has to be loaded, even if authorization is with policy.
27       // If label is present, make sure that it is a NULL-terminated string
28       if(!IsLabelProperlyFormatted(&in->label.b))
29           return TPM_RCS_VALUE + RC_RSA_Decrypt_label;
30   // Command Output
31       // Select a scheme for decrypt.
32       scheme = CryptRsaSelectScheme(in->keyHandle, &in->inScheme);
33       if(scheme == NULL)
34           return TPM_RCS_SCHEME + RC_RSA_Decrypt_inScheme;
35       // Decryption.  TPM_RC_VALUE, TPM_RC_SIZE, and TPM_RC_KEY error may be
36       // returned by CryptRsaDecrypt.
37       // NOTE: CryptRsaDecrypt can also return TPM_RC_ATTRIBUTES or TPM_RC_BINDING
38       // when the key is not a decryption key but that was checked above.
39       out->message.t.size = sizeof(out->message.t.buffer);
40       result = CryptRsaDecrypt(&out->message.b, &in->cipherText.b, rsaKey,
41                               scheme, &in->label.b);
42       return result;
43   }
44   #endif
```

45   **#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 $Q_e := [d_e]G$). It uses the private ephemeral key and a loaded public key (Q_S) to compute the shared secret value ($P := [hd_e]Q_S$).

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 $P := h[d_e]Q_s$
TPM2B_ECC_POINT	pubPoint	generated ephemeral public point (Q_e)

14.4.3   Detailed Actions

```
1    #include "Tpm.h"
2    #include "ECDH_KeyGen_fp.h"
3    #ifdef TPM_CC_ECDH_KeyGen  // Conditional expansion of this file
4    #ifdef TPM_ALG_ECC
```

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

```
5    TPM_RC
6    TPM2_ECDH_KeyGen(
7        ECDH_KeyGen_In      *in,            // IN: input parameter list
8        ECDH_KeyGen_Out     *out            // OUT: output parameter list
9        )
10   {
11       OBJECT                  *eccKey;
12       TPM2B_ECC_PARAMETER      sensitive;
13       TPM_RC                   result;
14   // Input Validation
15       eccKey = HandleToObject(in->keyHandle);
16       // Referenced key must be an ECC key
17       if(eccKey->publicArea.type != TPM_ALG_ECC)
18           return TPM_RCS_KEY + RC_ECDH_KeyGen_keyHandle;
19   // Command Output
20       do
21       {
22           TPMT_PUBLIC         *keyPublic = &eccKey->publicArea;
23           // Create ephemeral ECC key
24           result = CryptEccNewKeyPair(&out->pubPoint.point, &sensitive,
25                               keyPublic->parameters.eccDetail.curveID);
26           if(result == TPM_RC_SUCCESS)
27           {
28               // Compute Z
29               result = CryptEccPointMultiply(&out->zPoint.point,
30                                   keyPublic->parameters.eccDetail.curveID,
31                                   &keyPublic->unique.ecc,
32                                   &sensitive,
33                                   NULL, NULL);
34                   // The point in the key is not on the curve. Indicate
35                   // that the key is bad.
36               if(result == TPM_RC_ECC_POINT)
37                   return TPM_RCS_KEY + RC_ECDH_KeyGen_keyHandle;
38               // The other possible error from CryptEccPointMultiply is
39               // TPM_RC_NO_RESULT indicating that the multiplication resulted in
40               // the point at infinity, so get a new random key and start over
41               // BTW, this never happens.
42           }
43       } while(result == TPM_RC_NO_RESULT);
44       return result;
45   }
46   #endif // ALG_ECC
47   #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 ($Z = (x_Z, y_Z) \coloneqq [h d_s] Q_B$; 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 $Z = (x_Z, y_Z) := [hd_S]Q_B$

14.5.3   Detailed Actions

```
1   #include "Tpm.h"
2   #include "ECDH_ZGen_fp.h"
3   #ifdef TPM_CC_ECDH_ZGen  // Conditional expansion of this file
4   #ifdef TPM_ALG_ECC
```

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,

```
5   TPM_RC
6   TPM2_ECDH_ZGen(
7       ECDH_ZGen_In    *in,              // IN: input parameter list
8       ECDH_ZGen_Out   *out              // OUT: output parameter list
9       )
10  {
11      TPM_RC                  result;
12      OBJECT                  *eccKey;
13  // Input Validation
14      eccKey = HandleToObject(in->keyHandle);
15      // Selected key must be a non-restricted, decrypt ECC key
16      if(eccKey->publicArea.type != TPM_ALG_ECC)
17          return TPM_RCS_KEY + RC_ECDH_ZGen_keyHandle;
18      // Selected key needs to be unrestricted with the 'decrypt' attribute
19      if(eccKey->publicArea.objectAttributes.restricted == SET
20         || eccKey->publicArea.objectAttributes.decrypt != SET)
21          return TPM_RCS_ATTRIBUTES + RC_ECDH_ZGen_keyHandle;
22      // Make sure the scheme allows this use
23      if(eccKey->publicArea.parameters.eccDetail.scheme.scheme != TPM_ALG_ECDH
24         &&  eccKey->publicArea.parameters.eccDetail.scheme.scheme != TPM_ALG_NULL)
25          return TPM_RCS_SCHEME + RC_ECDH_ZGen_keyHandle;
26  // Command Output
27      // Compute Z. TPM_RC_ECC_POINT or TPM_RC_NO_RESULT may be returned here.
28      result = CryptEccPointMultiply(&out->outPoint.point,
29                                     eccKey->publicArea.parameters.eccDetail.curveID,
30                                     &in->inPoint.point,
31                                     &eccKey->sensitive.sensitive.ecc,
32                                     NULL, NULL);
33      if(result != TPM_RC_SUCCESS)
34          return RcSafeAddToResult(result, RC_ECDH_ZGen_inPoint);
35      return result;
36  }
37  #endif
38  #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*.

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
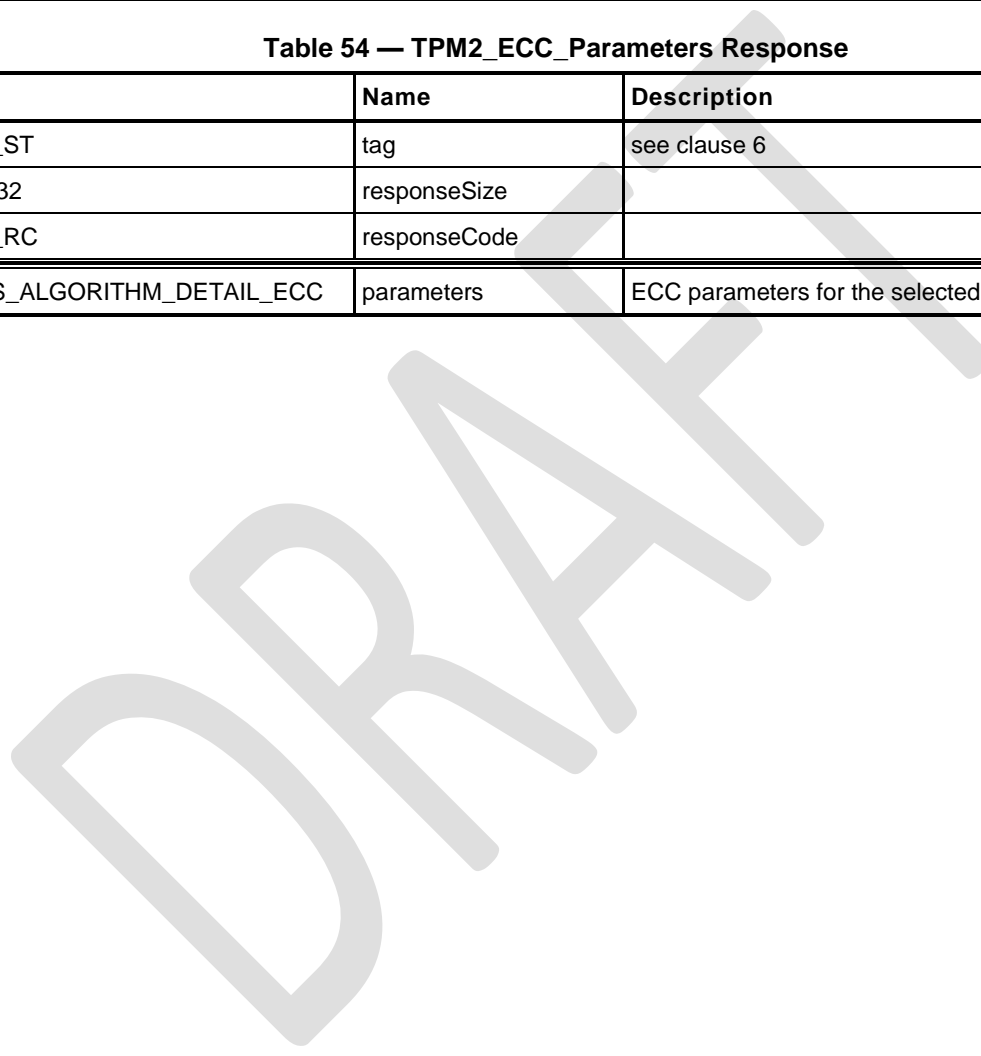
14.6.3   Detailed Actions

```
1    #include "Tpm.h"
2    #include "ECC_Parameters_fp.h"
3    #ifdef TPM_CC_ECC_Parameters  // Conditional expansion of this file
4    #ifdef TPM_ALG_ECC
```

Error Returns	Meaning
TPM_RC_VALUE	Unsupported ECC curve ID

```
5    TPM_RC
6    TPM2_ECC_Parameters(
7        ECC_Parameters_In   *in,              // IN: input parameter list
8        ECC_Parameters_Out  *out              // OUT: output parameter list
9        )
10   {
11   // Command Output
12       // Get ECC curve parameters
13       if(CryptEccGetParameters(in->curveID, &out->parameters))
14           return TPM_RC_SUCCESS;
15       else
16           return TPM_RCS_VALUE + RC_ECC_Parameters_curveID;
17   }
18   #endif
19   #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

```
1    #include "Tpm.h"
2    #include "ZGen_2Phase_fp.h"
3    #ifdef TPM_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, TPM_ALG_ECMQV or TPM_ALG_SM2

```
4    TPM_RC
5    TPM2_ZGen_2Phase(
6        ZGen_2Phase_In      *in,             // IN: input parameter list
7        ZGen_2Phase_Out     *out             // OUT: output parameter list
8        )
9    {
10       TPM_RC                   result;
11       OBJECT                   *eccKey;
12       TPM2B_ECC_PARAMETER      r;
13       TPM_ALG_ID               scheme;
14   // Input Validation
15       eccKey = HandleToObject(in->keyA);
16       // keyA must be an ECC key
17       if(eccKey->publicArea.type != TPM_ALG_ECC)
18           return TPM_RCS_KEY + RC_ZGen_2Phase_keyA;
19       // keyA must not be restricted and must be a decrypt key
20       if(eccKey->publicArea.objectAttributes.restricted == SET
21         || eccKey->publicArea.objectAttributes.decrypt != SET
22         )
23           return TPM_RCS_ATTRIBUTES + RC_ZGen_2Phase_keyA;
24       // if the scheme of keyA is TPM_ALG_NULL, then use the input scheme; otherwise
25       // the input scheme must be the same as the scheme of keyA
26       scheme = eccKey->publicArea.parameters.asymDetail.scheme.scheme;
27       if(scheme != TPM_ALG_NULL)
28       {
29           if(scheme != in->inScheme)
30               return TPM_RCS_SCHEME + RC_ZGen_2Phase_inScheme;
31       }
32       else
33           scheme = in->inScheme;
34       if(scheme == TPM_ALG_NULL)
35           return TPM_RCS_SCHEME + RC_ZGen_2Phase_inScheme;
36       // Input points must be on the curve of keyA
37       if(!CryptEccIsPointOnCurve(eccKey->publicArea.parameters.eccDetail.curveID,
38                           &in->inQsB.point))
39           return TPM_RCS_ECC_POINT + RC_ZGen_2Phase_inQsB;
40       if(!CryptEccIsPointOnCurve(eccKey->publicArea.parameters.eccDetail.curveID,
41                           &in->inQeB.point))
42           return TPM_RCS_ECC_POINT + RC_ZGen_2Phase_inQeB;
43       if(!CryptGenerateR(&r, &in->counter,
44                       eccKey->publicArea.parameters.eccDetail.curveID,
45                       NULL))
46           return TPM_RCS_VALUE + RC_ZGen_2Phase_counter;
47   // Command Output
48       result = CryptEcc2PhaseKeyExchange(&out->outZ1.point,
49                                           &out->outZ2.point,
50                                           eccKey->publicArea.parameters.eccDetail.curveID,
```

```
51                                          scheme,
52                                          &eccKey->sensitive.sensitive.ecc,
53                                          &r,
54                                          &in->inQsB.point,
55                                          &in->inQeB.point);
56          if(result == TPM_RC_SCHEME)
57              return TPM_RCS_SCHEME + RC_ZGen_2Phase_inScheme;
58          if(result == TPM_RC_SUCCESS)
59              CryptEndCommit(in->counter);
60          return result;
61      }
62  #endif
```

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_SYM_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
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

```
1   #include "Tpm.h"
2   #include "EncryptDecrypt_fp.h"
3   #ifdef TPM_CC_EncryptDecrypt2
4   #include  "EncryptDecrypt_spt_fp.h"
5   #endif
6   #ifdef TPM_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

```
7    TPM_RC
8    TPM2_EncryptDecrypt(
9        EncryptDecrypt_In   *in,          // IN: input parameter list
10       EncryptDecrypt_Out  *out          // OUT: output parameter list
11       )
12   {
13   #ifdef TPM_CC_EncryptDecrypt2
14       return EncryptDecryptShared(in->keyHandle, in->decrypt, in->mode,
15                                   &in->ivIn, &in->inData, out);
16   #else
17       OBJECT              *symKey;
18       UINT16               keySize;
19       UINT16               blockSize;
20       BYTE                *key;
21       TPM_ALG_ID           alg;
22       TPM_ALG_ID           mode;
23       TPM_RC               result;
24       BOOL                 OK;
25   // Input Validation
26       symKey = HandleToObject(in->keyHandle);
27       mode = symKey->publicArea.parameters.symDetail.sym.mode.sym;
28       // The input key should be a symmetric key
29       if(symKey->publicArea.type != TPM_ALG_SYMCIPHER)
30           return TPM_RCS_KEY + RC_EncryptDecrypt_keyHandle;
31       // The key must be unrestricted and allow the selected operation
32       OK = symKey->publicArea.objectAttributes.restricted != SET;
33       if(YES == in->decrypt)
34           OK = OK && symKey->publicArea.objectAttributes.decrypt == SET;
35       else
36           OK = OK && symKey->publicArea.objectAttributes.sign == SET;
37       if(!OK)
38           return TPM_RCS_ATTRIBUTES + RC_EncryptDecrypt_keyHandle;
39       // If the key mode is not TPM_ALG_NULL...
40       // or TPM_ALG_NULL
41       if(mode != TPM_ALG_NULL)
42       {
43           // then the input mode has to be TPM_ALG_NULL or the same as the key
44           if((in->mode != TPM_ALG_NULL) && (in->mode != mode))
45               return TPM_RCS_MODE + RC_EncryptDecrypt_mode;
46       }
47       else
48       {
49           // if the key mode is null, then the input can't be null
50           if(in->mode == TPM_ALG_NULL)
51               return TPM_RCS_MODE + RC_EncryptDecrypt_mode;
```

```
52          mode = in->mode;
53      }
54      // The input iv for ECB mode should be an Empty Buffer.  All the other modes
55      // should have an iv size same as encryption block size
56      keySize = symKey->publicArea.parameters.symDetail.sym.keyBits.sym;
57      alg = symKey->publicArea.parameters.symDetail.sym.algorithm;
58      blockSize = CryptGetSymmetricBlockSize(alg, keySize);
59      // Note: When an algorithm is not supported by a TPM, the TPM_ALG_xxx for that
60      // algorithm is not defined. However, it is assumed that the ALG_xxx_VALUE for
61      // the algorithm is always defined. Both have the same numeric value.
62      // ALG_xxx_VALUE is used here so that the code does not get cluttered with
63      // #ifdef's. Having this check does not mean that the algorithm is supported.
64      // If it was not supported the unmarshaling code would have rejected it before
65      // this function were called. This means that, depending on the implementation,
66      // the check could be redundant but it doesn't hurt.
67      if(((in->mode == ALG_ECB_VALUE) && (in->ivIn.t.size != 0))
68          || ((in->mode != ALG_ECB_VALUE) && (in->ivIn.t.size != blockSize)))
69          return TPM_RCS_SIZE + RC_EncryptDecrypt_ivIn;
70      // The input data size of CBC mode or ECB mode must be an even multiple of
71      // the symmetric algorithm's block size
72      if(((in->mode == ALG_CBC_VALUE) || (in->mode == ALG_ECB_VALUE))
73          && ((in->inData.t.size % blockSize) != 0))
74          return TPM_RCS_SIZE + RC_EncryptDecrypt_inData;
75      // Copy IV
76      // Note: This is copied here so that the calls to the encrypt/decrypt functions
77      // will modify the output buffer, not the input buffer
78      out->ivOut = in->ivIn;
79  // Command Output
80      key = symKey->sensitive.sensitive.sym.t.buffer;
81      // For symmetric encryption, the cipher data size is the same as plain data
82      // size.
83      out->outData.t.size = in->inData.t.size;
84      if(in->decrypt == YES)
85      {
86          // Decrypt data to output
87          result = CryptSymmetricDecrypt(out->outData.t.buffer, alg, keySize, key,
88                                         &(out->ivOut), mode, in->inData.t.size,
89                                         in->inData.t.buffer);
90      }
91      else
92      {
93          // Encrypt data to output
94          result = CryptSymmetricEncrypt(out->outData.t.buffer, alg, keySize, key,
95                                         &(out->ivOut), mode, in->inData.t.size,
96                                         in->inData.t.buffer);
97      }
98      return result;
99  #endif // CC_EncryptDecrypt2
100 }
101 #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_SYM_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

```
1    #include "Tpm.h"
2    #include "EncryptDecrypt2_fp.h"
3    #include "EncryptDecrypt_fp.h"
4    #include "EncryptDecrypt_spt_fp.h"
5    #ifdef TPM_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

```
6    TPM_RC
7    TPM2_EncryptDecrypt2(
8        EncryptDecrypt2_In    *in,              // IN: input parameter list
9        EncryptDecrypt2_Out   *out              // OUT: output parameter list
10       )
11   {
12       TPM_RC                result;
13       // EncryptDecyrptShared() performs the operations as shown in
14       // TPM2_EncrypDecrypt
15       result = EncryptDecryptShared(in->keyHandle, in->decrypt, in->mode,
16                                     &in->ivIn, &in->inData,
17                                     (EncryptDecrypt_Out *)out);
18       // Handle response code swizzle.
19       switch(result)
20       {
21           case TPM_RCS_MODE + RC_EncryptDecrypt_mode:
22               result = TPM_RCS_MODE + RC_EncryptDecrypt2_mode;
23               break;
24           case TPM_RCS_SIZE + RC_EncryptDecrypt_ivIn:
25               result = TPM_RCS_SIZE + RC_EncryptDecrypt2_ivIn;
26               break;
27           case TPM_RCS_SIZE + RC_EncryptDecrypt_inData:
28               result = TPM_RCS_SIZE + RC_EncryptDecrypt2_inData;
29               break;
30           default:
31               break;
32       }
33       return result;
34   }
35   #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

```c
1   #include "Tpm.h"
2   #include "Hash_fp.h"
3   #ifdef TPM_CC_Hash  // Conditional expansion of this file
4   TPM_RC
5   TPM2_Hash(
6       Hash_In          *in,            // IN: input parameter list
7       Hash_Out         *out            // OUT: output parameter list
8       )
9   {
10      HASH_STATE          hashState;
11  // Command Output
12      // Output hash
13          // Start hash stack
14      out->outHash.t.size = CryptHashStart(&hashState, in->hashAlg);
15          // Adding hash data
16      CryptDigestUpdate2B(&hashState, &in->data.b);
17          // Complete hash
18      CryptHashEnd2B(&hashState, &out->outHash.b);
19      // Output ticket
20      out->validation.tag = TPM_ST_HASHCHECK;
21      out->validation.hierarchy = in->hierarchy;
22      if(in->hierarchy == TPM_RH_NULL)
23      {
24          // Ticket is not required
25          out->validation.hierarchy = TPM_RH_NULL;
26          out->validation.digest.t.size = 0;
27      }
28      else if(in->data.t.size >= sizeof(TPM_GENERATED)
29              && !TicketIsSafe(&in->data.b))
30      {
31          // Ticket is not safe
32          out->validation.hierarchy = TPM_RH_NULL;
33          out->validation.digest.t.size = 0;
34      }
35      else
36      {
37          // Compute ticket
38          TicketComputeHashCheck(in->hierarchy, in->hashAlg,
39                                 &out->outHash, &out->validation);
40      }
41      return TPM_RC_SUCCESS;
42  }
43  #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.

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 1   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 70.)

NOTE 2   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

```
1    #include "Tpm.h"
2    #include "HMAC_fp.h"
3    #ifdef TPM_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*

```
4    TPM_RC
5    TPM2_HMAC(
6        HMAC_In          *in,           // IN: input parameter list
7        HMAC_Out         *out           // OUT: output parameter list
8        )
9    {
10       HMAC_STATE               hmacState;
11       OBJECT                  *hmacObject;
12       TPMI_ALG_HASH            hashAlg;
13       TPMT_PUBLIC             *publicArea;
14   // Input Validation
15       // Get HMAC key object and public area pointers
16       hmacObject = HandleToObject(in->handle);
17       publicArea = &hmacObject->publicArea;
18       // Make sure that the key is an HMAC key
19       if(publicArea->type != TPM_ALG_KEYEDHASH)
20           return TPM_RCS_TYPE + RC_HMAC_handle;
21       // and that it is unrestricted
22       if(publicArea->objectAttributes.restricted == SET)
23           return TPM_RCS_ATTRIBUTES + RC_HMAC_handle;
24       // and that it is a signing key
25       if(publicArea->objectAttributes.sign != SET)
26           return TPM_RCS_KEY + RC_HMAC_handle;
27       // See if the key has a default
28       if(publicArea->parameters.keyedHashDetail.scheme.scheme == TPM_ALG_NULL)
29           // it doesn't so use the input value
30           hashAlg = in->hashAlg;
31       else
32       {
33           // key has a default so use it
34           hashAlg
35               = publicArea->parameters.keyedHashDetail.scheme.details.hmac.hashAlg;
36           // and verify that the input was either the  TPM_ALG_NULL or the default
37           if(in->hashAlg != TPM_ALG_NULL && in->hashAlg != hashAlg)
38               hashAlg = TPM_ALG_NULL;
39       }
40       // if we ended up without a hash algorithm then return an error
41       if(hashAlg == TPM_ALG_NULL)
42           return TPM_RCS_VALUE + RC_HMAC_hashAlg;
43   // Command Output
44       // Start HMAC stack
45       out->outHMAC.t.size = CryptHmacStart2B(&hmacState, hashAlg,
46                                             &hmacObject->sensitive.sensitive.bits.b);
47       // Adding HMAC data
48       CryptDigestUpdate2B(&hmacState.hashState, &in->buffer.b);
49       // Complete HMAC
50       CryptHmacEnd2B(&hmacState, &out->outHMAC.b);
51       return TPM_RC_SUCCESS;
```

```
52    }
53    #endif // CC_HMAC
```

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 66 — 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 67 — 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

```
1   #include "Tpm.h"
2   #include "GetRandom_fp.h"
3   #ifdef TPM_CC_GetRandom   // Conditional expansion of this file
4   TPM_RC
5   TPM2_GetRandom(
6       GetRandom_In    *in,            // IN: input parameter list
7       GetRandom_Out   *out            // OUT: output parameter list
8       )
9   {
10  // Command Output
11      // if the requested bytes exceed the output buffer size, generates the
12      // maximum bytes that the output buffer allows
13      if(in->bytesRequested > sizeof(TPMU_HA))
14          out->randomBytes.t.size = sizeof(TPMU_HA);
15      else
16          out->randomBytes.t.size = in->bytesRequested;
17      CryptRandomGenerate(out->randomBytes.t.size, out->randomBytes.t.buffer);
18      return TPM_RC_SUCCESS;
19  }
20  #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 68 — 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 69 — TPM2_StirRandom Response

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

16.2.3 Detailed Actions

```
1    #include "Tpm.h"
2    #include "StirRandom_fp.h"
3    #ifdef TPM_CC_StirRandom  // Conditional expansion of this file
4    TPM_RC
5    TPM2_StirRandom(
6        StirRandom_In   *in           // IN: input parameter list
7        )
8    {
9    // Internal Data Update
10       CryptRandomStir(in->inData.t.size, in->inData.t.buffer);
11       return TPM_RC_SUCCESS;
12   }
13   #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, HMAC, and Event Sequences" in TPM 2.0 Part 1.

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 70 — 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.			

17.2.2   Command and Response

Table 71 — 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 72 — 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

```c
1   #include "Tpm.h"
2   #include "HMAC_Start_fp.h"
3   #ifdef TPM_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*

```c
4   TPM_RC
5   TPM2_HMAC_Start(
6       HMAC_Start_In    *in,            // IN: input parameter list
7       HMAC_Start_Out  *out            // OUT: output parameter list
8       )
9   {
10      OBJECT                  *keyObject;
11      TPMT_PUBLIC             *publicArea;
12      TPM_ALG_ID              hashAlg;
13  // Input Validation
14      // Get HMAC key object and public area pointers
15      keyObject = HandleToObject(in->handle);
16      publicArea = &keyObject->publicArea;
17      // Make sure that the key is an HMAC key
18      if(publicArea->type != TPM_ALG_KEYEDHASH)
19          return TPM_RCS_TYPE + RC_HMAC_Start_handle;
20      // and that it is unrestricted
21      if(publicArea->objectAttributes.restricted == SET)
22          return TPM_RCS_ATTRIBUTES + RC_HMAC_Start_handle;
23      // and that it is a signing key
24      if(publicArea->objectAttributes.sign != SET)
25          return TPM_RCS_KEY + RC_HMAC_Start_handle;
26      // See if the key has a default
27      if(publicArea->parameters.keyedHashDetail.scheme.scheme == TPM_ALG_NULL)
28          // it doesn't so use the input value
29          hashAlg = in->hashAlg;
30      else
31      {
32          // key has a default so use it
33          hashAlg
34              = publicArea->parameters.keyedHashDetail.scheme.details.hmac.hashAlg;
35          // and verify that the input was either the  TPM_ALG_NULL or the default
36          if(in->hashAlg != TPM_ALG_NULL && in->hashAlg != hashAlg)
37              hashAlg = TPM_ALG_NULL;
38      }
39      // if we ended up without a hash algorithm then return an error
40      if(hashAlg == TPM_ALG_NULL)
41          return TPM_RCS_VALUE + RC_HMAC_Start_hashAlg;
42  // Internal Data Update
43      // Create a HMAC sequence object. A TPM_RC_OBJECT_MEMORY error may be
44      // returned at this point
45      return ObjectCreateHMACSequence(hashAlg,
46                                      keyObject,
47                                      &in->auth,
48                                      &out->sequenceHandle);
49  }
50  #endif // CC_HMAC_Start
```

17.3   TPM2_HashSequenceStart

17.3.1   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.

17.3.2   Command and Response

Table 73 — 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 74 — 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

17.3.3   Detailed Actions

```
1   #include "Tpm.h"
2   #include "HashSequenceStart_fp.h"
3   #ifdef TPM_CC_HashSequenceStart  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_OBJECT_MEMORY	no space to create an internal object

```
4   TPM_RC
5   TPM2_HashSequenceStart(
6       HashSequenceStart_In    *in,            // IN: input parameter list
7       HashSequenceStart_Out   *out            // OUT: output parameter list
8       )
9   {
10  // Internal Data Update
11      if(in->hashAlg == TPM_ALG_NULL)
12          // Start a event sequence.  A TPM_RC_OBJECT_MEMORY error may be
13          // returned at this point
14          return ObjectCreateEventSequence(&in->auth, &out->sequenceHandle);
15      // Start a hash sequence.  A TPM_RC_OBJECT_MEMORY error may be
16      // returned at this point
17      return ObjectCreateHashSequence(in->hashAlg, &in->auth, &out->sequenceHandle);
18  }
19  #endif // CC_HashSequenceStart
```

17.4   TPM2_SequenceUpdate

17.4.1   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.

17.4.2   Command and Response

Table 75 — 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 76 — TPM2_SequenceUpdate Response

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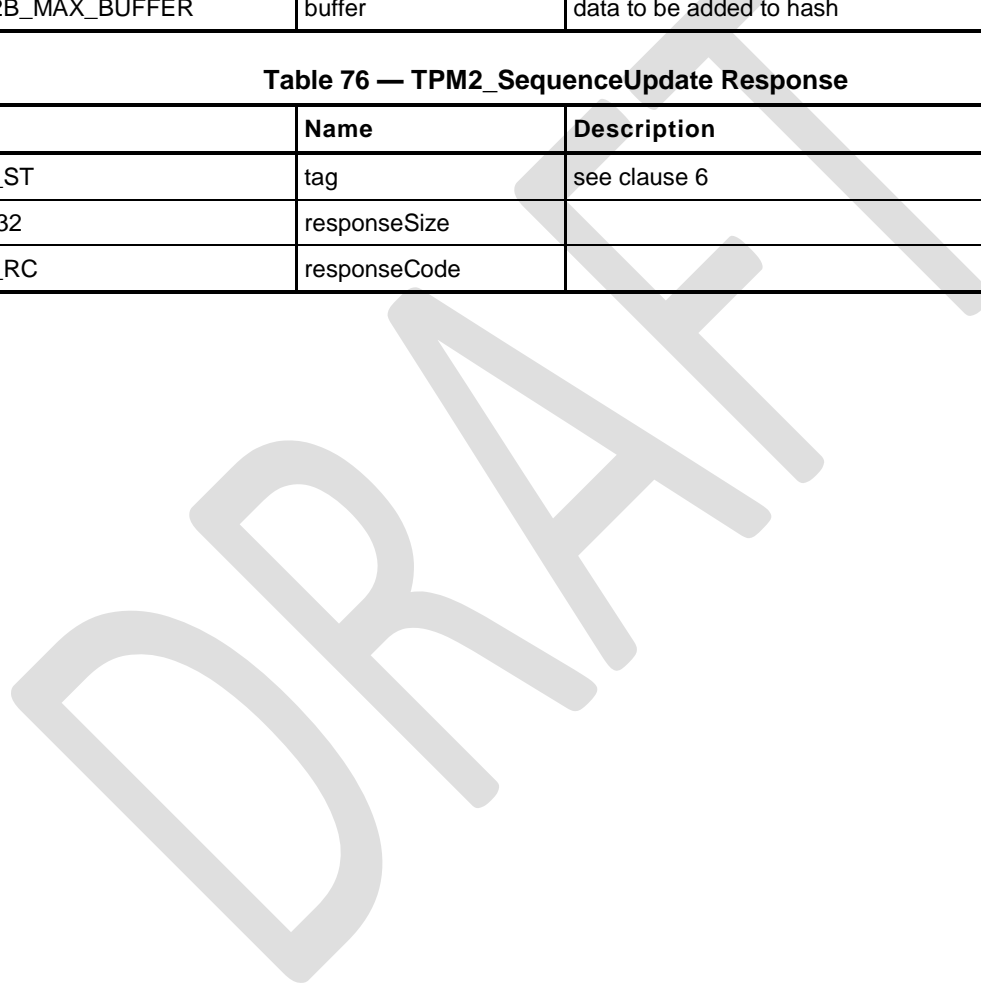
17.4.3   Detailed Actions

```
1    #include "Tpm.h"
2    #include "SequenceUpdate_fp.h"
3    #ifdef TPM_CC_SequenceUpdate  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_MODE	*sequenceHandle* does not reference a hash or HMAC sequence object

```
4    TPM_RC
5    TPM2_SequenceUpdate(
6        SequenceUpdate_In   *in              // IN: input parameter list
7        )
8    {
9        OBJECT                  *object;
10       HASH_OBJECT             *hashObject;
11   // Input Validation
12       // Get sequence object pointer
13       object = HandleToObject(in->sequenceHandle);
14       hashObject = (HASH_OBJECT *)object;
15       // Check that referenced object is a sequence object.
16       if(!ObjectIsSequence(object))
17           return TPM_RCS_MODE + RC_SequenceUpdate_sequenceHandle;
18   // Internal Data Update
19       if(object->attributes.eventSeq == SET)
20       {
21           // Update event sequence object
22           UINT32          i;
23           for(i = 0; i < HASH_COUNT; i++)
24           {
25               // Update sequence object
26               CryptDigestUpdate2B(&hashObject->state.hashState[i], &in->buffer.b);
27           }
28       }
29       else
30       {
31           // Update hash/HMAC sequence object
32           if(hashObject->attributes.hashSeq == SET)
33           {
34               // Is this the first block of the sequence
35               if(hashObject->attributes.firstBlock == CLEAR)
36               {
37                   // If so, indicate that first block was received
38                   hashObject->attributes.firstBlock = SET;
39                   // Check the first block to see if the first block can contain
40                   // the TPM_GENERATED_VALUE.  If it does, it is not safe for
41                   // a ticket.
42                   if(TicketIsSafe(&in->buffer.b))
43                       hashObject->attributes.ticketSafe = SET;
44               }
45               // Update sequence object hash/HMAC stack
46               CryptDigestUpdate2B(&hashObject->state.hashState[0], &in->buffer.b);
47           }
48           else if(object->attributes.hmacSeq == SET)
49           {
50               // Update sequence object hash/HMAC stack
51               CryptDigestUpdate2B(&hashObject->state.hmacState.hashState,
52                                   &in->buffer.b);
53           }
54       }
55       return TPM_RC_SUCCESS;
56   }
```

57   `#endif` `// CC_SequenceUpdate`

17.5 TPM2_SequenceComplete

17.5.1 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.

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.

17.5.2   Command and Response

Table 77 — 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 78 — 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.

17.5.3   Detailed Actions

```
1   #include "Tpm.h"
2   #include "SequenceComplete_fp.h"
3   #ifdef TPM_CC_SequenceComplete   // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_TYPE	*sequenceHandle* does not reference a hash or HMAC sequence object

```
4    TPM_RC
5    TPM2_SequenceComplete(
6        SequenceComplete_In     *in,             // IN: input parameter list
7        SequenceComplete_Out    *out             // OUT: output parameter list
8        )
9    {
10       OBJECT                          *object;
11   // Input validation
12       // Get hash object pointer
13       object = HandleToObject(in->sequenceHandle);
14       // input handle must be a hash or HMAC sequence object.
15       if(object->attributes.hashSeq == CLEAR
16          && object->attributes.hmacSeq == CLEAR)
17           return TPM_RCS_MODE + RC_SequenceComplete_sequenceHandle;
18   // Command Output
19       if(object->attributes.hashSeq == SET)            // sequence object for hash
20       {
21           // Update last piece of data
22           HASH_OBJECT     *hashObject = (HASH_OBJECT *)object;
23           // Get the hash algorithm before the algorithm is lost in CryptHashEnd
24           TPM_ALG_ID       hashAlg = hashObject->state.hashState[0].hashAlg;
25           CryptDigestUpdate2B(&hashObject->state.hashState[0], &in->buffer.b);
26           // Complete hash
27           out->result.t.size
28               = CryptHashGetDigestSize(
29                   CryptHashGetContextAlg(&hashObject->state.hashState[0]));
30           CryptHashEnd2B(&hashObject->state.hashState[0], &out->result.b);
31           // Check if the first block of the sequence has been received
32           if(hashObject->attributes.firstBlock == CLEAR)
33           {
34               // If not, then this is the first block so see if it is 'safe'
35               // to sign.
36               if(TicketIsSafe(&in->buffer.b))
37                   hashObject->attributes.ticketSafe = SET;
38           }
39           // Output ticket
40           out->validation.tag = TPM_ST_HASHCHECK;
41           out->validation.hierarchy = in->hierarchy;
42           if(in->hierarchy == TPM_RH_NULL)
43           {
44               // Ticket is not required
45               out->validation.digest.t.size = 0;
46           }
47           else if(object->attributes.ticketSafe == CLEAR)
48           {
49               // Ticket is not safe to generate
50               out->validation.hierarchy = TPM_RH_NULL;
51               out->validation.digest.t.size = 0;
52           }
53           else
54           {
55               // Compute ticket
56               TicketComputeHashCheck(out->validation.hierarchy, hashAlg,
```

```
57                                           &out->result, &out->validation);
58              }
59          }
60      else
61      {
62          HASH_OBJECT      *hashObject = (HASH_OBJECT *)object;
63          //   Update last piece of data
64          CryptDigestUpdate2B(&hashObject->state.hmacState.hashState, &in->buffer.b);
65          // Complete hash/HMAC
66          out->result.t.size =
67              CryptHashGetDigestSize(
68                  CryptHashGetContextAlg(&hashObject->state.hmacState.hashState));
69          CryptHmacEnd2B(&(hashObject->state.hmacState), &out->result.b);
70          // No ticket is generated for HMAC sequence
71          out->validation.tag = TPM_ST_HASHCHECK;
72          out->validation.hierarchy = TPM_RH_NULL;
73          out->validation.digest.t.size = 0;
74      }
75  // Internal Data Update
76      // mark sequence object as evict so it will be flushed on the way out
77      object->attributes.evict = SET;
78      return TPM_RC_SUCCESS;
79  }
80  #endif // CC_SequenceComplete
```

17.6 TPM2_EventSequenceComplete

17.6.1 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() with the *pcrHandle* in each bank extended with the associated digest value.

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.

17.6.2 Command and Response

Table 79 — 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 80 — 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

17.6.3 Detailed Actions

```
1   #include "Tpm.h"
2   #include "EventSequenceComplete_fp.h"
3   #ifdef TPM_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

```
4   TPM_RC
5   TPM2_EventSequenceComplete(
6       EventSequenceComplete_In    *in,            // IN: input parameter list
7       EventSequenceComplete_Out   *out            // OUT: output parameter list
8       )
9   {
10      HASH_OBJECT             *hashObject;
11      UINT32                  i;
12      TPM_ALG_ID              hashAlg;
13  // Input validation
14      // get the event sequence object pointer
15      hashObject = (HASH_OBJECT *)HandleToObject(in->sequenceHandle);
16      // input handle must reference an event sequence object
17      if(hashObject->attributes.eventSeq != SET)
18          return TPM_RCS_MODE + RC_EventSequenceComplete_sequenceHandle;
19      // see if a PCR extend is requested in call
20      if(in->pcrHandle != TPM_RH_NULL)
21      {
22          // see if extend of the PCR is allowed at the locality of the command,
23          if(!PCRIsExtendAllowed(in->pcrHandle))
24              return TPM_RC_LOCALITY;
25          // if an extend is going to take place, then check to see if there has
26          // been an orderly shutdown. If so, and the selected PCR is one of the
27          // state saved PCR, then the orderly state has to change. The orderly state
28          // does not change for PCR that are not preserved.
29          // NOTE: This doesn't just check for Shutdown(STATE) because the orderly
30          // state will have to change if this is a state-saved PCR regardless
31          // of the current state. This is because a subsequent Shutdown(STATE) will
32          // check to see if there was an orderly shutdown and not do anything if
33          // there was. So, this must indicate that a future Shutdown(STATE) has
34          // something to do.
35          if(PCRIsStateSaved(in->pcrHandle))
36              RETURN_IF_ORDERLY;
37      }
38  // Command Output
39      out->results.count = 0;
40      for(i = 0; i < HASH_COUNT; i++)
41      {
42          hashAlg = CryptHashGetAlgByIndex(i);
43          // Update last piece of data
44          CryptDigestUpdate2B(&hashObject->state.hashState[i], &in->buffer.b);
45          // Complete hash
46          out->results.digests[out->results.count].hashAlg = hashAlg;
47          CryptHashEnd(&hashObject->state.hashState[i],
48                      CryptHashGetDigestSize(hashAlg),
49                      (BYTE *)&out->results.digests[out->results.count].digest);
50      // Extend PCR
51          if(in->pcrHandle != TPM_RH_NULL)
52              PCRExtend(in->pcrHandle, hashAlg,
53                      CryptHashGetDigestSize(hashAlg),
54                      (BYTE *)&out->results.digests[out->results.count].digest);
55          out->results.count++;
56      }
```

```
57    // Internal Data Update
58        // mark sequence object as evict so it will be flushed on the way out
59        hashObject->attributes.evict = SET;
60        return TPM_RC_SUCCESS;
61    }
62    #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 *firmwareVersion* numbers are obfuscated when the signing key is not in the Endorsement or Platform hierarchies.

The obfuscation value is computed by:

$$obfuscation := \textbf{KDFa}(signHandle{\rightarrow}nameAlg, shProof, \text{``OBFUSCATE''}, signHandle{\rightarrow}QN, 0, 128) \quad (3)$$

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 81 — 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 82 — 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

```
1    #include "Tpm.h"
2    #include "Attest_spt_fp.h"
3    #include "Certify_fp.h"
4    #ifdef TPM_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).

```
5    TPM_RC
6    TPM2_Certify(
7        Certify_In      *in,             // IN: input parameter list
8        Certify_Out     *out             // OUT: output parameter list
9        )
10   {
11       TPMS_ATTEST              certifyInfo;
12       OBJECT                   *signObject = HandleToObject(in->signHandle);
13       OBJECT                   *certifiedObject = HandleToObject(in->objectHandle);
14   // Input validation
15       if(!IsSigningObject(signObject))
16           return TPM_RCS_KEY + RC_Certify_signHandle;
17       if(!CryptSelectSignScheme(signObject, &in->inScheme))
18           return TPM_RC_SCHEME + RC_Certify_inScheme;
19   // Command Output
20       // Filling in attest information
21       // Common fields
22       FillInAttestInfo(in->signHandle, &in->inScheme, &in->qualifyingData,
23                        &certifyInfo);
24       // Certify specific fields
25       certifyInfo.type = TPM_ST_ATTEST_CERTIFY;
26       // NOTE: the certified object is not allowed to be TPM_ALG_NULL so
27       // 'certifiedObject' will never be NULL
28       certifyInfo.attested.certify.name = certifiedObject->name;
29       certifyInfo.attested.certify.qualifiedName = certifiedObject->qualifiedName;
30       // Sign attestation structure.  A NULL signature will be returned if
31       // signHandle is TPM_RH_NULL.  A TPM_RC_NV_UNAVAILABLE, TPM_RC_NV_RATE,
32       // TPM_RC_VALUE, TPM_RC_SCHEME or TPM_RC_ATTRIBUTES error may be returned
33       // by SignAttestInfo()
34       return SignAttestInfo(signObject, &in->inScheme, &certifyInfo,
35                             &in->qualifyingData, &out->certifyInfo, &out->signature);
36   }
37   #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:

$$\mathbf{HMAC}(proof, (\text{TPM\_ST\_CREATION} \,||\, objectHandle{\rightarrow}Name \,||\, creationHash)) \qquad (4)$$

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 83 — 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 84 — 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

```
1    #include "Tpm.h"
2    #include "Attest_spt_fp.h"
3    #include "CertifyCreation_fp.h"
4    #ifdef TPM_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).

```
5    TPM_RC
6    TPM2_CertifyCreation(
7        CertifyCreation_In      *in,                // IN: input parameter list
8        CertifyCreation_Out     *out                // OUT: output parameter list
9        )
10   {
11       TPMT_TK_CREATION        ticket;
12       TPMS_ATTEST             certifyInfo;
13       OBJECT                  *certified = HandleToObject(in->objectHandle);
14       OBJECT                  *signObject = HandleToObject(in->signHandle);
15   // Input Validation
16       if(!IsSigningObject(signObject))
17           return TPM_RCS_KEY + RC_CertifyCreation_signHandle;
18       if(!CryptSelectSignScheme(signObject, &in->inScheme))
19           return TPM_RC_SCHEME + RC_CertifyCreation_inScheme;
20       // CertifyCreation specific input validation
21       // Re-compute ticket
22       TicketComputeCreation(in->creationTicket.hierarchy, &certified->name,
23                       &in->creationHash, &ticket);
24       // Compare ticket
25       if(!MemoryEqual2B(&ticket.digest.b, &in->creationTicket.digest.b))
26           return TPM_RCS_TICKET + RC_CertifyCreation_creationTicket;
27   // Command Output
28       // Common fields
29       FillInAttestInfo(in->signHandle, &in->inScheme, &in->qualifyingData,
30                   &certifyInfo);
31       // CertifyCreation specific fields
32       // Attestation type
33       certifyInfo.type = TPM_ST_ATTEST_CREATION;
34       certifyInfo.attested.creation.objectName = certified->name;
35       // Copy the creationHash
36       certifyInfo.attested.creation.creationHash = in->creationHash;
37       // Sign attestation structure.  A NULL signature will be returned if
38       // signObject is TPM_RH_NULL.  A TPM_RC_NV_UNAVAILABLE, TPM_RC_NV_RATE,
39       // TPM_RC_VALUE, TPM_RC_SCHEME or TPM_RC_ATTRIBUTES error may be returned at
40       // this point
41       return SignAttestInfo(signObject, &in->inScheme, &certifyInfo,
42                       &in->qualifyingData, &out->certifyInfo,
43                       &out->signature);
44   }
45   #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 85 — 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 86 — 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

```
1    #include "Tpm.h"
2    #include "Attest_spt_fp.h"
3    #include "Quote_fp.h"
4    #ifdef TPM_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

```
5    TPM_RC
6    TPM2_Quote(
7        Quote_In        *in,              // IN: input parameter list
8        Quote_Out       *out              // OUT: output parameter list
9        )
10   {
11       TPMI_ALG_HASH           hashAlg;
12       TPMS_ATTEST             quoted;
13       OBJECT                  *signObject = HandleToObject(in->signHandle);
14   // Input Validation
15       if(!IsSigningObject(signObject))
16           return TPM_RCS_KEY + RC_Quote_signHandle;
17       if(!CryptSelectSignScheme(signObject, &in->inScheme))
18           return TPM_RC_SCHEME + RC_Quote_inScheme;
19   // Command Output
20       // Filling in attest information
21       // Common fields
22       // FillInAttestInfo may return TPM_RC_SCHEME or TPM_RC_KEY
23       FillInAttestInfo(in->signHandle, &in->inScheme, &in->qualifyingData, &quoted);
24       // Quote specific fields
25       // Attestation type
26       quoted.type = TPM_ST_ATTEST_QUOTE;
27       // Get hash algorithm in sign scheme.  This hash algorithm is used to
28       // compute PCR digest. If there is no algorithm, then the PCR cannot
29       // be digested and this command returns TPM_RC_SCHEME
30       hashAlg = in->inScheme.details.any.hashAlg;
31       if(hashAlg == TPM_ALG_NULL)
32           return TPM_RCS_SCHEME + RC_Quote_inScheme;
33       // Compute PCR digest
34       PCRComputeCurrentDigest(hashAlg, &in->PCRselect,
35                               &quoted.attested.quote.pcrDigest);
36       // Copy PCR select.  "PCRselect" is modified in PCRComputeCurrentDigest
37       // function
38       quoted.attested.quote.pcrSelect = in->PCRselect;
39       // Sign attestation structure.  A NULL signature will be returned if
40       // signObject is NULL.
41       return SignAttestInfo(signObject, &in->inScheme, &quoted, &in->qualifyingData,
42                             &out->quoted, &out->signature);
43   }
44   #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 87 — 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 88 — 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

```
1    #include "Tpm.h"
2    #include "Attest_spt_fp.h"
3    #include "GetSessionAuditDigest_fp.h"
4    #ifdef TPM_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)

```
5    TPM_RC
6    TPM2_GetSessionAuditDigest(
7        GetSessionAuditDigest_In    *in,           // IN: input parameter list
8        GetSessionAuditDigest_Out   *out           // OUT: output parameter list
9        )
10   {
11       SESSION                 *session = SessionGet(in->sessionHandle);
12       TPMS_ATTEST              auditInfo;
13       OBJECT                  *signObject = HandleToObject(in->signHandle);
14   // Input Validation
15       if(!IsSigningObject(signObject))
16           return TPM_RCS_KEY + RC_GetSessionAuditDigest_signHandle;
17       if(!CryptSelectSignScheme(signObject, &in->inScheme))
18           return TPM_RC_SCHEME + RC_GetSessionAuditDigest_inScheme;
19       // session must be an audit session
20       if(session->attributes.isAudit == CLEAR)
21           return TPM_RCS_TYPE + RC_GetSessionAuditDigest_sessionHandle;
22   // Command Output
23       // Fill in attest information common fields
24       FillInAttestInfo(in->signHandle, &in->inScheme, &in->qualifyingData,
25                   &auditInfo);
26       // SessionAuditDigest specific fields
27       auditInfo.type = TPM_ST_ATTEST_SESSION_AUDIT;
28       auditInfo.attested.sessionAudit.sessionDigest = session->u2.auditDigest;
29       // Exclusive audit session
30       auditInfo.attested.sessionAudit.exclusiveSession
31           = (g_exclusiveAuditSession == in->sessionHandle);
32       // Sign attestation structure.  A NULL signature will be returned if
33       // signObject is NULL.
34       return SignAttestInfo(signObject, &in->inScheme, &auditInfo,
35                           &in->qualifyingData, &out->auditInfo,
36                           &out->signature);
37   }
38   #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 89 — 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 90 — 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

```
1    #include "Tpm.h"
2    #include "Attest_spt_fp.h"
3    #include "GetCommandAuditDigest_fp.h"
4    #ifdef TPM_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)

```
5    TPM_RC
6    TPM2_GetCommandAuditDigest(
7        GetCommandAuditDigest_In    *in,              // IN: input parameter list
8        GetCommandAuditDigest_Out   *out              // OUT: output parameter list
9        )
10   {
11       TPM_RC                  result;
12       TPMS_ATTEST             auditInfo;
13       OBJECT                  *signObject = HandleToObject(in->signHandle);
14   // Input validation
15       if(!IsSigningObject(signObject))
16           return TPM_RCS_KEY + RC_GetCommandAuditDigest_signHandle;
17       if(!CryptSelectSignScheme(signObject, &in->inScheme))
18           return TPM_RC_SCHEME + RC_GetCommandAuditDigest_inScheme;
19   // Command Output
20       // Fill in attest information common fields
21       FillInAttestInfo(in->signHandle, &in->inScheme, &in->qualifyingData,
22                   &auditInfo);
23       // CommandAuditDigest specific fields
24       auditInfo.type = TPM_ST_ATTEST_COMMAND_AUDIT;
25       auditInfo.attested.commandAudit.digestAlg = gp.auditHashAlg;
26       auditInfo.attested.commandAudit.auditCounter = gp.auditCounter;
27       // Copy command audit log
28       auditInfo.attested.commandAudit.auditDigest = gr.commandAuditDigest;
29       CommandAuditGetDigest(&auditInfo.attested.commandAudit.commandDigest);
30       // Sign attestation structure.  A NULL signature will be returned if
31       // signHandle is TPM_RH_NULL.  A TPM_RC_NV_UNAVAILABLE, TPM_RC_NV_RATE,
32       // TPM_RC_VALUE, TPM_RC_SCHEME or TPM_RC_ATTRIBUTES error may be returned at
33       // this point
34       result = SignAttestInfo(signObject, &in->inScheme, &auditInfo,
35                           &in->qualifyingData, &out->auditInfo,
36                           &out->signature);
37       // Internal Data Update
38       if(result == TPM_RC_SUCCESS && in->signHandle != TPM_RH_NULL)
39           // Reset log
40           gr.commandAuditDigest.t.size = 0;
41       return result;
42   }
43   #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 91 — 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 92 — 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

```
1   #include "Tpm.h"
2   #include "Attest_spt_fp.h"
3   #include "GetTime_fp.h"
4   #ifdef TPM_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)

```
5   TPM_RC
6   TPM2_GetTime(
7       GetTime_In       *in,              // IN: input parameter list
8       GetTime_Out      *out              // OUT: output parameter list
9       )
10  {
11      TPMS_ATTEST              timeInfo;
12      OBJECT                   *signObject = HandleToObject(in->signHandle);
13  // Input Validation
14      if(!IsSigningObject(signObject))
15          return TPM_RCS_KEY + RC_GetTime_signHandle;
16      if(!CryptSelectSignScheme(signObject, &in->inScheme))
17          return TPM_RC_SCHEME + RC_GetTime_inScheme;
18  // Command Output
19      // Fill in attest common fields
20      FillInAttestInfo(in->signHandle, &in->inScheme, &in->qualifyingData, &timeInfo);
21      // GetClock specific fields
22      timeInfo.type = TPM_ST_ATTEST_TIME;
23      timeInfo.attested.time.time.time = g_time;
24      TimeFillInfo(&timeInfo.attested.time.time.clockInfo);
25      // Firmware version in plain text
26      timeInfo.attested.time.firmwareVersion
27          = (((UINT64)gp.firmwareV1) << 32) + gp.firmwareV2;
28      // Sign attestation structure.  A NULL signature will be returned if
29      // signObject is NULL.
30      return SignAttestInfo(signObject, &in->inScheme, &timeInfo, &in->qualifyingData,
31                      &out->timeInfo, &out->signature);
32  }
33  #endif // CC_GetTime
```

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 93 — 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 94 — 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

```
1    #include "Tpm.h"
2    #include "Commit_fp.h"
3    #ifdef TPM_CC_Commit  // Conditional expansion of this file
4    #ifdef TPM_ALG_ECC
```

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

```
5    TPM_RC
6    TPM2_Commit(
7        Commit_In        *in,                // IN: input parameter list
8        Commit_Out       *out                // OUT: output parameter list
9        )
10   {
11       OBJECT                   *eccKey;
12       TPMS_ECC_POINT            P2;
13       TPMS_ECC_POINT           *pP2 = NULL;
14       TPMS_ECC_POINT           *pP1 = NULL;
15       TPM2B_ECC_PARAMETER       r;
16       TPM2B_ECC_PARAMETER       p;
17       TPM_RC                    result;
18       TPMS_ECC_PARMS           *parms;
19   // Input Validation
20       eccKey = HandleToObject(in->signHandle);
21       parms = &eccKey->publicArea.parameters.eccDetail;
22       // Input key must be an ECC key
23       if(eccKey->publicArea.type != TPM_ALG_ECC)
24           return TPM_RCS_KEY + RC_Commit_signHandle;
25       // This command may only be used with a sign-only key using an anonymous
26       // scheme.
27       // NOTE: a sign + decrypt key has no scheme so it will not be an anonymous one
28       // and an unrestricted sign key might no have a signing scheme but it can't
29       // be use in Commit()
30       if(!CryptIsSchemeAnonymous(parms->scheme.scheme))
31           return TPM_RCS_SCHEME + RC_Commit_signHandle;
32   // Make sure that both parts of P2 are present if either is present
33       if((in->s2.t.size == 0) != (in->y2.t.size == 0))
34           return TPM_RCS_SIZE + RC_Commit_y2;
35       // Get prime modulus for the curve. This is needed later but getting this now
36       // allows confirmation that the curve exists.
37       if(!CryptEccGetParameter(&p, 'p', parms->curveID))
38           return TPM_RCS_KEY + RC_Commit_signHandle;
39       // Get the random value that will be used in the point multiplications
40       // Note: this does not commit the count.
41       if(!CryptGenerateR(&r, NULL, parms->curveID, &eccKey->name))
42           return TPM_RC_NO_RESULT;
43       // Set up P2 if s2 and Y2 are provided
44       if(in->s2.t.size != 0)
45       {
46           TPM2B_DIGEST             x2;
```

```
47              pP2 = &P2;
48              // copy y2 for P2
49              P2.y = in->y2;
50              // Compute x2  HnameAlg(s2) mod p
51              //     do the hash operation on s2 with the size of curve 'p'
52              x2.t.size = CryptHashBlock(eccKey->publicArea.nameAlg,
53                                         in->s2.t.size,
54                                         in->s2.t.buffer,
55                                         sizeof(x2.t.buffer),
56                                         x2.t.buffer);
57          // If there were error returns in the hash routine, indicate a problem
58          // with the hash algorithm selection
59          if(x2.t.size == 0)
60              return TPM_RCS_HASH + RC_Commit_signHandle;
61          // The size of the remainder will be same as the size of p. DivideB() will
62          // pad the results (leading zeros) if necessary to make the size the same
63          P2.x.t.size = p.t.size;
64          //  set p2.x = hash(s2) mod p
65          if(DivideB(&x2.b, &p.b, NULL, &P2.x.b) != TPM_RC_SUCCESS)
66              return TPM_RC_NO_RESULT;
67          if(!CryptEccIsPointOnCurve(parms->curveID, pP2))
68              return TPM_RCS_ECC_POINT + RC_Commit_s2;
69          if(eccKey->attributes.publicOnly == SET)
70              return TPM_RCS_KEY + RC_Commit_signHandle;
71      }
72      // If there is a P1, make sure that it is on the curve
73      // NOTE: an "empty" point has two UINT16 values which are the size values
74      // for each of the coordinates.
75      if(in->P1.size > 4)
76      {
77          pP1 = &in->P1.point;
78          if(!CryptEccIsPointOnCurve(parms->curveID, pP1))
79              return TPM_RCS_ECC_POINT + RC_Commit_P1;
80      }
81      // Pass the parameters to CryptCommit.
82      // The work is not done in-line because it does several point multiplies
83      // with the same curve.  It saves work by not having to reload the curve
84      // parameters multiple times.
85      result = CryptEccCommitCompute(&out->K.point,
86                                     &out->L.point,
87                                     &out->E.point,
88                                     parms->curveID,
89                                     pP1,
90                                     pP2,
91                                     &eccKey->sensitive.sensitive.ecc,
92                                     &r);
93      if(result != TPM_RC_SUCCESS)
94          return result;
95      // The commit computation was successful so complete the commit by setting
96      // the bit
97      out->counter = CryptCommit();
98      return TPM_RC_SUCCESS;
99  }
100 #endif
101 #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 $Q :=$ $[r]G$ where G is the generator point associated with *curveID*.

19.3.2   Command and Response

Table 95 — 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 96 — TPM2_EC_Ephemeral Response

Type	Name	Description
TPM_ST	tag	see 6
UINT32	responseSize	
TPM_RC	responseCode	
TPM2B_ECC_POINT	Q	ephemeral public key $Q := [r]G$
UINT16	counter	least-significant 16 bits of *commitCount*

19.3.3   Detailed Actions

```
1    #include "Tpm.h"
2    #include "EC_Ephemeral_fp.h"
3    #ifdef TPM_CC_EC_Ephemeral   // Conditional expansion of this file
4    #ifdef TPM_ALG_ECC
```

Error Returns	Meaning
TPM_RC_NO_RESULT	the TPM is not able to generate an *r* value

```
5    TPM_RC
6    TPM2_EC_Ephemeral(
7        EC_Ephemeral_In     *in,              // IN: input parameter list
8        EC_Ephemeral_Out    *out              // OUT: output parameter list
9        )
10   {
11       TPM2B_ECC_PARAMETER      r;
12       TPM_RC                   result;
13   //
14       do
15       {
16           // Get the random value that will be used in the point multiplications
17           // Note: this does not commit the count.
18           if(!CryptGenerateR(&r, NULL, in->curveID, NULL))
19               return TPM_RC_NO_RESULT;
20           // do a point multiply
21           result = CryptEccPointMultiply(&out->Q.point, in->curveID, NULL, &r,
22                                          NULL, NULL);
23           // commit the count value if either the r value results in the point at
24           // infinity or if the value is good. The commit on the r value for infinity
25           // is so that the r value will be skipped.
26           if((result == TPM_RC_SUCCESS) || (result == TPM_RC_NO_RESULT))
27               out->counter = CryptCommit();
28       } while(result == TPM_RC_NO_RESULT);
29       return TPM_RC_SUCCESS;
30   }
31   #endif
32   #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.

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 97 — 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 98 — 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

```
1   #include "Tpm.h"
2   #include "VerifySignature_fp.h"
3   #ifdef TPM_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

```
4   TPM_RC
5   TPM2_VerifySignature(
6       VerifySignature_In      *in,              // IN: input parameter list
7       VerifySignature_Out     *out              // OUT: output parameter list
8       )
9   {
10      TPM_RC                    result;
11      OBJECT                   *signObject = HandleToObject(in->keyHandle);
12      TPMI_RH_HIERARCHY         hierarchy;
13  // Input Validation
14      // The object to validate the signature must be a signing key.
15      if(signObject->publicArea.objectAttributes.sign != SET)
16          return TPM_RCS_ATTRIBUTES + RC_VerifySignature_keyHandle;
17      // Validate Signature.  TPM_RC_SCHEME, TPM_RC_HANDLE or TPM_RC_SIGNATURE
18      // error may be returned by CryptCVerifySignatrue()
19      result = CryptValidateSignature(in->keyHandle, &in->digest, &in->signature);
20      if(result != TPM_RC_SUCCESS)
21          return RcSafeAddToResult(result, RC_VerifySignature_signature);
22  // Command Output
23      hierarchy = GetHeriarchy(in->keyHandle);
24      if(hierarchy == TPM_RH_NULL
25          || signObject->publicArea.nameAlg == TPM_ALG_NULL)
26      {
27          // produce empty ticket if hierarchy is TPM_RH_NULL or nameAlg is
28          // TPM_ALG_NULL
29          out->validation.tag = TPM_ST_VERIFIED;
30          out->validation.hierarchy = TPM_RH_NULL;
31          out->validation.digest.t.size = 0;
32      }
33      else
34      {
35          // Compute ticket
36          TicketComputeVerified(hierarchy, &in->digest, &signObject->name,
37                                &out->validation);
38      }
39      return TPM_RC_SUCCESS;
40  }
41  #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.

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.

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 an 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 99 — 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 100 — 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

```
1    #include "Tpm.h"
2    #include "Sign_fp.h"
3    #ifdef TPM_CC_Sign  // Conditional expansion of this file
4    #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*

```
5    TPM_RC
6    TPM2_Sign(
7        Sign_In           *in,              // IN: input parameter list
8        Sign_Out          *out              // OUT: output parameter list
9        )
10   {
11       TPM_RC                    result;
12       TPMT_TK_HASHCHECK         ticket;
13       OBJECT                    *signObject = HandleToObject(in->keyHandle);
14   //
15   // Input Validation
16       if(!IsSigningObject(signObject))
17           return TPM_RCS_KEY + RC_Sign_keyHandle;
18       // pick a scheme for sign.  If the input sign scheme is not compatible with
19       // the default scheme, return an error.
20       if(!CryptSelectSignScheme(signObject, &in->inScheme))
21           return TPM_RCS_SCHEME + RC_Sign_inScheme;
22       // If validation is provided, or the key is restricted, check the ticket
23       if(in->validation.digest.t.size != 0
24          || signObject->publicArea.objectAttributes.restricted == SET)
25       {
26           // Compute and compare ticket
27           TicketComputeHashCheck(in->validation.hierarchy,
28                                  in->inScheme.details.any.hashAlg,
29                                  &in->digest, &ticket);
30           if(!MemoryEqual2B(&in->validation.digest.b, &ticket.digest.b))
31               return TPM_RCS_TICKET + RC_Sign_validation;
32       }
33       else
34       // If we don't have a ticket, at least verify that the provided 'digest'
35       // is the size of the scheme hashAlg digest.
36       // NOTE: this does not guarantee that the 'digest' is actually produced using
37       // the indicated hash algorithm, but at least it might be.
38       {
39           if(in->digest.t.size
40              != CryptHashGetDigestSize(in->inScheme.details.any.hashAlg))
41               return TPM_RCS_SIZE + RC_Sign_digest;
42       }
43   // Command Output
44       // Sign the hash. A TPM_RC_VALUE or TPM_RC_SCHEME
45       // error may be returned at this point
46       result = CryptSign(signObject, &in->inScheme, &in->digest, &out->signature);
47       return result;
48   }
49   #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:

$$commandAuditDigest_{new} := \mathbf{H}_{auditAlg}(commandAuditDigest_{old} \parallel cpHash \parallel rpHash) \qquad (5)$$

where

$\mathbf{H}_{auditAlg}$	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 101 — 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 102 — TPM2_SetCommandCodeAuditStatus Response

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

21.2.3   Detailed Actions

```
1    #include "Tpm.h"
2    #include "SetCommandCodeAuditStatus_fp.h"
3    #ifdef TPM_CC_SetCommandCodeAuditStatus  // Conditional expansion of this file
4    TPM_RC
5    TPM2_SetCommandCodeAuditStatus(
6        SetCommandCodeAuditStatus_In     *in              // IN: input parameter list
7        )
8    {
9        // The command needs NV update.  Check if NV is available.
10       // A TPM_RC_NV_UNAVAILABLE or TPM_RC_NV_RATE error may be returned at
11       // this point
12       RETURN_IF_NV_IS_NOT_AVAILABLE;
13   // Internal Data Update
14       // Update hash algorithm
15       if(in->auditAlg != TPM_ALG_NULL && in->auditAlg != gp.auditHashAlg)
16       {
17           // Can't change the algorithm and command list at the same time
18           if(in->setList.count != 0 || in->clearList.count != 0)
19               return TPM_RCS_VALUE + RC_SetCommandCodeAuditStatus_auditAlg;
20           // Change the hash algorithm for audit
21           gp.auditHashAlg = in->auditAlg;
22           // Set the digest size to a unique value that indicates that the digest
23           // algorithm has been changed. The size will be cleared to zero in the
24           // command audit processing on exit.
25           gr.commandAuditDigest.t.size = 1;
26           // Save the change of command audit data (this sets g_updateNV so that NV
27           // will be updated on exit.)
28           NV_SYNC_PERSISTENT(auditHashAlg);
29       }
30       else
31       {
32           UINT32          i;
33           BOOL            changed = FALSE;
34           // Process set list
35           for(i = 0; i < in->setList.count; i++)
36               // If change is made in CommandAuditSet, set changed flag
37               if(CommandAuditSet(in->setList.commandCodes[i]))
38                   changed = TRUE;
39           // Process clear list
40           for(i = 0; i < in->clearList.count; i++)
41               // If change is made in CommandAuditClear, set changed flag
42               if(CommandAuditClear(in->clearList.commandCodes[i]))
43                   changed = TRUE;
44           // if change was made to command list, update NV
45           if(changed)
46               // this sets g_updateNV so that NV will be updated on exit.
47               NV_SYNC_PERSISTENT(auditCommands);
48       }
49       return TPM_RC_SUCCESS;
50   }
51   #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 may be incremented either once or once for each bank.

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:

$$PCR.digest_{new}\,[pcrNum][alg] := \mathbf{H}_{alg}(PCR.digest_{old}\,[pcrNum][alg]\,||\,data[alg].buffer)) \qquad (6)$$

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 103 — 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 104 — TPM2_PCR_Extend Response

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

22.2.3   Detailed Actions

```
1   #include "Tpm.h"
2   #include "PCR_Extend_fp.h"
3   #ifdef TPM_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*

```
4   TPM_RC
5   TPM2_PCR_Extend(
6       PCR_Extend_In   *in               // IN: input parameter list
7       )
8   {
9       UINT32              i;
10  // Input Validation
11      // NOTE: This function assumes that the unmarshaling function for 'digests' will
12      // have validated that all of the indicated hash algorithms are valid. If the
13      // hash algorithms are correct, the unmarshaling code will unmarshal a digest
14      // of the size indicated by the hash algorithm. If the overall size is not
15      // consistent, the unmarshaling code will run out of input data or have input
16      // data left over. In either case, it will cause an unmarshaling error and this
17      // function will not be called.
18      // For NULL handle, do nothing and return success
19      if(in->pcrHandle == TPM_RH_NULL)
20          return TPM_RC_SUCCESS;
21      // Check if the extend operation is allowed by the current command locality
22      if(!PCRIsExtendAllowed(in->pcrHandle))
23          return TPM_RC_LOCALITY;
24      // If PCR is state saved and we need to update orderlyState, check NV
25      // availability
26      if(PCRIsStateSaved(in->pcrHandle))
27          RETURN_IF_ORDERLY;
28  // Internal Data Update
29      // Iterate input digest list to extend
30      for(i = 0; i < in->digests.count; i++)
31      {
32          PCRExtend(in->pcrHandle, in->digests.digests[i].hashAlg,
33                  CryptHashGetDigestSize(in->digests.digests[i].hashAlg),
34                  (BYTE *)&in->digests.digests[i].digest);
35      }
36      return TPM_RC_SUCCESS;
37  }
38  #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 105 — 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 106 — TPM2_PCR_Event Response

Type	Name	Description
TPM_ST	tag	see clause 6
UINT32	responseSize	
TPM_RC	responseCode	.
TPML_DIGEST_VALUES	digests	
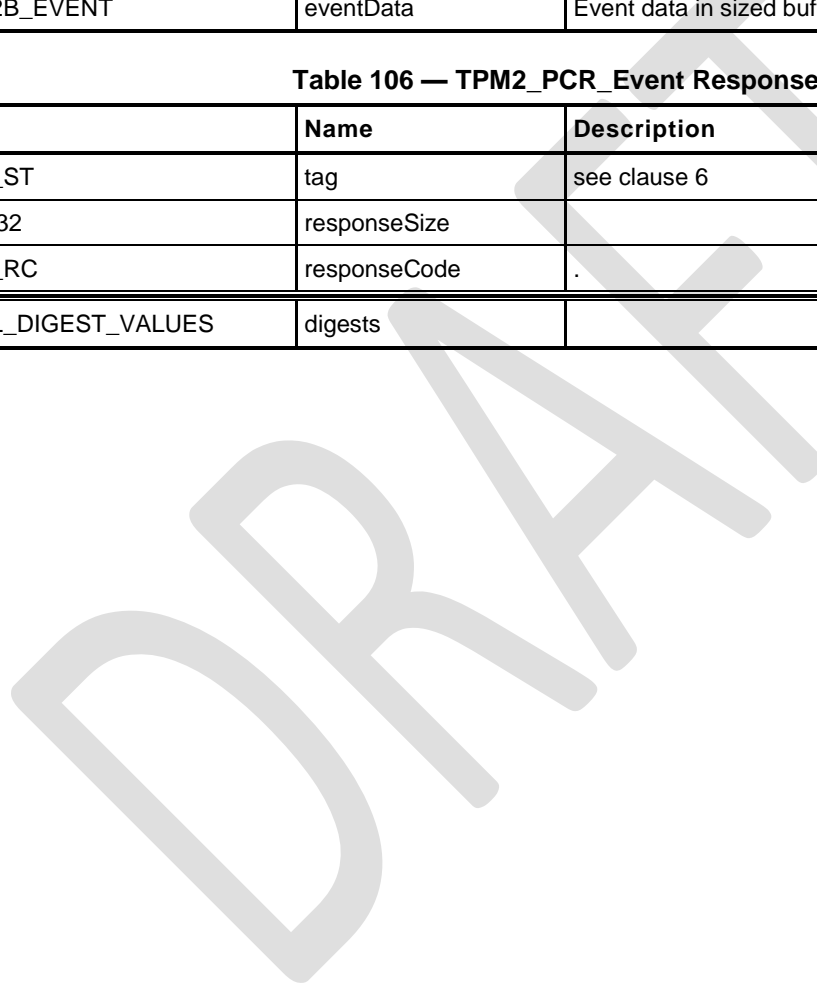
22.3.3   Detailed Actions

```c
1   #include "Tpm.h"
2   #include "PCR_Event_fp.h"
3   #ifdef TPM_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*

```c
4    TPM_RC
5    TPM2_PCR_Event(
6        PCR_Event_In    *in,            // IN: input parameter list
7        PCR_Event_Out   *out            // OUT: output parameter list
8        )
9    {
10       HASH_STATE          hashState;
11       UINT32              i;
12       UINT16              size;
13   // Input Validation
14       // If a PCR extend is required
15       if(in->pcrHandle != TPM_RH_NULL)
16       {
17           // If the PCR is not allow to extend, return error
18           if(!PCRIsExtendAllowed(in->pcrHandle))
19               return TPM_RC_LOCALITY;
20           // If PCR is state saved and we need to update orderlyState, check NV
21           // availability
22           if(PCRIsStateSaved(in->pcrHandle))
23               RETURN_IF_ORDERLY;
24       }
25   // Internal Data Update
26       out->digests.count = HASH_COUNT;
27       // Iterate supported PCR bank algorithms to extend
28       for(i = 0; i < HASH_COUNT; i++)
29       {
30           TPM_ALG_ID  hash = CryptHashGetAlgByIndex(i);
31           out->digests.digests[i].hashAlg = hash;
32           size = CryptHashStart(&hashState, hash);
33           CryptDigestUpdate2B(&hashState, &in->eventData.b);
34           CryptHashEnd(&hashState, size,
35                       (BYTE *)&out->digests.digests[i].digest);
36           if(in->pcrHandle != TPM_RH_NULL)
37               PCRExtend(in->pcrHandle, hash, size,
38                       (BYTE *)&out->digests.digests[i].digest);
39       }
40       return TPM_RC_SUCCESS;
41   }
42   #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 107 — 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 108 — 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

```
1   #include "Tpm.h"
2   #include "PCR_Read_fp.h"
3   #ifdef TPM_CC_PCR_Read  // Conditional expansion of this file
4   TPM_RC
5   TPM2_PCR_Read(
6       PCR_Read_In      *in,              // IN: input parameter list
7       PCR_Read_Out     *out              // OUT: output parameter list
8       )
9   {
10  // Command Output
11      // Call PCR read function.  input pcrSelectionIn parameter could be changed
12      // to reflect the actual PCR being returned
13      PCRRead(&in->pcrSelectionIn, &out->pcrValues, &out->pcrUpdateCounter);
14      out->pcrSelectionOut = in->pcrSelectionIn;
15      return TPM_RC_SUCCESS;
16  }
17  #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 TPM2_Startup(TPM_SU_CLEAR) operation. The PCR allocation in place when this command is executed will be retained until the next TPM2_Startup(TPM_SU_CLEAR). If this command is received multiple times before a TPM2_Startup(TPM_SU_CLEAR), 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 a TPM_PCR_ALLOCATE() 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.

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 109 — 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 110 — 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

```
1   #include "Tpm.h"
2   #include "PCR_Allocate_fp.h"
3   #ifdef TPM_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

```
4   TPM_RC
5   TPM2_PCR_Allocate(
6       PCR_Allocate_In     *in,               // IN: input parameter list
7       PCR_Allocate_Out    *out               // OUT: output parameter list
8       )
9   {
10      TPM_RC      result;
11      // The command needs NV update.  Check if NV is available.
12      // A TPM_RC_NV_UNAVAILABLE or TPM_RC_NV_RATE error may be returned at
13      // this point.
14      // Note: These codes are not listed in the return values above because it is
15      // an implementation choice to check in this routine rather than in a common
16      // function that is called before these actions are called. These return values
17      // are described in the Response Code section of Part 3.
18      RETURN_IF_NV_IS_NOT_AVAILABLE;
19  // Command Output
20      // Call PCR Allocation function.
21      result = PCRAllocate(&in->pcrAllocation, &out->maxPCR,
22                          &out->sizeNeeded, &out->sizeAvailable);
23      if(result == TPM_RC_PCR)
24          return result;
25      //
26      out->allocationSuccess = (result == TPM_RC_SUCCESS);
27      // if re-configuration succeeds, set the flag to indicate PCR configuration is
28      // going to be changed in next boot
29      if(out->allocationSuccess == YES)
30          g_pcrReConfig = TRUE;
31      return TPM_RC_SUCCESS;
32  }
33  #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 111 — 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 112 — TPM2_PCR_SetAuthPolicy Response

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

22.6.3 Detailed Actions

```
1   #include "Tpm.h"
2   #include "PCR_SetAuthPolicy_fp.h"
3   #ifdef TPM_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

```
4   TPM_RC
5   TPM2_PCR_SetAuthPolicy(
6       PCR_SetAuthPolicy_In    *in              // IN: input parameter list
7       )
8   {
9       UINT32      groupIndex;
10      // The command needs NV update.  Check if NV is available.
11      // A TPM_RC_NV_UNAVAILABLE or TPM_RC_NV_RATE error may be returned at
12      // this point
13      RETURN_IF_NV_IS_NOT_AVAILABLE;
14  // Input Validation:
15      // Check the authPolicy consistent with hash algorithm
16      if(in->authPolicy.t.size != CryptHashGetDigestSize(in->hashAlg))
17          return TPM_RCS_SIZE + RC_PCR_SetAuthPolicy_authPolicy;
18      // If PCR does not belong to a policy group, return TPM_RC_VALUE
19      if(!PCRBelongsPolicyGroup(in->pcrNum, &groupIndex))
20          return TPM_RCS_VALUE + RC_PCR_SetAuthPolicy_pcrNum;
21  // Internal Data Update
22      // Set PCR policy
23      gp.pcrPolicies.hashAlg[groupIndex] = in->hashAlg;
24      gp.pcrPolicies.policy[groupIndex] = in->authPolicy;
25      // Save new policy to NV
26      NV_SYNC_PERSISTENT(pcrPolicies);
27      return TPM_RC_SUCCESS;
28  }
29  #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 113 — 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 114 — TPM2_PCR_SetAuthValue Response

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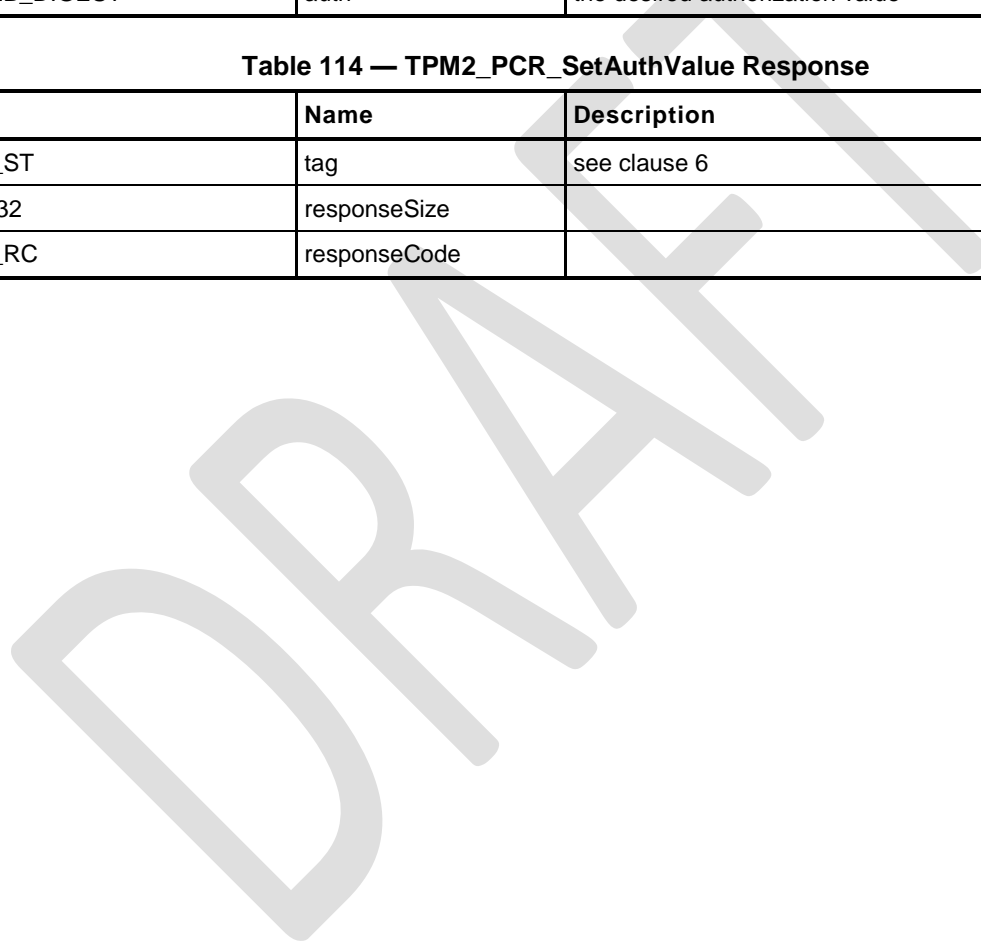
22.7.3   Detailed Actions

```
1   #include "Tpm.h"
2   #include "PCR_SetAuthValue_fp.h"
3   #ifdef TPM_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

```
4   TPM_RC
5   TPM2_PCR_SetAuthValue(
6       PCR_SetAuthValue_In     *in              // IN: input parameter list
7       )
8   {
9       UINT32      groupIndex;
10  // Input Validation:
11      // If PCR does not belong to an auth group, return TPM_RC_VALUE
12      if(!PCRBelongsAuthGroup(in->pcrHandle, &groupIndex))
13          return TPM_RC_VALUE;
14      // The command may cause the orderlyState to be cleared due to the update of
15      // state clear data.  If this is the case, Check if NV is available.
16      // A TPM_RC_NV_UNAVAILABLE or TPM_RC_NV_RATE error may be returned at
17      // this point
18      RETURN_IF_ORDERLY;
19  // Internal Data Update
20      // Set PCR authValue
21      MemoryRemoveTrailingZeros(&in->auth);
22      gc.pcrAuthValues.auth[groupIndex] = in->auth;
23      return TPM_RC_SUCCESS;
24  }
25  #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 115 — 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 116 — TPM2_PCR_Reset Response

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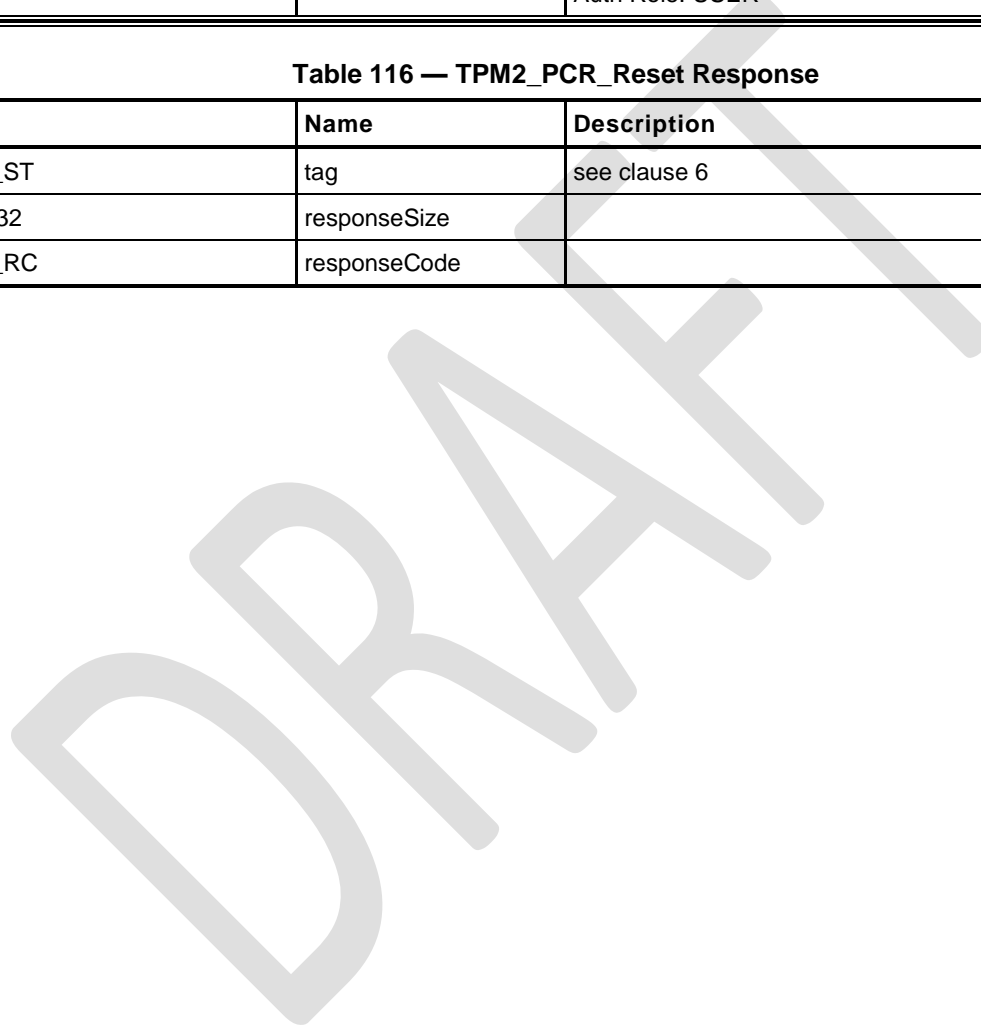
22.8.3   Detailed Actions

```
1   #include "Tpm.h"
2   #include "PCR_Reset_fp.h"
3   #ifdef TPM_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*

```
4   TPM_RC
5   TPM2_PCR_Reset(
6       PCR_Reset_In    *in              // IN: input parameter list
7       )
8   {
9   // Input Validation
10      // Check if the reset operation is allowed by the current command locality
11      if(!PCRIsResetAllowed(in->pcrHandle))
12          return TPM_RC_LOCALITY;
13      // If PCR is state saved and we need to update orderlyState, check NV
14      // availability
15      if(PCRIsStateSaved(in->pcrHandle))
16          RETURN_IF_ORDERLY;
17  // Internal Data Update
18      // Reset selected PCR in all banks to 0
19      PCRSetValue(in->pcrHandle, 0);
20      // Indicate that the PCR changed so that pcrCounter will be incremented if
21      // necessary.
22      PCRChanged(in->pcrHandle);
23      return TPM_RC_SUCCESS;
24  }
25  #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

```
1   #include "Tpm.h"
```

This function is called to process a _TPM_Hash_Start() indication.

```
2    LIB_EXPORT void
3    _TPM_Hash_Start(
4        void
5        )
6    {
7        TPM_RC              result;
8        TPMI_DH_OBJECT      handle;
9        // If a DRTM sequence object exists, free it up
10       if(g_DRTMHandle != TPM_RH_UNASSIGNED)
11       {
12           FlushObject(g_DRTMHandle);
13           g_DRTMHandle = TPM_RH_UNASSIGNED;
14       }
15       // Create an event sequence object and store the handle in global
16       // g_DRTMHandle. A TPM_RC_OBJECT_MEMORY error may be returned at this point
17       // The NULL value for the first parameter will cause the sequence structure to
18       // be allocated without being set as present. This keeps the sequence from
19       // being left behind if the sequence is terminated early.
20       result = ObjectCreateEventSequence(NULL, &g_DRTMHandle);
21       // If a free slot was not available, then free up a slot.
22       if(result != TPM_RC_SUCCESS)
23       {
24           // An implementation does not need to have a fixed relationship between
25           // slot numbers and handle numbers. To handle the general case, scan for
26           // a handle that is assigned and free it for the DRTM sequence.
27           // In the reference implementation, the relationship between handles and
28           // slots is fixed. So, if the call to ObjectCreateEvenSequence()
29           // failed indicating that all slots are occupied, then the first handle we
30           // are going to check (TRANSIENT_FIRST) will be occupied. It will be freed
31           // so that it can be assigned for use as the DRTM sequence object.
32           for(handle = TRANSIENT_FIRST; handle < TRANSIENT_LAST; handle++)
33           {
34               // try to flush the first object
35               if(IsObjectPresent(handle))
36                   break;
37           }
38           // If the first call to find a slot fails but none of the slots is occupied
39           // then there's a big problem
40           pAssert(handle < TRANSIENT_LAST);
41           // Free the slot
42           FlushObject(handle);
43           // Try to create an event sequence object again.  This time, we must
44           // succeed.
45           result = ObjectCreateEventSequence(NULL, &g_DRTMHandle);
46           if(result != TPM_RC_SUCCESS)
47               FAIL(FATAL_ERROR_INTERNAL);
48       }
49       return;
50   }
```

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

```
1    #include "Tpm.h"
```

This function is called to process a _TPM_Hash_Data() indication.

```
2    LIB_EXPORT void
3    _TPM_Hash_Data(
4        uint32_t         dataSize,       // IN: size of data to be extend
5        unsigned char   *data           // IN: data buffer
6        )
7    {
8        UINT32          i;
9        HASH_OBJECT    *hashObject;
10       TPMI_DH_PCR     pcrHandle = TPMIsStarted()
11           ? PCR_FIRST + DRTM_PCR : PCR_FIRST + HCRTM_PCR;
12   // If there is no DRTM sequence object, then _TPM_Hash_Start
13   // was not called so this function returns without doing
14   // anything.
15       if(g_DRTMHandle == TPM_RH_UNASSIGNED)
16           return;
17       hashObject = (HASH_OBJECT *)HandleToObject(g_DRTMHandle);
18       pAssert(hashObject->attributes.eventSeq);
19       // For each of the implemented hash algorithms, update the digest with the
20       // data provided.
21       for(i = 0; i < HASH_COUNT; i++)
22       {
23           // make sure that the PCR is implemented for this algorithm
24           if(PcrIsAllocated(pcrHandle,
25                           hashObject->state.hashState[i].hashAlg))
26             // Update sequence object
27               CryptDigestUpdate(&hashObject->state.hashState[i], dataSize, data);
28       }
29       return;
30   }
```

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]).

$$PCR[17][hashAlg] := \mathbf{H}_{hashAlg}\left(initial\_value \,||\, \mathbf{H}_{hashAlg}\left(hash\_data\right)\right) \tag{7}$$

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].

$$PCR[0][hashAlg] := \mathbf{H}_{hashAlg}\left(0...04 \,||\, \mathbf{H}_{hashAlg}\left(hash\_data\right)\right) \tag{8}$$

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

```
1    #include "Tpm.h"
```

This function is called to process a _TPM_Hash_End() indication.

```
2    LIB_EXPORT void
3    _TPM_Hash_End(
4        void
5        )
6    {
7        UINT32          i;
8        TPM2B_DIGEST    digest;
9        HASH_OBJECT     *hashObject;
10       TPMI_DH_PCR     pcrHandle;
11       // If the DRTM handle is not being used, then either _TPM_Hash_Start has not
12       // been called, _TPM_Hash_End was previously called, or some other command
13       // was executed and the sequence was aborted.
14       if(g_DRTMHandle == TPM_RH_UNASSIGNED)
15           return;
16       // Get DRTM sequence object
17       hashObject = (HASH_OBJECT *)HandleToObject(g_DRTMHandle);
18       // Is this _TPM_Hash_End after Startup or before
19       if(TPMIsStarted())
20       {
21           // After
22           // Reset the DRTM PCR
23           PCRResetDynamics();
24           // Extend the DRTM_PCR.
25           pcrHandle = PCR_FIRST + DRTM_PCR;
26           // DRTM sequence increments restartCount
27           gr.restartCount++;
28       }
29       else
30       {
31           pcrHandle = PCR_FIRST + HCRTM_PCR;
32           g_DrtmPreStartup = TRUE;
33       }
34       // Complete hash and extend PCR, or if this is an HCRTM, complete
35       // the hash, reset the H-CRTM register (PCR[0]) to 0...04, and then
36       // extend the H-CRTM data
37       for(i = 0; i < HASH_COUNT; i++)
38       {
39           TPMI_ALG_HASH        hash = CryptHashGetAlgByIndex(i);
40           // make sure that the PCR is implemented for this algorithm
41           if(PcrIsAllocated(pcrHandle,
42                         hashObject->state.hashState[i].hashAlg))
43           {
44               // Complete hash
45               digest.t.size = CryptHashGetDigestSize(hash);
46               CryptHashEnd2B(&hashObject->state.hashState[i], &digest.b);
47               PcrDrtm(pcrHandle, hash, &digest);
48           }
49       }
50       // Flush sequence object.
51       FlushObject(g_DRTMHandle);
52       g_DRTMHandle = TPM_RH_UNASSIGNED;
53       return;
54   }
```

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. This parameter is required to be present if expiration is non-zero (TPM_RC_EXPIRED).

b) *expiration* – If this parameter is not zero, then its absolute value is compared to the time in seconds since the *policySession→nonceTPM* was generated. If more time has passed than indicated in *expiration,* the TPM shall return TPM_RC_EXPIRED. If *nonceTPM* is the Empty buffer, and *expiration* is non-zero, then the TPM shall return TPM_RC_EXPIRED.

If *policySession→timeout* is greater than *policySession→startTime* plus the absolute value of *expiration,* then *policySession→timeout* is set to *policySession→startTime* plus the absolute value of expiration. That is, *policySession→timeout* can only be changed to a smaller value.

c) *timeout* – This parameter is compared to the current TPM time. If *policySession→timeout* is in the past, then the TPM shall return TPM_RC_EXPIRED.

> NOTE 1   The *expiration* parameter is present in the TPM2_PolicySigned and TPM2_PolicySecret command and *timeout* is the analogous parameter in the TPM2_PolicyTicket command.

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:

$$\textbf{PolicyUpdate}(commandCode, arg2, arg3) \tag{9}$$

where

arg2	a TPM2B_NAME
arg3	a TPM2B

These parameters are used to update *policySession→policyDigest* by

$$policyDigest_{new} := \mathbf{H}_{policyAlg}(policyDigest_{old} \parallel commandCode \parallel arg2.name) \tag{10}$$

followed by

$$policyDigest_{new+1} := \mathbf{H}_{policyAlg}(policyDigest_{new} \parallel arg3.buffer) \tag{11}$$

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 insures 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*, and the nonceTPM matches *policySession->nonceTPM*, then the TPM will return a ticket that includes a value indicating when the authorization expires. The TPM will also return a ticket when nonceTPM is not included and *expiration* is non-zero. Otherwise, the TPM will return a NULL ticket.

The required computation for the digest in the authorization ticket is:

$$\text{HMAC}(proof, \text{H}_{policyAlg}(ticketType \,||\, timeout \,||\, timeEpoch \,||\, cpHashA \,||\, policyRef \,||\, authObject{\rightarrow}Name)) \tag{12}$$

where

proof	secret associated with the hierarchy of the object that provided the authorization
$H_{policyAlg}$	hash function using the hash algorithm associated with the policy session
ticketType	either TPM_ST_AUTH_SECRET or TPM_ST_AUTH_SIGNED, used to indicate type of the ticket
NOTE 1	If the ticket is produced by TPM2_PolicySecret() then *ticketType* is TPM_ST_AUTH_SECRET and if produced by TPM2_PolicySigned() then *ticketType* is TPM_ST_AUTH_SIGNED.
timeout	implementation-specific representation of the expiration time of the ticket; required to be the implementation equivalent of *policySession→startTime* plus the absolute value of *expiration*
NOTE 2	*timeout* is not the same as *expiration*. The *expiration* value in the *aHash* is a relative time, using the creation time of the authorization session (TPM2_StartAuthSession()) as its reference. The *timeout* parameter is an absolute time, using TPM *Clock* as the reference.
timeEpoch	implementation-specific representation of the *timeEpoch* at the time the ticket was created
NOTE 3	When *timeout* is zero, a value of 0 should be used for the *timeEpoch*.
cpHashA	the command parameter digest for the command being authorized; computed using the hash algorithm of the policy session
policyRef	the commands that use this function have a *policyRef* parameter and the value of that parameter is used here
authObject→Name	Name associated with the *authObject* parameter

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:

$$aHash := \mathbf{H}_{authAlg}(nonceTPM \, || \, expiration \, || \, cpHashA \, || \, policyRef) \tag{13}$$

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.
NOTE 2	This parameter must be present if *expiration* is non-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 EmptyAuth 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.
EXAMPLE	The computation for an *aHash* if there are no restrictions is:

$$aHash := \mathbf{H}_{authAlg}(00\ 00\ 00\ 00_{16})$$

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).

$$\mathbf{PolicyUpdate}(\text{TPM\_CC\_PolicySigned}, authObject{\rightarrow}Name, policyRef) \qquad (14)$$

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 117 — 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 118 — 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

```
1   #include "Tpm.h"
2   #include "Policy_spt_fp.h"
3   #include "PolicySigned_fp.h"
4   #ifdef TPM_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_VALUE	input *policyID* or expiration does not match the internal data in policy session

```
5    TPM_RC
6    TPM2_PolicySigned(
7        PolicySigned_In    *in,           // IN: input parameter list
8        PolicySigned_Out   *out           // OUT: output parameter list
9        )
10   {
11       TPM_RC                   result = TPM_RC_SUCCESS;
12       SESSION                 *session;
13       TPM2B_NAME               entityName;
14       TPM2B_DIGEST             authHash;
15       HASH_STATE               hashState;
16       UINT64                   authTimeout = 0;
17   // Input Validation
18       // Set up local pointers
19       session = SessionGet(in->policySession);     // the session structure
20       // Only do input validation if this is not a trial policy session
21       if(session->attributes.isTrialPolicy == CLEAR)
22       {
23           authTimeout = ComputeAuthTimeout(session, in->expiration, &in->nonceTPM);
24           result = PolicyParameterChecks(session, authTimeout,
25                                     &in->cpHashA, &in->nonceTPM,
26                                     RC_PolicySigned_nonceTPM,
27                                     RC_PolicySigned_cpHashA,
28                                     RC_PolicySigned_expiration);
29           if(result != TPM_RC_SUCCESS)
30               return result;
31           // Re-compute the digest being signed
32           /*(See part 3 specification)
33           // The digest is computed as:
34           //     aHash := hash ( nonceTPM | expiration | cpHashA | policyRef)
35           //   where:
36           //       hash()     the hash associated with the signed authorization
37           //       nonceTPM   the nonceTPM value from the TPM2_StartAuthSession .
38           //                  response If the authorization is not limited to this
39           //                  session, the size of this value is zero.
40           //       expiration time limit on authorization set by authorizing object.
41           //                  This 32-bit value is set to zero if the expiration
42           //                  time is not being set.
43           //       cpHashA    hash of the command parameters for the command being
44           //                  approved using the hash algorithm of the PSAP session.
45           //                  Set to NULLauth if the authorization is not limited
```

```
46              //                  to a specific command.
47              //      policyRef    hash of an opaque value determined by the authorizing
48              //                   object.  Set to the NULLdigest if no hash is present.
49              */
50              // Start hash
51              authHash.t.size = CryptHashStart(&hashState,
52                                          CryptGetSignHashAlg(&in->auth));
53              // If there is no digest size, then we don't have a verification function
54              // for this algorithm (e.g. TPM_ALG_ECDAA) so indicate that it is a
55              // bad scheme.
56              if(authHash.t.size == 0)
57                  return TPM_RCS_SCHEME + RC_PolicySigned_auth;
58              // nonceTPM
59              CryptDigestUpdate2B(&hashState, &in->nonceTPM.b);
60              // expiration
61              CryptDigestUpdateInt(&hashState, sizeof(UINT32), in->expiration);
62              // cpHashA
63              CryptDigestUpdate2B(&hashState, &in->cpHashA.b);
64              // policyRef
65              CryptDigestUpdate2B(&hashState, &in->policyRef.b);
66              // Complete digest
67              CryptHashEnd2B(&hashState, &authHash.b);
68              // Validate Signature.  A TPM_RC_SCHEME, TPM_RC_HANDLE or TPM_RC_SIGNATURE
69              // error may be returned at this point
70              result = CryptValidateSignature(in->authObject, &authHash, &in->auth);
71              if(result != TPM_RC_SUCCESS)
72                  return RcSafeAddToResult(result, RC_PolicySigned_auth);
73          }
74  // Internal Data Update
75      // Update policy with input policyRef and name of authorization key
76      // These values are updated even if the session is a trial session
77      PolicyContextUpdate(TPM_CC_PolicySigned,
78                      EntityGetName(in->authObject, &entityName),
79                      &in->policyRef,
80                      &in->cpHashA, authTimeout, session);
81  // Command Output
82      // Create ticket and timeout buffer if in->expiration < 0 and this is not
83      // a trial session.
84      // NOTE: PolicyParameterChecks() makes sure that nonceTPM is present
85      // when expiration is non-zero.
86      if(in->expiration < 0
87          && session->attributes.isTrialPolicy == CLEAR)
88      {
89          BOOL        expiresOnReset = (in->nonceTPM.t.size == 0);
90          // Generate timeout buffer.  The format of output timeout buffer is
91          // TPM-specific.
92          // In this implementation, the timeout parameter is the timeout relative
93          // to g_time with a one byte flag to indicate if the ticket will expire on
94          // TPM Reset
95          out->timeout.t.size = sizeof(authTimeout) + 1;
96          UINT64_TO_BYTE_ARRAY(authTimeout, out->timeout.t.buffer);
97          out->timeout.t.buffer[sizeof(authTimeout)] = (BYTE)expiresOnReset;
98          // Compute policy ticket
99          TicketComputeAuth(TPM_ST_AUTH_SIGNED, EntityGetHierarchy(in->authObject),
100                         authTimeout, expiresOnReset, &in->cpHashA, &in->policyRef,
101                         &entityName, &out->policyTicket);
102      }
103      else
104      {
105          // Generate a null ticket.
106          // timeout buffer is null
107          out->timeout.t.size = 0;
108          // authorization ticket is null
109          out->policyTicket.tag = TPM_ST_AUTH_SIGNED;
110          out->policyTicket.hierarchy = TPM_RH_NULL;
111          out->policyTicket.digest.t.size = 0;
```

```
112         }
113     return TPM_RC_SUCCESS;
114 }
115 #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. An object must have *userWithAuth* SET. An NV Index must have TPMA_NV_AUTHREAD SET.

NOTE 1   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.

nonceTPM must be present if *expiration* is non-zero.

When all validations have succeeded, *policySession→policyDigest* is updated by **PolicyUpdate**() (see 23.2.3).

$$\textbf{PolicyUpdate}(\text{TPM\_CC\_PolicySecret}, \textit{authObject→Name, policyRef}) \qquad (15)$$

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 119 — 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 120 — 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; this ticket will use the TPMT_ST_AUTH_SECRET structure tag
TPMT_TK_AUTH	policyTicket	produced if the command succeeds and *expiration* in the command was non-zero. See 23.2.5

23.4.3   Detailed Actions

```
1    #include "Tpm.h"
2    #include "PolicySecret_fp.h"
3    #ifdef TPM_CC_PolicySecret  // Conditional expansion of this file
4    #include "Policy_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_TYPE	*authHandle* references an object loaded using TMP2_LoadExternal().
TPM_RC_VALUE	input *policyID* or expiration does not match the internal data in policy session

```
5    TPM_RC
6    TPM2_PolicySecret(
7        PolicySecret_In    *in,                // IN: input parameter list
8        PolicySecret_Out   *out                // OUT: output parameter list
9        )
10   {
11       TPM_RC                      result;
12       SESSION                     *session;
13       TPM2B_NAME                  entityName;
14       UINT64                      authTimeout = 0;
15   // Input Validation
16       // Get pointer to the session structure
17       session = SessionGet(in->policySession);
18       //Only do input validation if this is not a trial policy session
19       if(session->attributes.isTrialPolicy == CLEAR)
20       {
21           authTimeout = ComputeAuthTimeout(session, in->expiration, &in->nonceTPM);
22           result = PolicyParameterChecks(session, authTimeout,
23                                   &in->cpHashA, &in->nonceTPM,
24                                   RC_PolicySecret_nonceTPM,
25                                   RC_PolicySecret_cpHashA,
26                                   RC_PolicySecret_expiration);
27           if(result != TPM_RC_SUCCESS)
28               return result;
29       }
30   // Internal Data Update
31       // Update policy context with input policyRef and name of authorizing key
32       // This value is computed even for trial sessions. Possibly update the cpHash
33       PolicyContextUpdate(TPM_CC_PolicySecret,
34                       EntityGetName(in->authHandle, &entityName), &in->policyRef,
35                       &in->cpHashA, authTimeout, session);
36   // Command Output
37       // Create ticket and timeout buffer if in->expiration < 0 and this is not
38       // a trial session.
39       // NOTE: PolicyParameterChecks() makes sure that nonceTPM is present
40       // when expiration is non-zero.
41       if(in->expiration < 0
42          && session->attributes.isTrialPolicy == CLEAR)
43       {
44           BOOL        expiresOnReset = (in->nonceTPM.t.size == 0);
45           // Generate timeout buffer.  The format of output timeout buffer is
46           // TPM-specific.
```

```
47            // In this implementation, the timeout parameter is the timeout relative
48            // to g_time with a one byte flag to indicate if the ticket will expire on
49            // TPM Reset
50            out->timeout.t.size = sizeof(authTimeout) + 1;
51            UINT64_TO_BYTE_ARRAY(authTimeout, out->timeout.t.buffer);
52            out->timeout.t.buffer[sizeof(authTimeout)] = (BYTE)expiresOnReset;
53            // Compute policy ticket
54            TicketComputeAuth(TPM_ST_AUTH_SECRET, EntityGetHierarchy(in->authHandle),
55                              authTimeout, expiresOnReset, &in->cpHashA, &in->policyRef,
56                              &entityName, &out->policyTicket);
57        }
58    else
59        {
60            // timeout buffer is null
61            out->timeout.t.size = 0;
62            // authorization ticket is null
63            out->policyTicket.tag = TPM_ST_AUTH_SECRET;
64            out->policyTicket.hierarchy = TPM_RH_NULL;
65            out->policyTicket.digest.t.size = 0;
66        }
67    return TPM_RC_SUCCESS;
68 }
69 #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 121 — 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 122 — TPM2_PolicyTicket Response

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

23.5.3   Detailed Actions

```
1   #include "Tpm.h"
2   #include "PolicyTicket_fp.h"
3   #ifdef TPM_CC_PolicyTicket  // Conditional expansion of this file
4   #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

```
5   TPM_RC
6   TPM2_PolicyTicket(
7       PolicyTicket_In     *in                // IN: input parameter list
8       )
9   {
10      TPM_RC                  result;
11      SESSION                *session;
12      UINT64                  authTimeout;
13      TPMT_TK_AUTH            ticketToCompare;
14      TPM_CC                  commandCode = TPM_CC_PolicySecret;
15      BOOL                    expiresOnReset;
16  // Input Validation
17      // Get pointer to the session structure
18      session = SessionGet(in->policySession);
19      // NOTE: A trial policy session is not allowed to use this command.
20      // A ticket is used in place of a previously given authorization. Since
21      // a trial policy doesn't actually authenticate, the validated
22      // ticket is not necessary and, in place of using a ticket, one
23      // should use the intended authorization for which the ticket
24      // would be a substitute.
25      if(session->attributes.isTrialPolicy)
26          return TPM_RCS_ATTRIBUTES + RC_PolicyTicket_policySession;
27      // Restore timeout data.  The format of timeout buffer is TPM-specific.
28      // In this implementation, we simply copy the value of timeout to the
29      // buffer.
30      if(in->timeout.t.size != sizeof(UINT64) + 1)
31          return TPM_RCS_SIZE + RC_PolicyTicket_timeout;
32      authTimeout = BYTE_ARRAY_TO_UINT64(in->timeout.t.buffer);
33      expiresOnReset = in->timeout.t.buffer[sizeof(authTimeout)];
34      // Do the normal checks on the cpHashA and timeout values
35      result = PolicyParameterChecks(session, authTimeout,
36                                     &in->cpHashA,
37                                     NULL,                    // no nonce
38                                     0,                       // no bad nonce return
39                                     RC_PolicyTicket_cpHashA,
40                                     RC_PolicyTicket_timeout);
41      if(result != TPM_RC_SUCCESS)
42          return result;
43      // Validate Ticket
44      // Re-generate policy ticket by input parameters
45      TicketComputeAuth(in->ticket.tag, in->ticket.hierarchy,
46                      authTimeout, expiresOnReset, &in->cpHashA, &in->policyRef,
47                      &in->authName, &ticketToCompare);
48      // Compare generated digest with input ticket digest
49      if(!MemoryEqual2B(&in->ticket.digest.b, &ticketToCompare.digest.b))
50          return TPM_RCS_TICKET + RC_PolicyTicket_ticket;
51  // Internal Data Update
52      // Is this ticket to take the place of a TPM2_PolicySigned() or
```

```
53          // a TPM2_PolicySecret()?
54          if(in->ticket.tag == TPM_ST_AUTH_SIGNED)
55              commandCode = TPM_CC_PolicySigned;
56          else if(in->ticket.tag == TPM_ST_AUTH_SECRET)
57              commandCode = TPM_CC_PolicySecret;
58          else
59              // There could only be two possible tag values.  Any other value should
60              // be caught by the ticket validation process.
61              FAIL(FATAL_ERROR_INTERNAL);
62          // Update policy context
63          PolicyContextUpdate(commandCode, &in->authName, &in->policyRef,
64                              &in->cpHashA, authTimeout, session);
65          return TPM_RC_SUCCESS;
66      }
67  #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*:

$$digests \coloneqq pHashList.digests[1].buffer \mathbin{||} ... \mathbin{||} pHashList.digests[n].buffer \qquad (17)$$

> 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:

$$policyDigest_{new} \coloneqq \mathbf{H}_{policyAlg}(policyDigest_{old} \mathbin{||} \text{TPM\_CC\_PolicyOR} \mathbin{||} digests) \qquad (18)$$

> NOTE 2   The computation in b) and c) above is equivalent to:
> $$policyDigest_{new} \coloneqq \mathbf{H}_{policyAlg}(0...0 \mathbin{||} \text{TPM\_CC\_PolicyOR} \mathbin{||} digests)$$

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 123 — 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 124 — TPM2_PolicyOR Response

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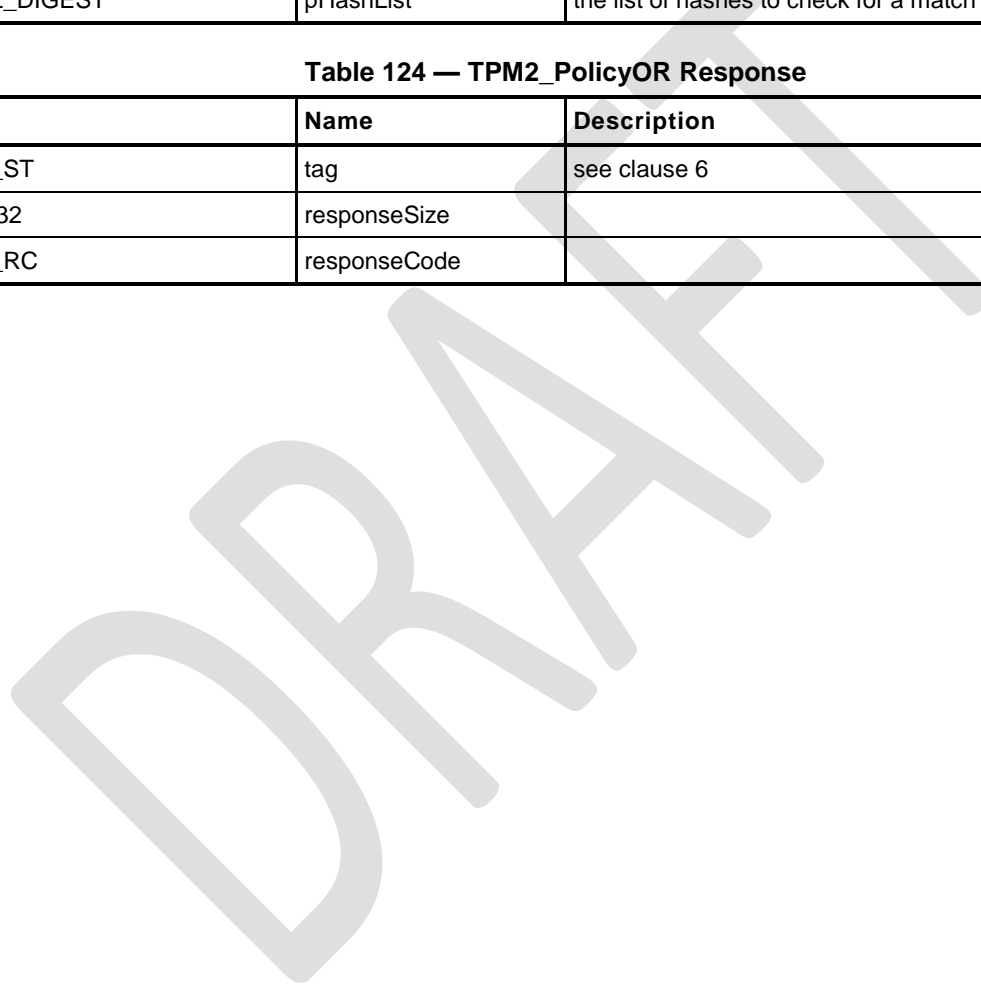
23.6.3   Detailed Actions

```
1   #include "Tpm.h"
2   #include "PolicyOR_fp.h"
3   #ifdef TPM_CC_PolicyOR  // Conditional expansion of this file
4   #include "Policy_spt_fp.h"
```

Error Returns	Meaning
TPM_RC_VALUE	no digest in *pHashList* matched the current value of *policyDigest* for *policySession*

```
5   TPM_RC
6   TPM2_PolicyOR(
7       PolicyOR_In     *in              // IN: input parameter list
8       )
9   {
10      SESSION     *session;
11      UINT32          i;
12  // Input Validation and Update
13      // Get pointer to the session structure
14      session = SessionGet(in->policySession);
15      // Compare and Update Internal Session policy if match
16      for(i = 0; i < in->pHashList.count; i++)
17      {
18          if(session->attributes.isTrialPolicy == SET
19              || (MemoryEqual2B(&session->u2.policyDigest.b,
20                          &in->pHashList.digests[i].b)))
21          {
22              // Found a match
23              HASH_STATE      hashState;
24              TPM_CC          commandCode = TPM_CC_PolicyOR;
25              // Start hash
26              session->u2.policyDigest.t.size
27                  = CryptHashStart(&hashState, session->authHashAlg);
28              // Set policyDigest to 0 string and add it to hash
29              MemorySet(session->u2.policyDigest.t.buffer, 0,
30                      session->u2.policyDigest.t.size);
31              CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);
32              // add command code
33              CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), commandCode);
34              // Add each of the hashes in the list
35              for(i = 0; i < in->pHashList.count; i++)
36              {
37                  // Extend policyDigest
38                  CryptDigestUpdate2B(&hashState, &in->pHashList.digests[i].b);
39              }
40              // Complete digest
41              CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);
42              return TPM_RC_SUCCESS;
43          }
44      }
45      // None of the values in the list matched the current policyDigest
46      return TPM_RCS_VALUE + RC_PolicyOR_pHashList;
47  }
48  #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:

$$policyDigest_{new} := \mathbf{H}_{policyAlg}(policyDigest_{old} \,||\, \text{TPM\_CC\_PolicyPCR} \,||\, pcrs \,||\, digestTPM) \qquad (19)$$

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:

$$policyDigest_{new} := \mathbf{H}_{policyAlg}(policyDigest_{old} \,||\, \text{TPM\_CC\_PolicyPCR} \,||\, pcrs \,||\, pcrDigest) \qquad (20)$$

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 125 — 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 126 — TPM2_PolicyPCR Response

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

23.7.3 Detailed Actions

```
1   #include "Tpm.h"
2   #include "PolicyPCR_fp.h"
3   #ifdef TPM_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

```
4    TPM_RC
5    TPM2_PolicyPCR(
6        PolicyPCR_In    *in                // IN: input parameter list
7        )
8    {
9        SESSION         *session;
10       TPM2B_DIGEST     pcrDigest;
11       BYTE             pcrs[sizeof(TPML_PCR_SELECTION)];
12       UINT32           pcrSize;
13       BYTE            *buffer;
14       TPM_CC           commandCode = TPM_CC_PolicyPCR;
15       HASH_STATE       hashState;
16   // Input Validation
17       // Get pointer to the session structure
18       session = SessionGet(in->policySession);
19       // Do validation for non trial session
20       if(session->attributes.isTrialPolicy == CLEAR)
21       {
22           // Make sure that this is not going to invalidate a previous PCR check
23           if(session->pcrCounter != 0 && session->pcrCounter != gr.pcrCounter)
24               return TPM_RC_PCR_CHANGED;
25           // Compute current PCR digest
26           PCRComputeCurrentDigest(session->authHashAlg, &in->pcrs, &pcrDigest);
27           // If the caller specified the PCR digest and it does not
28           // match the current PCR settings, return an error..
29           if(in->pcrDigest.t.size != 0)
30           {
31               if(!MemoryEqual2B(&in->pcrDigest.b, &pcrDigest.b))
32                   return TPM_RCS_VALUE + RC_PolicyPCR_pcrDigest;
33           }
34       }
35       else
36       {
37           // For trial session, just use the input PCR digest
38           pcrDigest = in->pcrDigest;
39       }
40   // Internal Data Update
41       // Update policy hash
42       // policyDigestnew = hash(   policyDigestold || TPM_CC_PolicyPCR
43       //                          || PCRS || pcrDigest)
44       //  Start hash
45       CryptHashStart(&hashState, session->authHashAlg);
46       //  add old digest
47       CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);
48       //  add commandCode
49       CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), commandCode);
50       //  add PCRS
51       buffer = pcrs;
52       pcrSize = TPML_PCR_SELECTION_Marshal(&in->pcrs, &buffer, NULL);
53       CryptDigestUpdate(&hashState, pcrSize, pcrs);
54       //  add PCR digest
55       CryptDigestUpdate2B(&hashState, &pcrDigest.b);
56       //  complete the hash and get the results
```

```
57          CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);
58          //  update pcrCounter in session context for non trial session
59          if(session->attributes.isTrialPolicy == CLEAR)
60          {
61              session->pcrCounter = gr.pcrCounter;
62          }
63          return TPM_RC_SUCCESS;
64      }
65  #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

$$policyDigest_{new} := \mathbf{H}_{policyAlg}(policyDigest_{old} \,||\, \text{TPM\_CC\_PolicyLocality} \,||\, locality) \tag{21}$$

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 127 — 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 128 — TPM2_PolicyLocality Response

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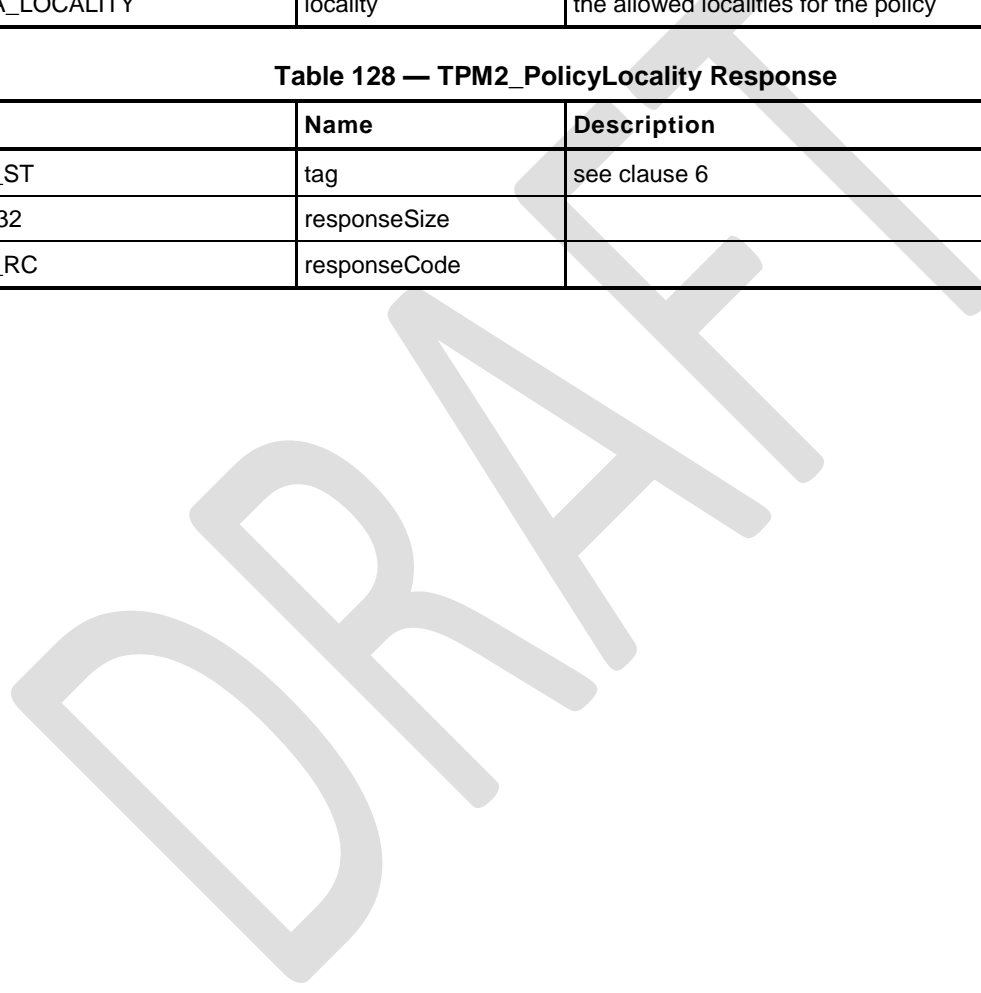
23.8.3   Detailed Actions

```
1    #include "Tpm.h"
2    #include "PolicyLocality_fp.h"
3    #ifdef TPM_CC_PolicyLocality  // Conditional expansion of this file
```

Limit a policy to a specific locality

Error Returns	Meaning
TPM_RC_RANGE	all the locality values selected by *locality* have been disabled by previous TPM2_PolicyLocality() calls.

```
4    TPM_RC
5    TPM2_PolicyLocality(
6        PolicyLocality_In   *in              // IN: input parameter list
7        )
8    {
9        SESSION       *session;
10       BYTE           marshalBuffer[sizeof(TPMA_LOCALITY)];
11       BYTE           prevSetting[sizeof(TPMA_LOCALITY)];
12       UINT32         marshalSize;
13       BYTE          *buffer;
14       TPM_CC         commandCode = TPM_CC_PolicyLocality;
15       HASH_STATE    hashState;
16   // Input Validation
17       // Get pointer to the session structure
18       session = SessionGet(in->policySession);
19       // Get new locality setting in canonical form
20       marshalBuffer[0] = 0;   // Code analysis says that this is not initialized
21       buffer = marshalBuffer;
22       marshalSize = TPMA_LOCALITY_Marshal(&in->locality, &buffer, NULL);
23       // Its an error if the locality parameter is zero
24       if(marshalBuffer[0] == 0)
25           return TPM_RCS_RANGE + RC_PolicyLocality_locality;
26       // Get existing locality setting in canonical form
27       prevSetting[0] = 0;        // Code analysis says that this is not initialized
28       buffer = prevSetting;
29       TPMA_LOCALITY_Marshal(&session->commandLocality, &buffer, NULL);
30       // If the locality has previously been set
31       if(prevSetting[0] != 0
32           // then the current locality setting and the requested have to be the same
33           // type (that is, either both normal or both extended
34           && ((prevSetting[0] < 32) != (marshalBuffer[0] < 32)))
35           return TPM_RCS_RANGE + RC_PolicyLocality_locality;
36       // See if the input is a regular or extended locality
37       if(marshalBuffer[0] < 32)
38       {
39           // if there was no previous setting, start with all normal localities
40           // enabled
41           if(prevSetting[0] == 0)
42               prevSetting[0] = 0x1F;
43           // AND the new setting with the previous setting and store it in prevSetting
44           prevSetting[0] &= marshalBuffer[0];
45           // The result setting can not be 0
46           if(prevSetting[0] == 0)
47               return TPM_RCS_RANGE + RC_PolicyLocality_locality;
48       }
49       else
50       {
51           // for extended locality
52           // if the locality has already been set, then it must match the
53           if(prevSetting[0] != 0 && prevSetting[0] != marshalBuffer[0])
54               return TPM_RCS_RANGE + RC_PolicyLocality_locality;
55           // Setting is OK
```

```
56          prevSetting[0] = marshalBuffer[0];
57      }
58  // Internal Data Update
59      // Update policy hash
60      // policyDigestnew = hash(policyDigestold || TPM_CC_PolicyLocality || locality)
61      // Start hash
62      CryptHashStart(&hashState, session->authHashAlg);
63      // add old digest
64      CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);
65      // add commandCode
66      CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), commandCode);
67      // add input locality
68      CryptDigestUpdate(&hashState, marshalSize, marshalBuffer);
69      // complete the digest
70      CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);
71      // update session locality by unmarshal function.  The function must succeed
72      // because both input and existing locality setting have been validated.
73      buffer = prevSetting;
74      TPMA_LOCALITY_Unmarshal(&session->commandLocality, &buffer,
75                          (INT32 *)&marshalSize);
76      return TPM_RC_SUCCESS;
77  }
78  #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, *offset* is ignored and an offset of 0 is used.

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:

$$args \coloneqq \mathbf{H}_{policyAlg}(operandB.buffer \,||\, offset \,||\, operation) \tag{22}$$

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

$$policyDigest_{new} \coloneqq \mathbf{H}_{policyAlg}(policyDigest_{old} \,||\, \text{TPM\_CC\_PolicyNV} \,||\, args \,||\, nvIndex{\rightarrow}Name) \tag{23}$$

where

$\mathbf{H}_{policyAlg}()$	hash function using the algorithm of the policy session
args	value computed in equation (22)
nvIndex→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 129 — 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 offset in the NV Index for the start of operand A
TPM_EO	operation	the comparison to make

Table 130 — TPM2_PolicyNV Response

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

23.9.3 Detailed Actions

```
1   #include "Tpm.h"
2   #include "PolicyNV_fp.h"
3   #ifdef TPM_CC_PolicyNV  // Conditional expansion of this file
4   #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

```
5   TPM_RC
6   TPM2_PolicyNV(
7       PolicyNV_In      *in               // IN: input parameter list
8       )
9   {
10      TPM_RC              result;
11      SESSION             *session;
12      NV_REF              locator;
13      NV_INDEX            *nvIndex = NvGetIndexInfo(in->nvIndex, &locator);
14      BYTE                nvBuffer[sizeof(in->operandB.t.buffer)];
15      TPM2B_NAME          nvName;
16      TPM_CC              commandCode = TPM_CC_PolicyNV;
17      HASH_STATE          hashState;
18      TPM2B_DIGEST        argHash;
19  // Input Validation
20      // Get pointer to the session structure
21      session = SessionGet(in->policySession);
22      //If this is a trial policy, skip all validations and the operation
23      if(session->attributes.isTrialPolicy == CLEAR)
24      {
25          // Common read access checks. NvReadAccessChecks() may return
26          // TPM_RC_NV_AUTHORIZATION, TPM_RC_NV_LOCKED, or TPM_RC_NV_UNINITIALIZED
27          result = NvReadAccessChecks(in->authHandle,
28                                      in->nvIndex,
29                                      nvIndex->publicArea.attributes);
30          if(result != TPM_RC_SUCCESS)
31              return result;
32          // Make sure that offset is withing range
33          if(in->offset > nvIndex->publicArea.dataSize)
34              return TPM_RCS_VALUE + RC_PolicyNV_offset;
35          // Valid NV data size should not be smaller than input operandB size
36          if((nvIndex->publicArea.dataSize - in->offset) < in->operandB.t.size)
37              return TPM_RCS_SIZE + RC_PolicyNV_operandB;
38          // Arithmetic Comparison
39          // Get NV data.  The size of NV data equals the input operand B size
40          NvGetIndexData(nvIndex,
41                      locator,
42                      in->offset,
43                      in->operandB.t.size,
44                      nvBuffer);
45          switch(in->operation)
46          {
47              case TPM_EO_EQ:
48                  // compare A = B
```

```
49                    if(UnsignedCompareB(in->operandB.t.size, nvBuffer,
50                                    in->operandB.t.size, in->operandB.t.buffer)
51                 != 0)
52                     return TPM_RC_POLICY;
53              break;
54          case TPM_EO_NEQ:
55              // compare A != B
56              if(UnsignedCompareB(in->operandB.t.size, nvBuffer,
57                                    in->operandB.t.size, in->operandB.t.buffer)
58                 == 0)
59                     return TPM_RC_POLICY;
60              break;
61          case TPM_EO_SIGNED_GT:
62              // compare A > B signed
63              if(SignedCompareB(in->operandB.t.size, nvBuffer,
64                                  in->operandB.t.size, in->operandB.t.buffer)
65                 <= 0)
66                     return TPM_RC_POLICY;
67              break;
68          case TPM_EO_UNSIGNED_GT:
69              // compare A > B unsigned
70              if(UnsignedCompareB(in->operandB.t.size, nvBuffer,
71                                    in->operandB.t.size, in->operandB.t.buffer)
72                 <= 0)
73                     return TPM_RC_POLICY;
74              break;
75          case TPM_EO_SIGNED_LT:
76              // compare A < B signed
77              if(SignedCompareB(in->operandB.t.size, nvBuffer,
78                                  in->operandB.t.size, in->operandB.t.buffer)
79                 >= 0)
80                     return TPM_RC_POLICY;
81              break;
82          case TPM_EO_UNSIGNED_LT:
83              // compare A < B unsigned
84              if(UnsignedCompareB(in->operandB.t.size, nvBuffer,
85                                    in->operandB.t.size, in->operandB.t.buffer)
86                 >= 0)
87                     return TPM_RC_POLICY;
88              break;
89          case TPM_EO_SIGNED_GE:
90              // compare A >= B signed
91              if(SignedCompareB(in->operandB.t.size, nvBuffer,
92                                  in->operandB.t.size, in->operandB.t.buffer)
93                 < 0)
94                     return TPM_RC_POLICY;
95              break;
96          case TPM_EO_UNSIGNED_GE:
97              // compare A >= B unsigned
98              if(UnsignedCompareB(in->operandB.t.size, nvBuffer,
99                                    in->operandB.t.size, in->operandB.t.buffer)
100                < 0)
101                    return TPM_RC_POLICY;
102             break;
103         case TPM_EO_SIGNED_LE:
104             // compare A <= B signed
105             if(SignedCompareB(in->operandB.t.size, nvBuffer,
106                                 in->operandB.t.size, in->operandB.t.buffer)
107                > 0)
108                    return TPM_RC_POLICY;
109             break;
110         case TPM_EO_UNSIGNED_LE:
111             // compare A <= B unsigned
112             if(UnsignedCompareB(in->operandB.t.size, nvBuffer,
113                                   in->operandB.t.size, in->operandB.t.buffer)
114                > 0)
```

```
115                          return TPM_RC_POLICY;
116                      break;
117                  case TPM_EO_BITSET:
118                      // All bits SET in B are SET in A. ((A&B)=B)
119                  {
120                      UINT32 i;
121                      for(i = 0; i < in->operandB.t.size; i++)
122                          if((nvBuffer[i] & in->operandB.t.buffer[i])
123                              != in->operandB.t.buffer[i])
124                               return TPM_RC_POLICY;
125                  }
126                  break;
127                  case TPM_EO_BITCLEAR:
128                      // All bits SET in B are CLEAR in A. ((A&B)=0)
129                  {
130                      UINT32 i;
131                      for(i = 0; i < in->operandB.t.size; i++)
132                          if((nvBuffer[i] & in->operandB.t.buffer[i]) != 0)
133                               return TPM_RC_POLICY;
134                  }
135                  break;
136                  default:
137                      FAIL(FATAL_ERROR_INTERNAL);
138                      break;
139              }
140          }
141      else
142          result = TPM_RC_SUCCESS;
143  // Internal Data Update
144      // Start argument hash
145      argHash.t.size = CryptHashStart(&hashState, session->authHashAlg);
146      //  add operandB
147      CryptDigestUpdate2B(&hashState, &in->operandB.b);
148      //  add offset
149      CryptDigestUpdateInt(&hashState, sizeof(UINT16), in->offset);
150      //  add operation
151      CryptDigestUpdateInt(&hashState, sizeof(TPM_EO), in->operation);
152      //  complete argument digest
153      CryptHashEnd2B(&hashState, &argHash.b);
154      // Update policyDigest
155      //  Start digest
156      CryptHashStart(&hashState, session->authHashAlg);
157      //  add old digest
158      CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);
159      //  add commandCode
160      CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), commandCode);
161      //  add argument digest
162      CryptDigestUpdate2B(&hashState, &argHash.b);
163      // Adding nvName
164      CryptDigestUpdate2B(&hashState, &EntityGetName(in->nvIndex, &nvName)->b);
165      // complete the digest
166      CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);
167      return result;
168  }
169  #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:

$$args := \mathbf{H}_{policyAlg}(operandB.buffer \,||\, offset \,||\, operation) \tag{24}$$

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

$$policyDigest_{new} := \mathbf{H}_{policyAlg}(policyDigest_{old} \,||\, \text{TPM\_CC\_PolicyCounterTimer} \,||\, args) \tag{25}$$

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 131 — 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 offset in TPMS_TIME_INFO structure for the start of operand A
TPM_EO	operation	the comparison to make

Table 132 — TPM2_PolicyCounterTimer Response

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

23.10.3 Detailed Actions

```
1    #include "Tpm.h"
2    #include "PolicyCounterTimer_fp.h"
3    #ifdef TPM_CC_PolicyCounterTimer  // Conditional expansion of this file
4    #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

```
5    TPM_RC
6    TPM2_PolicyCounterTimer(
7        PolicyCounterTimer_In   *in              // IN: input parameter list
8        )
9    {
10       TPM_RC               result;
11       SESSION              *session;
12       TIME_INFO            infoData;        // data buffer of  TPMS_TIME_INFO
13       TPM_CC               commandCode = TPM_CC_PolicyCounterTimer;
14       HASH_STATE           hashState;
15       TPM2B_DIGEST         argHash;
16   // Input Validation
17       // If the command is going to use any part of the counter or timer, need
18       // to verify that time is advancing.
19       // The time and clock vales are the first two 64-bit values in the clock
20       if(in->offset < sizeof(UINT64) + sizeof(UINT64))
21       {
22           // Using Clock or Time so see if clock is running. Clock doesn't run while
23           // NV is unavailable.
24           // TPM_RC_NV_UNAVAILABLE or TPM_RC_NV_RATE error may be returned here.
25           RETURN_IF_NV_IS_NOT_AVAILABLE;
26       }
27       // Get pointer to the session structure
28       session = SessionGet(in->policySession);
29       //If this is a trial policy, skip all validations and the operation
30       if(session->attributes.isTrialPolicy == CLEAR)
31       {
32           // Get time data info.  The size of time info data equals the input
33           // operand B size.  A TPM_RC_RANGE or TPM_RC_VALUE error may be returned.
34           // NOTE: The reasons that there is no check in this function that the offset
35           // is within the allowed range is because the allowed range depends on the
36           // marshaled size of the TPMS_TIME_INFO structure. The size for that is
37           // not known in this function so the offset check is deferred to
38           // TimeGetRange().
39           result = TimeGetRange(in->offset, in->operandB.t.size, &infoData);
40           if(result != TPM_RC_SUCCESS)
41           {
42               // handle offset out of range
43               if(result == TPM_RC_VALUE)
44                   return TPM_RCS_VALUE + RC_PolicyCounterTimer_offset;
45               return result;
46           }
47           // Arithmetic Comparison
48           switch(in->operation)
49           {
50               case TPM_EO_EQ:
51                   // compare A = B
52                   if(UnsignedCompareB(in->operandB.t.size, infoData,
53                                       in->operandB.t.size, in->operandB.t.buffer)
54                       != 0)
55                       return TPM_RC_POLICY;
```

```
56                  break;
57              case TPM_EO_NEQ:
58                  // compare A != B
59                  if(UnsignedCompareB(in->operandB.t.size, infoData,
60                                  in->operandB.t.size, in->operandB.t.buffer)
61                      == 0)
62                      return TPM_RC_POLICY;
63                  break;
64              case TPM_EO_SIGNED_GT:
65                  // compare A > B signed
66                  if(SignedCompareB(in->operandB.t.size, infoData,
67                                  in->operandB.t.size, in->operandB.t.buffer)
68                      <= 0)
69                      return TPM_RC_POLICY;
70                  break;
71              case TPM_EO_UNSIGNED_GT:
72                  // compare A > B unsigned
73                  if(UnsignedCompareB(in->operandB.t.size, infoData,
74                                  in->operandB.t.size, in->operandB.t.buffer)
75                      <= 0)
76                      return TPM_RC_POLICY;
77                  break;
78              case TPM_EO_SIGNED_LT:
79                  // compare A < B signed
80                  if(SignedCompareB(in->operandB.t.size, infoData,
81                                  in->operandB.t.size, in->operandB.t.buffer)
82                      >= 0)
83                      return TPM_RC_POLICY;
84                  break;
85              case TPM_EO_UNSIGNED_LT:
86                  // compare A < B unsigned
87                  if(UnsignedCompareB(in->operandB.t.size, infoData,
88                                  in->operandB.t.size, in->operandB.t.buffer)
89                      >= 0)
90                      return TPM_RC_POLICY;
91                  break;
92              case TPM_EO_SIGNED_GE:
93                  // compare A >= B signed
94                  if(SignedCompareB(in->operandB.t.size, infoData,
95                                  in->operandB.t.size, in->operandB.t.buffer)
96                      < 0)
97                      return TPM_RC_POLICY;
98                  break;
99              case TPM_EO_UNSIGNED_GE:
100                 // compare A >= B unsigned
101                 if(UnsignedCompareB(in->operandB.t.size, infoData,
102                                 in->operandB.t.size, in->operandB.t.buffer)
103                     < 0)
104                     return TPM_RC_POLICY;
105                 break;
106             case TPM_EO_SIGNED_LE:
107                 // compare A <= B signed
108                 if(SignedCompareB(in->operandB.t.size, infoData,
109                                 in->operandB.t.size, in->operandB.t.buffer)
110                     > 0)
111                     return TPM_RC_POLICY;
112                 break;
113             case TPM_EO_UNSIGNED_LE:
114                 // compare A <= B unsigned
115                 if(UnsignedCompareB(in->operandB.t.size, infoData,
116                                 in->operandB.t.size, in->operandB.t.buffer)
117                     > 0)
118                     return TPM_RC_POLICY;
119                 break;
120             case TPM_EO_BITSET:
121                 // All bits SET in B are SET in A. ((A&B)=B)
```

```
122                 {
123                     UINT32 i;
124                     for(i = 0; i < in->operandB.t.size; i++)
125                         if((infoData[i] & in->operandB.t.buffer[i])
126                             != in->operandB.t.buffer[i])
127                             return TPM_RC_POLICY;
128                 }
129                 break;
130                 case TPM_EO_BITCLEAR:
131                     // All bits SET in B are CLEAR in A.  ((A&B)=0)
132                 {
133                     UINT32 i;
134                     for(i = 0; i < in->operandB.t.size; i++)
135                         if((infoData[i] & in->operandB.t.buffer[i]) != 0)
136                             return TPM_RC_POLICY;
137                 }
138                 break;
139                 default:
140                     FAIL(FATAL_ERROR_INTERNAL);
141                     break;
142             }
143         }
144     else
145             result = TPM_RC_SUCCESS;
146 // Internal Data Update
147     // Start argument list hash
148     argHash.t.size = CryptHashStart(&hashState, session->authHashAlg);
149     //  add operandB
150     CryptDigestUpdate2B(&hashState, &in->operandB.b);
151     //  add offset
152     CryptDigestUpdateInt(&hashState, sizeof(UINT16), in->offset);
153     //  add operation
154     CryptDigestUpdateInt(&hashState, sizeof(TPM_EO), in->operation);
155     //  complete argument hash
156     CryptHashEnd2B(&hashState, &argHash.b);
157     // update policyDigest
158     //  start hash
159     CryptHashStart(&hashState, session->authHashAlg);
160     //  add old digest
161     CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);
162     //  add commandCode
163     CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), commandCode);
164     //  add argument digest
165     CryptDigestUpdate2B(&hashState, &argHash.b);
166     // complete the digest
167     CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);
168     return result;
169 }
170 #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

$$policyDigest_{new} := \mathbf{H}_{policyAlg}(policyDigest_{old} \parallel \text{TPM\_CC\_PolicyCommandCode} \parallel code) \qquad (26)$$

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.

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 133 — 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 134 — TPM2_PolicyCommandCode Response

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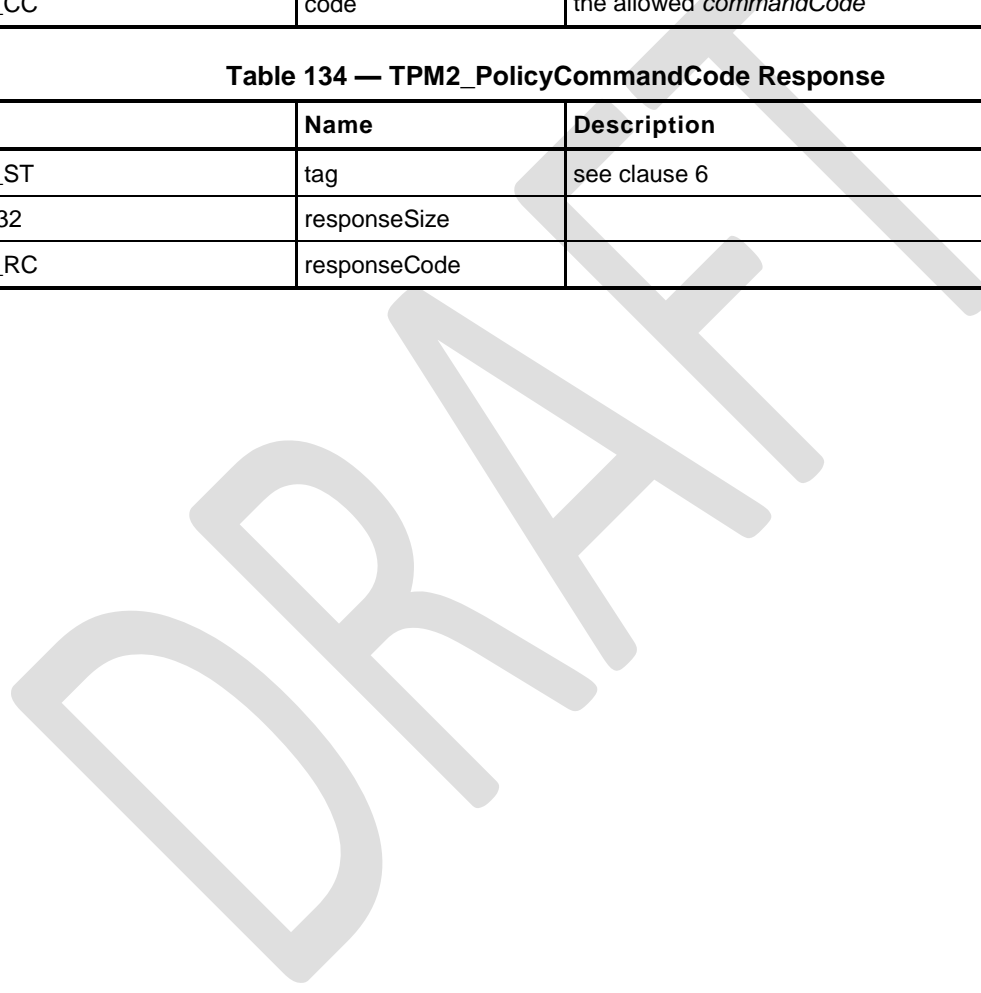
23.11.3 Detailed Actions

```c
1   #include "Tpm.h"
2   #include "PolicyCommandCode_fp.h"
3   #ifdef TPM_CC_PolicyCommandCode   // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_VALUE	*commandCode* of *policySession* previously set to a different value

```c
4   TPM_RC
5   TPM2_PolicyCommandCode(
6       PolicyCommandCode_In    *in             // IN: input parameter list
7       )
8   {
9       SESSION     *session;
10      TPM_CC      commandCode = TPM_CC_PolicyCommandCode;
11      HASH_STATE  hashState;
12  // Input validation
13      // Get pointer to the session structure
14      session = SessionGet(in->policySession);
15      if(session->commandCode != 0 && session->commandCode != in->code)
16          return TPM_RCS_VALUE + RC_PolicyCommandCode_code;
17      if(CommandCodeToCommandIndex(in->code) == UNIMPLEMENTED_COMMAND_INDEX)
18          return TPM_RCS_POLICY_CC + RC_PolicyCommandCode_code;
19  // Internal Data Update
20      // Update policy hash
21      // policyDigestnew = hash(policyDigestold || TPM_CC_PolicyCommandCode || code)
22      //  Start hash
23      CryptHashStart(&hashState, session->authHashAlg);
24      //  add old digest
25      CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);
26      //  add commandCode
27      CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), commandCode);
28      //  add input commandCode
29      CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), in->code);
30      //  complete the hash and get the results
31      CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);
32      // update commandCode value in session context
33      session->commandCode = in->code;
34      return TPM_RC_SUCCESS;
35  }
36  #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

$$policyDigest_{new} := \mathbf{H}_{policyAlg}(policyDigest_{old} \,||\, \text{TPM\_CC\_PolicyPhysicalPresence}) \qquad (27)$$

23.12.2 Command and Response

Table 135 — 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 136 — TPM2_PolicyPhysicalPresence Response

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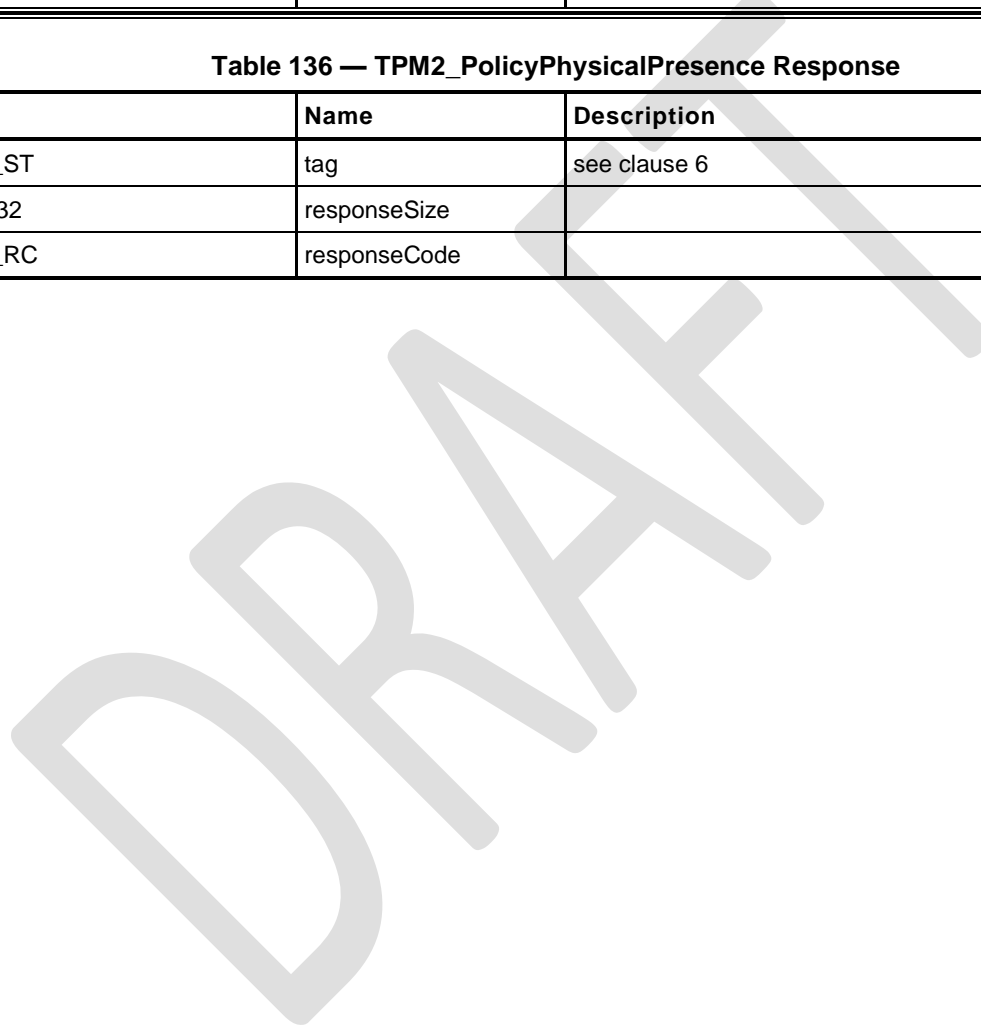
23.12.3 Detailed Actions

```
1   #include "Tpm.h"
2   #include "PolicyPhysicalPresence_fp.h"
3   #ifdef TPM_CC_PolicyPhysicalPresence   // Conditional expansion of this file
4   TPM_RC
5   TPM2_PolicyPhysicalPresence(
6       PolicyPhysicalPresence_In    *in              // IN: input parameter list
7       )
8   {
9       SESSION        *session;
10      TPM_CC         commandCode = TPM_CC_PolicyPhysicalPresence;
11      HASH_STATE   hashState;
12  // Internal Data Update
13      // Get pointer to the session structure
14      session = SessionGet(in->policySession);
15      // Update policy hash
16      // policyDigestnew = hash(policyDigestold || TPM_CC_PolicyPhysicalPresence)
17      //  Start hash
18      CryptHashStart(&hashState, session->authHashAlg);
19      //  add old digest
20      CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);
21      //  add commandCode
22      CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), commandCode);
23      //  complete the digest
24      CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);
25      // update session attribute
26      session->attributes.isPPRequired = SET;
27      return TPM_RC_SUCCESS;
28  }
29  #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_VALUE. 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

$$policyDigest_{new} := \mathbf{H}_{policyAlg}(policyDigest_{old} \,||\, \text{TPM\_CC\_PolicyCpHash} \,||\, cpHashA) \tag{28}$$

23.13.2 Command and Response

Table 137 — 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 138 — TPM2_PolicyCpHash Response

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

23.13.3 Detailed Actions

```
1   #include "Tpm.h"
2   #include "PolicyCpHash_fp.h"
3   #ifdef TPM_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*

```
4   TPM_RC
5   TPM2_PolicyCpHash(
6       PolicyCpHash_In     *in                 // IN: input parameter list
7       )
8   {
9       SESSION      *session;
10      TPM_CC       commandCode = TPM_CC_PolicyCpHash;
11      HASH_STATE   hashState;
12  // Input Validation
13      // Get pointer to the session structure
14      session = SessionGet(in->policySession);
15      // A new cpHash is given in input parameter, but cpHash in session context
16      // is not empty, or is not the same as the new cpHash
17      if(in->cpHashA.t.size != 0
18          && session->u1.cpHash.t.size != 0
19          && !MemoryEqual2B(&in->cpHashA.b, &session->u1.cpHash.b))
20          return TPM_RC_CPHASH;
21      // A valid cpHash must have the same size as session hash digest
22      if(in->cpHashA.t.size != CryptHashGetDigestSize(session->authHashAlg))
23          return TPM_RCS_SIZE + RC_PolicyCpHash_cpHashA;
24  // Internal Data Update
25      // Update policy hash
26      // policyDigestnew = hash(policyDigestold || TPM_CC_PolicyCpHash || cpHashA)
27      //  Start hash
28      CryptHashStart(&hashState, session->authHashAlg);
29      //  add old digest
30      CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);
31      //  add commandCode
32      CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), commandCode);
33      //  add cpHashA
34      CryptDigestUpdate2B(&hashState, &in->cpHashA.b);
35      //  complete the digest and get the results
36      CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);
37      // update cpHash in session context
38      session->u1.cpHash = in->cpHashA;
39      session->attributes.isCpHashDefined = SET;
40      return TPM_RC_SUCCESS;
41  }
42  #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:
$$nameHash := \mathbf{H}_{policyAlg}(objectHandle{\rightarrow}Name \ || \ newParentHandle{\rightarrow}Name)$$

If *policySession→cpHash* is already set, the TPM shall return TPM_RC_VALUE. 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

$$policyDigest_{new} := \mathbf{H}_{policyAlg}(policyDigest_{old} \ || \ \text{TPM\_CC\_PolicyNameHash} \ || \ nameHash) \qquad (29)$$

NOTE 2 This command will often be used with TPM2_PolicyAuthorize() where the owner of the object being duplicated provides approval for their object to be migrated to a specific new parent.

23.14.2 Command and Response

Table 139 — 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 140 — TPM2_PolicyNameHash Response

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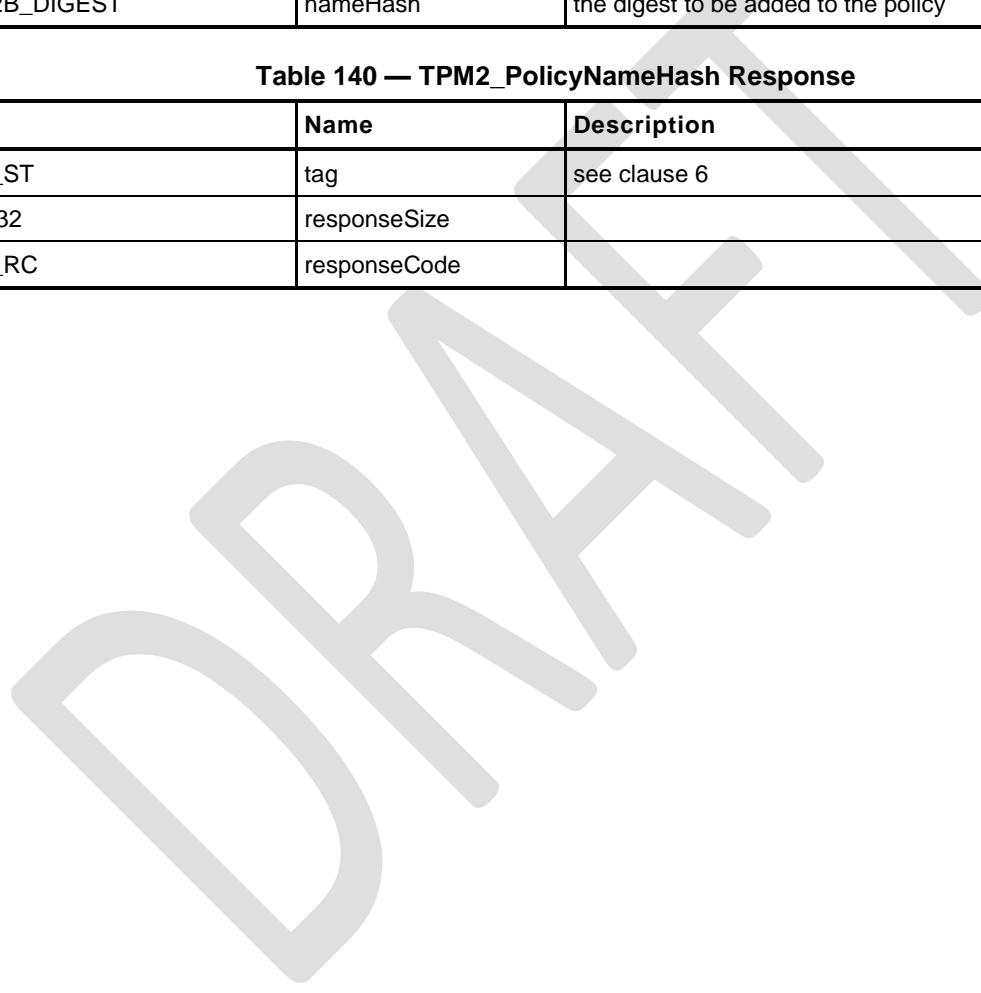
23.14.3 Detailed Actions

```
1   #include "Tpm.h"
2   #include "PolicyNameHash_fp.h"
3   #ifdef TPM_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*

```
4    TPM_RC
5    TPM2_PolicyNameHash(
6        PolicyNameHash_In   *in              // IN: input parameter list
7        )
8    {
9        SESSION                *session;
10       TPM_CC                  commandCode = TPM_CC_PolicyNameHash;
11       HASH_STATE              hashState;
12   // Input Validation
13       // Get pointer to the session structure
14       session = SessionGet(in->policySession);
15       // A new nameHash is given in input parameter, but cpHash in session context
16       // is not empty
17       if(in->nameHash.t.size != 0 && session->u1.cpHash.t.size != 0)
18           return TPM_RC_CPHASH;
19       // A valid nameHash must have the same size as session hash digest
20       if(in->nameHash.t.size != CryptHashGetDigestSize(session->authHashAlg))
21           return TPM_RCS_SIZE + RC_PolicyNameHash_nameHash;
22   // Internal Data Update
23       // Update policy hash
24       // policyDigestnew = hash(policyDigestold || TPM_CC_PolicyNameHash || nameHash)
25       //  Start hash
26       CryptHashStart(&hashState, session->authHashAlg);
27       //  add old digest
28       CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);
29       //  add commandCode
30       CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), commandCode);
31       //  add nameHash
32       CryptDigestUpdate2B(&hashState, &in->nameHash.b);
33       //  complete the digest
34       CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);
35       //  clear iscpHashDefined bit to indicate now this field contains a nameHash
36       session->attributes.isCpHashDefined = CLEAR;
37       // update nameHash in session context
38       session->u1.cpHash = in->nameHash;
39       return TPM_RC_SUCCESS;
40   }
41   #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:

$$nameHash := \mathbf{H}_{policyAlg}(objectName \,||\, newParentName) \tag{30}$$

NOTE 3   It is allowed that policySesion→nameHash and policySession→cpHash share the same memory space.

The *policySession→policyDigest* will be updated according to the setting of *includeObject*. If equal to YES, *policySession→policyDigest* is updated by:

$$policyDigest_{new} := \mathbf{H}_{policyAlg}(policyDigest_{old} \,||\, \text{TPM\_CC\_PolicyDuplicationSelect} \,||\, objectName \,||\, newParentName \,||\, includeObject) \tag{31}$$

If includeObject is NO, *policySession→policyDigest* is updated by:

$$policyDigest_{new} := \mathbf{H}_{policyAlg}(policyDigest_{old} \,||\, \text{TPM\_CC\_PolicyDuplicationSelect} \,||\, newParentName \,||\, includeObject) \tag{32}$$

NOTE 4   *policySession→cpHash* 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 5   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 141 — 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 142 — TPM2_PolicyDuplicationSelect Response

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

23.15.3 Detailed Actions

```
1    #include "Tpm.h"
2    #include "PolicyDuplicationSelect_fp.h"
3    #ifdef TPM_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

```
4    TPM_RC
5    TPM2_PolicyDuplicationSelect(
6        PolicyDuplicationSelect_In  *in              // IN: input parameter list
7        )
8    {
9        SESSION          *session;
10       HASH_STATE       hashState;
11       TPM_CC           commandCode = TPM_CC_PolicyDuplicationSelect;
12   // Input Validation
13       // Get pointer to the session structure
14       session = SessionGet(in->policySession);
15       // cpHash in session context must be empty
16       if(session->u1.cpHash.t.size != 0)
17           return TPM_RC_CPHASH;
18       // commandCode in session context must be empty
19       if(session->commandCode != 0)
20           return TPM_RC_COMMAND_CODE;
21   // Internal Data Update
22       // Update name hash
23       session->u1.cpHash.t.size = CryptHashStart(&hashState, session->authHashAlg);
24       //  add objectName
25       CryptDigestUpdate2B(&hashState, &in->objectName.b);
26       //  add new parent name
27       CryptDigestUpdate2B(&hashState, &in->newParentName.b);
28       //  complete hash
29       CryptHashEnd2B(&hashState, &session->u1.cpHash.b);
30       // update policy hash
31       // Old policyDigest size should be the same as the new policyDigest size since
32       // they are using the same hash algorithm
33       session->u2.policyDigest.t.size
34           = CryptHashStart(&hashState, session->authHashAlg);
35   //  add old policy
36       CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);
37       //  add command code
38       CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), commandCode);
39       //  add objectName
40       if(in->includeObject == YES)
41           CryptDigestUpdate2B(&hashState, &in->objectName.b);
42       //  add new parent name
43       CryptDigestUpdate2B(&hashState, &in->newParentName.b);
44       //  add includeObject
45       CryptDigestUpdateInt(&hashState, sizeof(TPMI_YES_NO), in->includeObject);
46       //  complete digest
47       CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);
48       // clear iscpHashDefined bit to indicate now this field contains a nameHash
49       session->attributes.isCpHashDefined = CLEAR;
50       // set commandCode in session context
51       session->commandCode = TPM_CC_Duplicate;
52       return TPM_RC_SUCCESS;
53   }
54   #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

$$aHash := \mathbf{H}_{aHashAlg}(approvedPolicy \,||\, policyRef) \tag{33}$$

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 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 143 — 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 144 — TPM2_PolicyAuthorize Response

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

23.16.3 Detailed Actions

```
1   #include "Tpm.h"
2   #include "PolicyAuthorize_fp.h"
3   #ifdef TPM_CC_PolicyAuthorize   // Conditional expansion of this file
4   #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

```
5   TPM_RC
6   TPM2_PolicyAuthorize(
7       PolicyAuthorize_In  *in              // IN: input parameter list
8       )
9   {
10      SESSION                 *session;
11      TPM2B_DIGEST            authHash;
12      HASH_STATE              hashState;
13      TPMT_TK_VERIFIED        ticket;
14      TPM_ALG_ID              hashAlg;
15      UINT16                  digestSize;
16  // Input Validation
17      // Get pointer to the session structure
18      session = SessionGet(in->policySession);
19      // Extract from the Name of the key, the algorithm used to compute it's Name
20      hashAlg = BYTE_ARRAY_TO_UINT16(in->keySign.t.name);
21      // 'keySign' parameter needs to use a supported hash algorithm, otherwise
22      // can't tell how large the digest should be
23      if(!CryptHashIsImplemented(hashAlg, 0))
24          return TPM_RCS_HASH + RC_PolicyAuthorize_keySign;
25      digestSize = CryptHashGetDigestSize(hashAlg);
26      if(digestSize != (in->keySign.t.size - 2))
27          return TPM_RCS_SIZE + RC_PolicyAuthorize_keySign;
28      //If this is a trial policy, skip all validations
29      if(session->attributes.isTrialPolicy == CLEAR)
30      {
31          // Check that "approvedPolicy" matches the current value of the
32          // policyDigest in policy session
33          if(!MemoryEqual2B(&session->u2.policyDigest.b,
34                          &in->approvedPolicy.b))
35              return TPM_RCS_VALUE + RC_PolicyAuthorize_approvedPolicy;
36          // Validate ticket TPMT_TK_VERIFIED
37          // Compute aHash.  The authorizing object sign a digest
38          // aHash := hash(approvedPolicy || policyRef).
39          // Start hash
40          authHash.t.size = CryptHashStart(&hashState, hashAlg);
41          // add approvedPolicy
42          CryptDigestUpdate2B(&hashState, &in->approvedPolicy.b);
43          // add policyRef
44          CryptDigestUpdate2B(&hashState, &in->policyRef.b);
45          // complete hash
46          CryptHashEnd2B(&hashState, &authHash.b);
47          // re-compute TPMT_TK_VERIFIED
48          TicketComputeVerified(in->checkTicket.hierarchy, &authHash,
49                              &in->keySign, &ticket);
50          // Compare ticket digest.  If not match, return error
51          if(!MemoryEqual2B(&in->checkTicket.digest.b, &ticket.digest.b))
52              return TPM_RCS_VALUE + RC_PolicyAuthorize_checkTicket;
53      }
```

```
54    // Internal Data Update
55        // Set policyDigest to zero digest
56        PolicyDigestClear(session);
57        // Update policyDigest
58        PolicyContextUpdate(TPM_CC_PolicyAuthorize, &in->keySign, &in->policyRef,
59                            NULL, 0, session);
60        return TPM_RC_SUCCESS;
61    }
62    #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

$$policyDigest_{new} := \mathbf{H}_{policyAlg}(policyDigest_{old} \,||\, \text{TPM\_CC\_PolicyAuthValue}) \tag{36}$$

23.17.2 Command and Response

<p align="center">**Table 145 — TPM2_PolicyAuthValue Command**</p>

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

<p align="center">**Table 146 — TPM2_PolicyAuthValue Response**</p>

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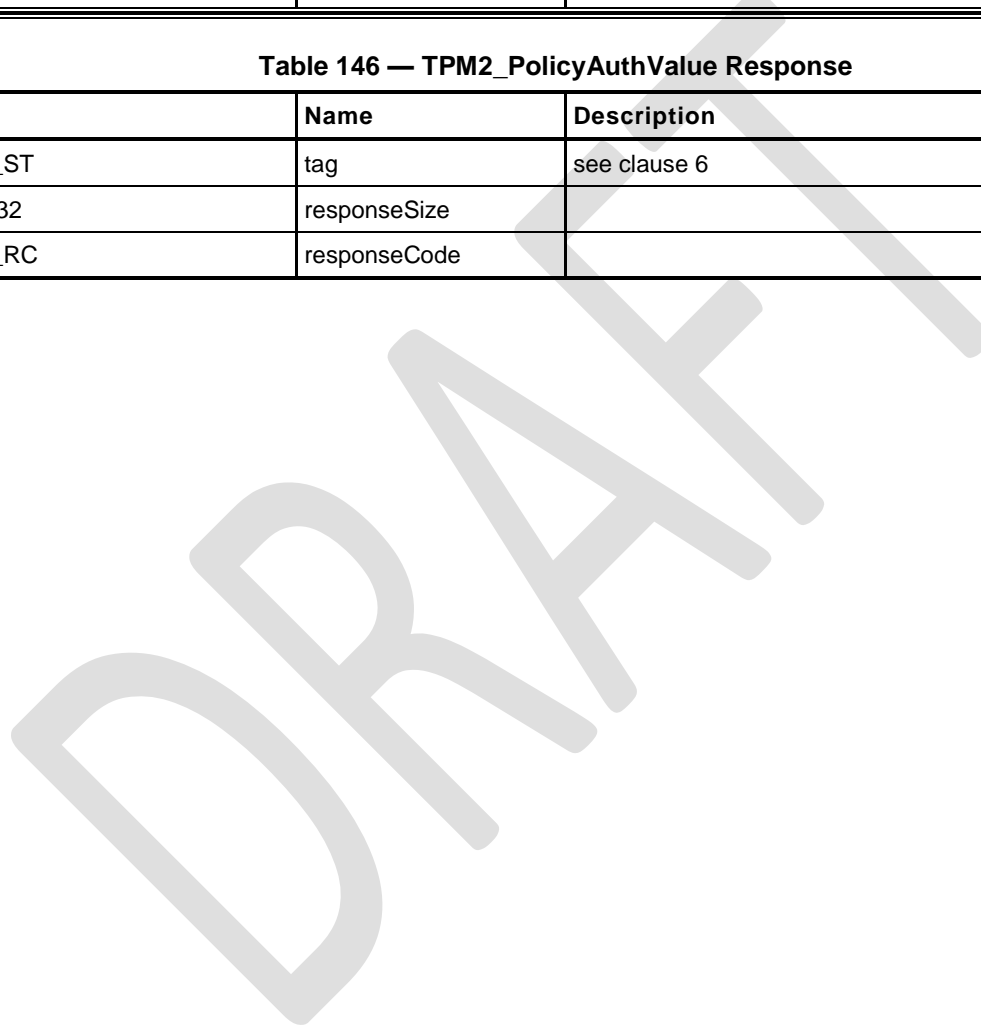
23.17.3 Detailed Actions

```
1   #include "Tpm.h"
2   #include "PolicyAuthValue_fp.h"
3   #ifdef TPM_CC_PolicyAuthValue  // Conditional expansion of this file
4   #include "Policy_spt_fp.h"
5   TPM_RC
6   TPM2_PolicyAuthValue(
7       PolicyAuthValue_In  *in            // IN: input parameter list
8       )
9   {
10      SESSION             *session;
11      TPM_CC               commandCode = TPM_CC_PolicyAuthValue;
12      HASH_STATE           hashState;
13  // Internal Data Update
14      // Get pointer to the session structure
15      session = SessionGet(in->policySession);
16      // Update policy hash
17      // policyDigestnew = hash(policyDigestold || TPM_CC_PolicyAuthValue)
18      //    Start hash
19      CryptHashStart(&hashState, session->authHashAlg);
20      //  add old digest
21      CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);
22      //  add commandCode
23      CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), commandCode);
24      //  complete the hash and get the results
25      CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);
26      // update isAuthValueNeeded bit in the session context
27      session->attributes.isAuthValueNeeded = SET;
28      session->attributes.isPasswordNeeded = CLEAR;
29      return TPM_RC_SUCCESS;
30  }
31  #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

$$policyDigest_{new} := \mathbf{H}_{policyAlg}(policyDigest_{old} \,||\, \text{TPM\_CC\_PolicyAuthValue}) \qquad (37)$$

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

<p align="center">**Table 147 — TPM2_PolicyPassword Command**</p>

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

<p align="center">**Table 148 — TPM2_PolicyPassword Response**</p>

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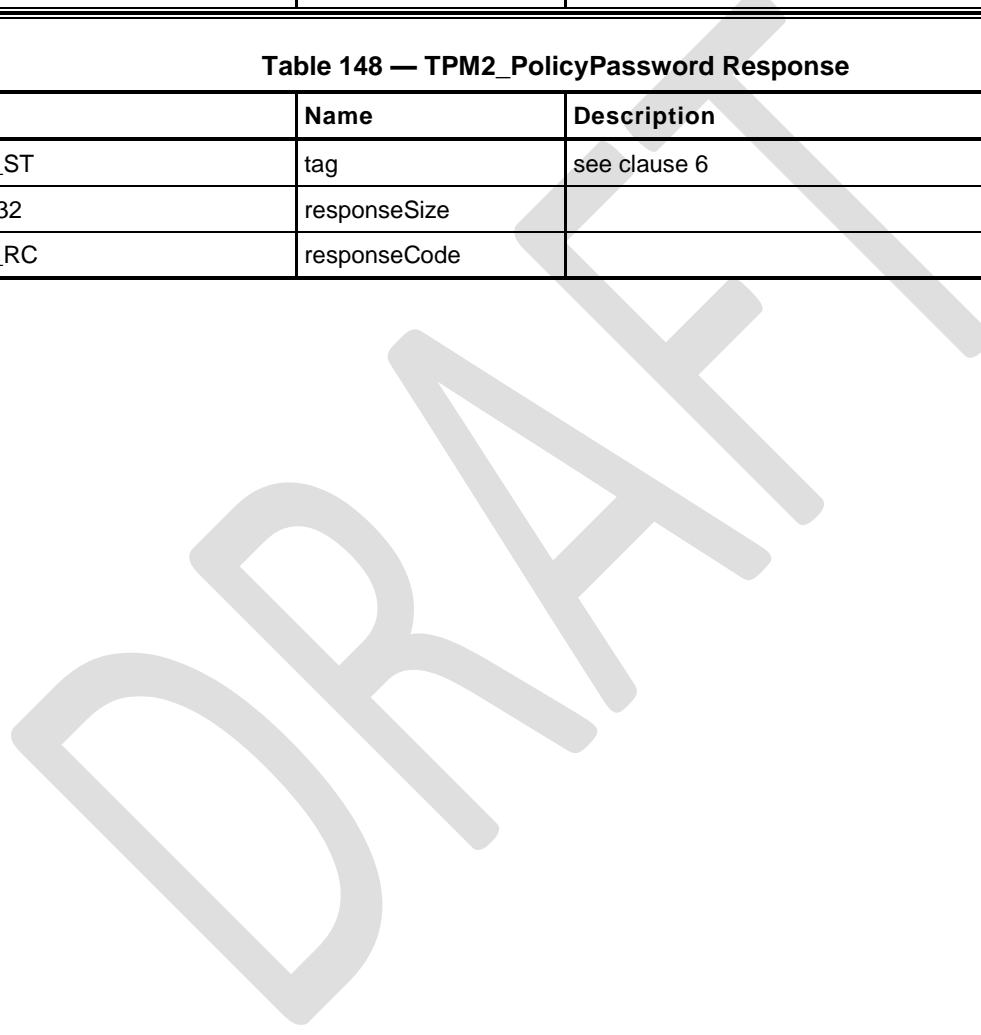
23.18.3 Detailed Actions

```
1   #include "Tpm.h"
2   #include "PolicyPassword_fp.h"
3   #ifdef TPM_CC_PolicyPassword  // Conditional expansion of this file
4   #include "Policy_spt_fp.h"
5   TPM_RC
6   TPM2_PolicyPassword(
7       PolicyPassword_In   *in              // IN: input parameter list
8       )
9   {
10      SESSION             *session;
11      TPM_CC               commandCode = TPM_CC_PolicyAuthValue;
12      HASH_STATE           hashState;
13  // Internal Data Update
14      // Get pointer to the session structure
15      session = SessionGet(in->policySession);
16      // Update policy hash
17      // policyDigestnew = hash(policyDigestold || TPM_CC_PolicyAuthValue)
18      //  Start hash
19      CryptHashStart(&hashState, session->authHashAlg);
20      //  add old digest
21      CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);
22      //  add commandCode
23      CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), commandCode);
24      //  complete the digest
25      CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);
26      //  Update isPasswordNeeded bit
27      session->attributes.isPasswordNeeded = SET;
28      session->attributes.isAuthValueNeeded = CLEAR;
29      return TPM_RC_SUCCESS;
30  }
31  #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 149 — 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 150 — 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*
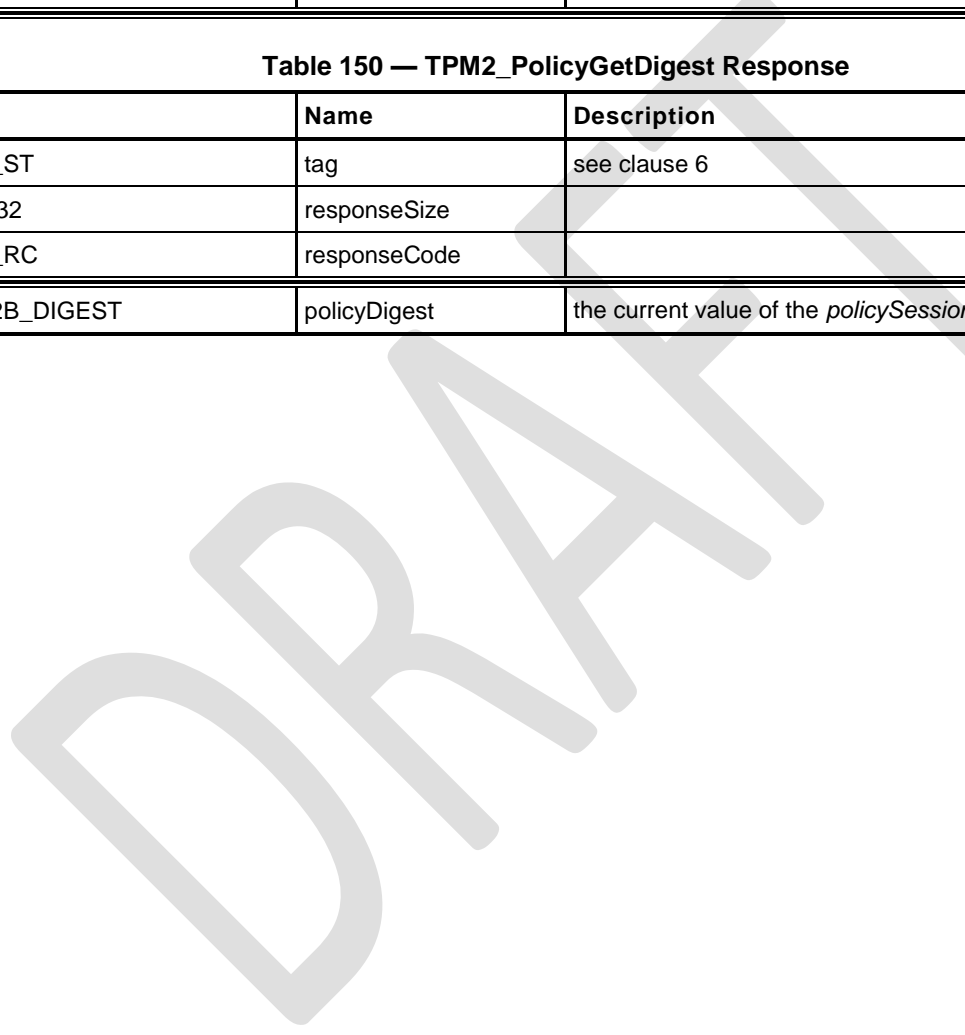
23.19.3 Detailed Actions

```
1    #include "Tpm.h"
2    #include "PolicyGetDigest_fp.h"
3    #ifdef TPM_CC_PolicyGetDigest  // Conditional expansion of this file
4    TPM_RC
5    TPM2_PolicyGetDigest(
6        PolicyGetDigest_In      *in,            // IN: input parameter list
7        PolicyGetDigest_Out     *out            // OUT: output parameter list
8        )
9    {
10       SESSION      *session;
11   // Command Output
12       // Get pointer to the session structure
13       session = SessionGet(in->policySession);
14       out->policyDigest = session->u2.policyDigest;
15       return TPM_RC_SUCCESS;
16   }
17   #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

$$policyDigest_{new} := \mathbf{H}_{policyAlg}(policyDigest_{old} \,||\, \text{TPM\_CC\_PolicyNvWritten} \,||\, writtenSet) \qquad (38)$$

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 151 — 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 152 — TPM2_PolicyNvWritten Response

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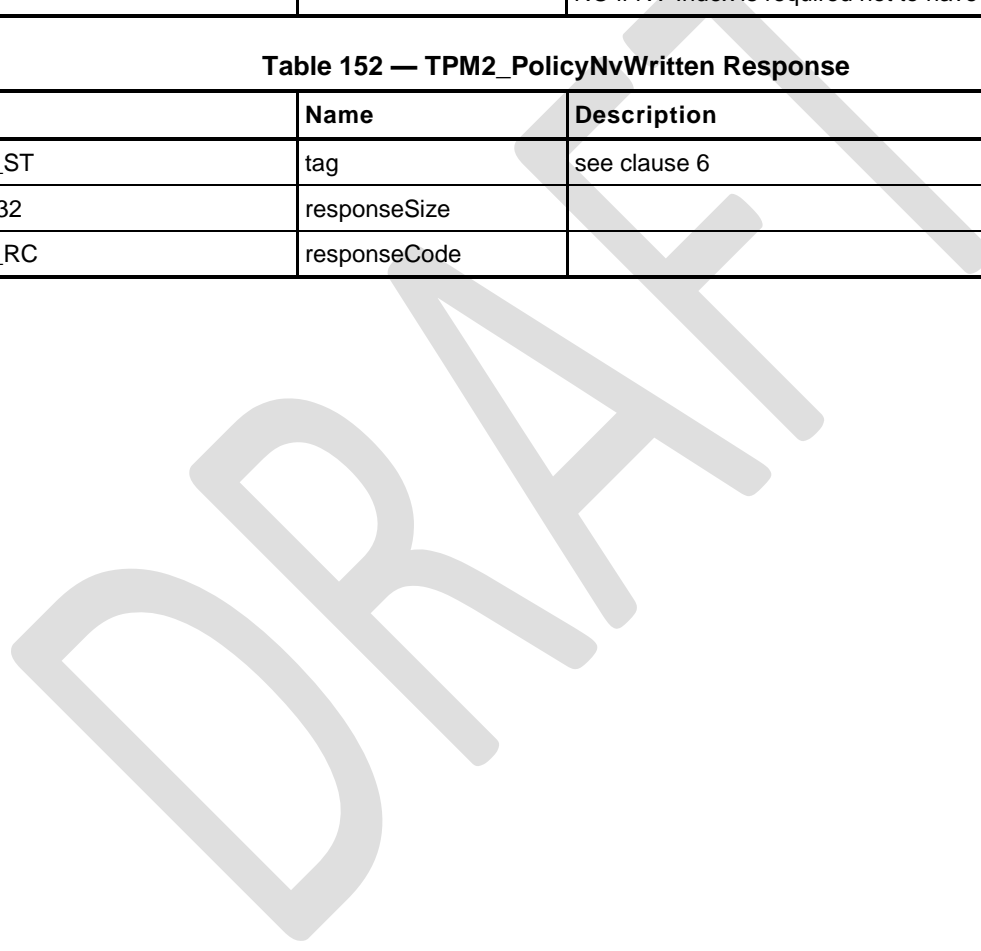
23.20.3 Detailed Actions

```
1   #include "Tpm.h"
2   #include "PolicyNvWritten_fp.h"
3   #ifdef TPM_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

```
4   TPM_RC
5   TPM2_PolicyNvWritten(
6       PolicyNvWritten_In  *in              // IN: input parameter list
7       )
8   {
9       SESSION      *session;
10      TPM_CC        commandCode = TPM_CC_PolicyNvWritten;
11      HASH_STATE    hashState;
12  // Input Validation
13      // Get pointer to the session structure
14      session = SessionGet(in->policySession);
15      // If already set is this a duplicate (the same setting)? If it
16      // is a conflicting setting, it is an error
17      if(session->attributes.checkNvWritten == SET)
18      {
19          if(((session->attributes.nvWrittenState == SET)
20              != (in->writtenSet == YES)))
21              return TPM_RCS_VALUE + RC_PolicyNvWritten_writtenSet;
22      }
23  // Internal Data Update
24      // Set session attributes so that the NV Index needs to be checked
25      session->attributes.checkNvWritten = SET;
26      session->attributes.nvWrittenState = (in->writtenSet == YES);
27      // Update policy hash
28      // policyDigestnew = hash(policyDigestold || TPM_CC_PolicyNvWritten
29      //                        || writtenSet)
30      // Start hash
31      CryptHashStart(&hashState, session->authHashAlg);
32      // add old digest
33      CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);
34      // add commandCode
35      CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), commandCode);
36      // add the byte of writtenState
37      CryptDigestUpdateInt(&hashState, sizeof(TPMI_YES_NO), in->writtenSet);
38      // complete the digest
39      CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);
40      return TPM_RC_SUCCESS;
41  }
42  #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→cpHash* is already set, the TPM shall 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*.

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   This allows the space normally used to hold *policySession→cpHash* to be used for *policySession→templateHash* instead.

The *policySession→policyDigest* will be updated with

$$policyDigest_{new} := \mathbf{H}_{policyAlg}(policyDigest_{old} \;||\; \text{TPM\_CC\_PolicyTemplate} \;||\; templateHash) \qquad (39)$$

23.21.2 Command and Response

Table 153 — 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 154 — TPM2_PolicyTemplate Response

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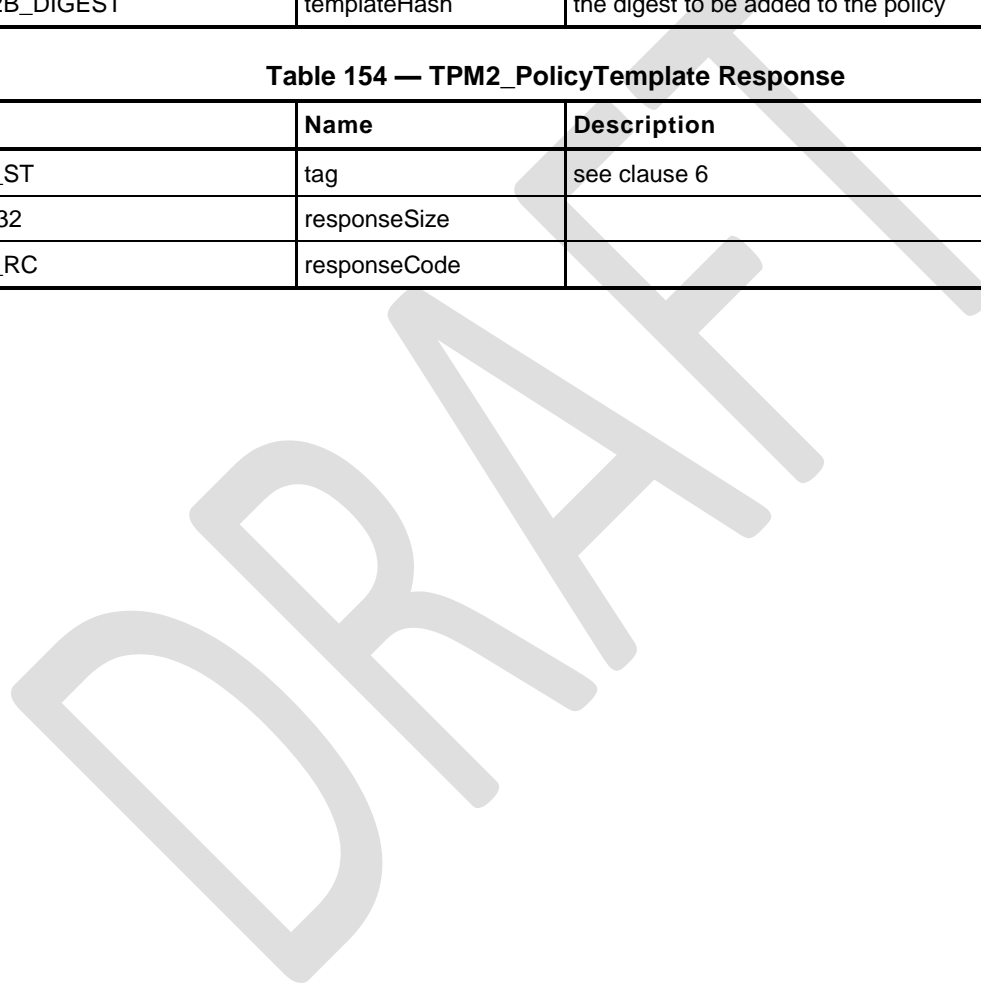
23.21.3 Detailed Actions

```
1    #include "Tpm.h"
2    #include "PolicyTemplate_fp.h"
3    #ifdef TPM_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*

```
4    TPM_RC
5    TPM2_PolicyTemplate(
6        PolicyTemplate_In     *in               // IN: input parameter list
7        )
8    {
9        SESSION      *session;
10       TPM_CC       commandCode = TPM_CC_PolicyTemplate;
11       HASH_STATE   hashState;
12   // Input Validation
13       // Get pointer to the session structure
14       session = SessionGet(in->policySession);
15       // The cpHash in session context
16       // is not empty, or is not the same as 'templateHash'
17       if(session->u1.cpHash.t.size != 0
18          && !MemoryEqual2B(&in->templateHash.b, &session->u1.cpHash.b))
19           return TPM_RC_CPHASH + RC_PolicyTemplate_templateHash;
20       // Can't have cpHash defined (competing uses)
21       if(session->attributes.isCpHashDefined)
22           return TPM_RC_CPHASH + RC_PolicyTemplate_templateHash;
23       // A valid templateHash must have the same size as session hash digest
24       if(in->templateHash.t.size != CryptHashGetDigestSize(session->authHashAlg))
25           return TPM_RCS_SIZE + RC_PolicyTemplate_templateHash;
26   // Internal Data Update
27       // Update policy hash
28       // policyDigestnew = hash(policyDigestold || TPM_CC_PolicyCpHash
29       //   || cpHashA.buffer)
30       //   Start hash
31       CryptHashStart(&hashState, session->authHashAlg);
32       //   add old digest
33       CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);
34       //   add commandCode
35       CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), commandCode);
36       //   add cpHashA
37       CryptDigestUpdate2B(&hashState, &in->templateHash.b);
38       //   complete the digest and get the results
39       CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);
40       // update cpHash in session context
41       session->u1.cpHash = in->templateHash;
42       session->attributes.isTemplateSet = SET;
43       return TPM_RC_SUCCESS;
44   }
45   #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

$$policyDigest_{new} := \mathbf{H}_{policyAlg}(policyDigest_{old} \parallel \text{TPM\_CC\_PolicyAuthorizeNV} \parallel nvIndex{\rightarrow}Name) \quad (40)$$

23.22.2 Command and Response

Table 155 — 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 156 — TPM2_PolicyAuthorizeNV Response

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

23.22.3 Detailed Actions

```
1   #include "Tpm.h"
2   //#include "PolicyAuthorizeNV_fp.h"
3   #ifdef TPM_CC_PolicyAuthorizeNV  // Conditional expansion of this file
4   #include "PolicyAuthorizeNV_fp.h"
5   #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

```
6    TPM_RC
7    TPM2_PolicyAuthorizeNV(
8        PolicyAuthorizeNV_In    *in
9        )
10   {
11       SESSION                 *session;
12       TPM_RC                   result;
13       NV_REF                   locator;
14       NV_INDEX                *nvIndex = NvGetIndexInfo(in->nvIndex, &locator);
15       TPM2B_NAME               name;
16       TPMT_HA                  policyInNv;
17       BYTE                     nvTemp[sizeof(TPMT_HA)];
18       BYTE                    *buffer = nvTemp;
19       INT32                    size;
20   // Input Validation
21       // Get pointer to the session structure
22       session = SessionGet(in->policySession);
23       // Skip checks if this is a trial policy
24       if(!session->attributes.isTrialPolicy)
25       {
26           // Check the authorizations for reading
27           // Common read access checks. NvReadAccessChecks() returns
28           // TPM_RC_NV_AUTHORIZATION, TPM_RC_NV_LOCKED, or TPM_RC_NV_UNINITIALIZED
29           // error may be returned at this point
30           result = NvReadAccessChecks(in->authHandle, in->nvIndex,
31                                       nvIndex->publicArea.attributes);
32           if(result != TPM_RC_SUCCESS)
33               return result;
34           // Read the contents of the index into a temp buffer
35           size = MIN(nvIndex->publicArea.dataSize, sizeof(TPMT_HA));
36           // Perform the read
37           NvGetIndexData(nvIndex, locator, 0, (UINT16)size, nvTemp);
38           result = TPMT_HA_Unmarshal(&policyInNv, &buffer, &size, FALSE);
39           if(result != TPM_RC_SUCCESS)
40               return TPM_RC_POLICY_FAIL;
41           // See if the contents of the first N bytes of the index match the value
42           // in the policy
43           if(policyInNv.hashAlg != session->authHashAlg
44              || !MemoryEqual(&policyInNv.digest, &session->u2.policyDigest.t.buffer,
45                          session->u2.policyDigest.t.size))
46               return TPM_RC_POLICY_FAIL;
47       }
48   // Internal Data Update
49       // Set policyDigest to zero digest
50       PolicyDigestClear(session);
51       // Update policyDigest
52       PolicyContextUpdate(TPM_CC_PolicyAuthorizeNV, EntityGetName(in->nvIndex, &name),
53                       NULL, NULL, 0, session);
```

```
54          return TPM_RC_SUCCESS;
55      }
56  #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 157 — 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_TEMPLATE	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 158 — 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

```
1    #include "Tpm.h"
2    #include "CreatePrimary_fp.h"
3    #ifdef TPM_CC_CreatePrimary   // 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; 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	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

```
4    TPM_RC
5    TPM2_CreatePrimary(
6        CreatePrimary_In    *in,              // IN: input parameter list
7        CreatePrimary_Out   *out              // OUT: output parameter list
8        )
9    {
10   // Local variables
11       TPM_RC              result = TPM_RC_SUCCESS;
12       TPMT_PUBLIC         *publicArea;
13       DRBG_STATE          rand;
14       OBJECT              *newObject;
15   // Input Validation
16       // Will need a place to put the result
17       newObject = FindEmptyObjectSlot(&out->objectHandle);
18       if(newObject == NULL)
19           return TPM_RC_OBJECT_MEMORY;
20       // Get the address of the public area in the new object
21       // (this is just to save typing)
22       publicArea = &newObject->publicArea;
23       // Unmarshal the public area into the allocated object
24       result = UnmarshalToPublic(publicArea, &in->inPublic, FALSE);
25       if(result != TPM_RC_SUCCESS)
26           return result + RC_CreatePrimary_inPublic;
27       // Check attributes in input public area. CreateChecks() checks the things that
28       // are unique to creation and then validates the attributes and values that are
29       // common to create and load.
30       result = CreateChecks(NULL, publicArea);
31       if(result != TPM_RC_SUCCESS)
32           return RcSafeAddToResult(result, RC_CreatePrimary_inPublic);
33       // Validate the sensitive area values
```

```
34      if(!AdjustAuthSize(&in->inSensitive.sensitive.userAuth,
35                     publicArea->nameAlg))
36          return TPM_RCS_SIZE + RC_CreatePrimary_inSensitive;
37  // Command output
38      // Compute the name using out->name as a scratch area (this is not the value
39      // that ultimately will be returned, then instantiate the state that will be
40      // used as a random number generator during the object creation.
41      DRBG_InstantiateSeeded(&rand, &HierarchyGetPrimarySeed(in->primaryHandle)->b,
42                         PRIMARY_OBJECT_CREATION,
43                         &ObjectComputeName(&in->inPublic.b,
44                                            publicArea->nameAlg,
45                                            &out->name)->b,
46                         &in->inSensitive.sensitive.data.b);
47      newObject->attributes.primary = SET;
48      if(in->primaryHandle == TPM_RH_ENDORSEMENT)
49          newObject->attributes.epsHierarchy = SET;
50      // Create the primary object.
51      result = CryptCreateObject(newObject, &in->inSensitive.sensitive,
52                         (RAND_STATE *)&rand);
53      if(result != TPM_RC_SUCCESS)
54          return result;
55      // Set the publicArea and name from the computed values
56      out->outPublic.publicArea = newObject->publicArea;
57      out->name = newObject->name;
58      // Fill in creation data
59      FillInCreationData(in->primaryHandle, publicArea->nameAlg,
60                         &in->creationPCR, &in->outsideInfo, &out->creationData,
61                         &out->creationHash);
62      // Compute creation ticket
63      TicketComputeCreation(EntityGetHierarchy(in->primaryHandle), &out->name,
64                         &out->creationHash, &out->creationTicket);
65      // Set the remaining attributes for a loaded object
66      ObjectSetLoadedAttributes(newObject, in->primaryHandle);
67      return result;
68  }
69  #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 159 — 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 160 — TPM2_HierarchyControl Response

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

24.2.3 Detailed Actions

```
1    #include "Tpm.h"
2    #include "HierarchyControl_fp.h"
3    #ifdef TPM_CC_HierarchyControl  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_AUTH_TYPE	*authHandle* is not applicable to *hierarchy* in its current state

```
4    TPM_RC
5    TPM2_HierarchyControl(
6        HierarchyControl_In    *in              // IN: input parameter list
7        )
8    {
9        BOOL        select = (in->state == YES);
10       BOOL        *selected = NULL;
11   // Input Validation
12       switch(in->enable)
13       {
14           // Platform hierarchy has to be disabled by PlatformAuth
15           // If the platform hierarchy has already been disabled, only a reboot
16           // can enable it again
17           case TPM_RH_PLATFORM:
18           case TPM_RH_PLATFORM_NV:
19               if(in->authHandle != TPM_RH_PLATFORM)
20                   return TPM_RC_AUTH_TYPE;
21               break;
22           // ShEnable may be disabled if PlatformAuth/PlatformPolicy or
23           // OwnerAuth/OwnerPolicy is provided.  If ShEnable is disabled, then it
24           // may only be enabled if PlatformAuth/PlatformPolicy is provided.
25           case TPM_RH_OWNER:
26               if(in->authHandle != TPM_RH_PLATFORM
27                  && in->authHandle != TPM_RH_OWNER)
28                   return TPM_RC_AUTH_TYPE;
29               if(gc.shEnable == FALSE && in->state == YES
30                  && in->authHandle != TPM_RH_PLATFORM)
31                   return TPM_RC_AUTH_TYPE;
32               break;
33           // EhEnable may be disabled if either PlatformAuth/PlatformPolicy or
34           // EndosementAuth/EndorsementPolicy is provided.  If EhEnable is disabled,
35           // then it may only be enabled if PlatformAuth/PlatformPolicy is
36           // provided.
37           case TPM_RH_ENDORSEMENT:
38               if(in->authHandle != TPM_RH_PLATFORM
39                  && in->authHandle != TPM_RH_ENDORSEMENT)
40                   return TPM_RC_AUTH_TYPE;
41               if(gc.ehEnable == FALSE && in->state == YES
42                  && in->authHandle != TPM_RH_PLATFORM)
43                   return TPM_RC_AUTH_TYPE;
44               break;
45           default:
46               FAIL(FATAL_ERROR_INTERNAL);
47               break;
48       }
49   // Internal Data Update
50       // Enable or disable the selected hierarchy
51       // Note: the authorization processing for this command may keep these
52       // command actions from being executed. For example, if phEnable is
53       // CLEAR, then platformAuth cannot be used for authorization. This
54       // means that would not be possible to use platformAuth to change the
55       // state of phEnable from CLEAR to SET.
56       // If it is decided that platformPolicy can still be used when phEnable
57       // is CLEAR, then this code could SET phEnable when proper platform
```

```
58        // policy is provided.
59        switch(in->enable)
60        {
61            case TPM_RH_OWNER:
62                selected = &gc.shEnable;
63                break;
64            case TPM_RH_ENDORSEMENT:
65                selected = &gc.ehEnable;
66                break;
67            case TPM_RH_PLATFORM:
68                selected = &g_phEnable;
69                break;
70            case TPM_RH_PLATFORM_NV:
71                selected = &gc.phEnableNV;
72                break;
73            default:
74                FAIL(FATAL_ERROR_INTERNAL);
75                break;
76        }
77        if(selected != NULL && *selected != select)
78        {
79            // Before changing the internal state, make sure that NV is available.
80            // Only need to update NV if changing the orderly state
81            RETURN_IF_ORDERLY;
82            // state is changing and NV is available so modify
83            *selected = select;
84            // If a hierarchy was just disabled, flush it
85            if(select == CLEAR && in->enable != TPM_RH_PLATFORM_NV)
86            // Flush hierarchy
87                ObjectFlushHierarchy(in->enable);
88            // orderly state should be cleared because of the update to state clear data
89            // This gets processed in ExecuteCommand() on the way out.
90            g_clearOrderly = TRUE;
91        }
92        return TPM_RC_SUCCESS;
93    }
94    #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*).

The command requires an authorization session. The session shall use the current *authValue* or satisfy the current *authPolicy* for the referenced hierarchy.

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 161 — 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_AUTH	@authHandle	TPM_RH_LOCKOUT, TPM_RH_ENDORSEMENT, TPM_RH_OWNER 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 162 — TPM2_SetPrimaryPolicy Response

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

24.3.3   Detailed Actions

```c
1   #include "Tpm.h"
2   #include "SetPrimaryPolicy_fp.h"
3   #ifdef TPM_CC_SetPrimaryPolicy  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_SIZE	size of input *authPolicy* is not consistent with input hash algorithm

```c
4    TPM_RC
5    TPM2_SetPrimaryPolicy(
6        SetPrimaryPolicy_In      *in             // IN: input parameter list
7        )
8    {
9    // Input Validation
10       // Check the authPolicy consistent with hash algorithm. If the policy size is
11       // zero, then the algorithm is required to be TPM_ALG_NULL
12       if(in->authPolicy.t.size != CryptHashGetDigestSize(in->hashAlg))
13           return TPM_RCS_SIZE + RC_SetPrimaryPolicy_authPolicy;
14       // The command need NV update for OWNER and ENDORSEMENT hierarchy, and
15       // might need orderlyState update for PLATFROM hierarchy.
16       // Check if NV is available.  A TPM_RC_NV_UNAVAILABLE or TPM_RC_NV_RATE
17       // error may be returned at this point
18       RETURN_IF_NV_IS_NOT_AVAILABLE;
19   // Internal Data Update
20       // Set hierarchy policy
21       switch(in->authHandle)
22       {
23           case TPM_RH_OWNER:
24               gp.ownerAlg = in->hashAlg;
25               gp.ownerPolicy = in->authPolicy;
26               NV_SYNC_PERSISTENT(ownerAlg);
27               NV_SYNC_PERSISTENT(ownerPolicy);
28               break;
29           case TPM_RH_ENDORSEMENT:
30               gp.endorsementAlg = in->hashAlg;
31               gp.endorsementPolicy = in->authPolicy;
32               NV_SYNC_PERSISTENT(endorsementAlg);
33               NV_SYNC_PERSISTENT(endorsementPolicy);
34               break;
35           case TPM_RH_PLATFORM:
36               gc.platformAlg = in->hashAlg;
37               gc.platformPolicy = in->authPolicy;
38               // need to update orderly state
39               g_clearOrderly = TRUE;
40               break;
41           case TPM_RH_LOCKOUT:
42               gp.lockoutAlg = in->hashAlg;
43               gp.lockoutPolicy = in->authPolicy;
44               NV_SYNC_PERSISTENT(lockoutAlg);
45               NV_SYNC_PERSISTENT(lockoutPolicy);
46               break;
47           default:
48               FAIL(FATAL_ERROR_INTERNAL);
49               break;
50       }
51       return TPM_RC_SUCCESS;
52   }
53   #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 163 — 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 164 — TPM2_ChangePPS Response

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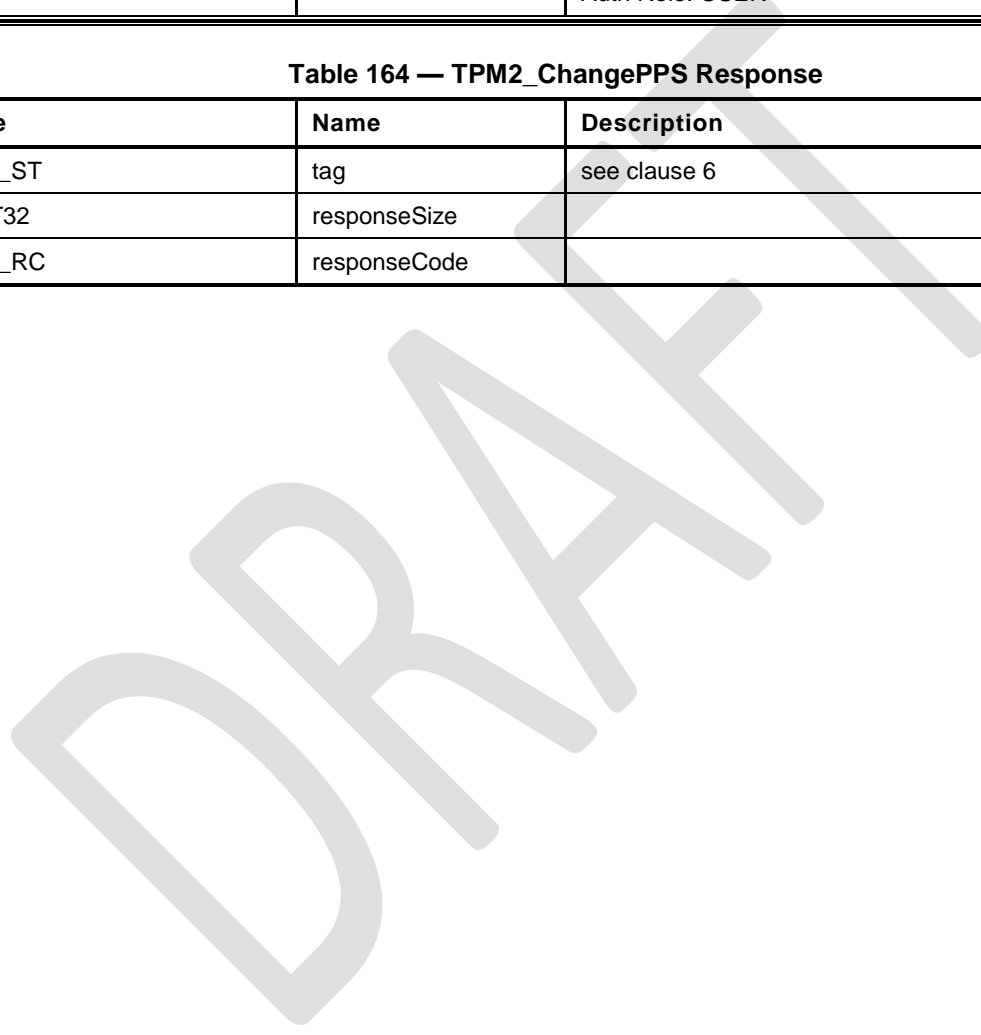
24.4.3 Detailed Actions

```
1   #include "Tpm.h"
2   #include "ChangePPS_fp.h"
3   #ifdef TPM_CC_ChangePPS  // Conditional expansion of this file
4   TPM_RC
5   TPM2_ChangePPS(
6       ChangePPS_In    *in              // IN: input parameter list
7       )
8   {
9       UINT32          i;
10      // Check if NV is available.  A TPM_RC_NV_UNAVAILABLE or TPM_RC_NV_RATE
11      // error may be returned at this point
12      RETURN_IF_NV_IS_NOT_AVAILABLE;
13      // Input parameter is not reference in command action
14      in = NULL;
15  // Internal Data Update
16      // Reset platform hierarchy seed from RNG
17      CryptRandomGenerate(PRIMARY_SEED_SIZE, gp.PPSeed.t.buffer);
18      // Create a new phProof value from RNG to prevent the saved platform
19      // hierarchy contexts being loaded
20      CryptRandomGenerate(PROOF_SIZE, gp.phProof.t.buffer);
21      // Set platform authPolicy to null
22      gc.platformAlg = TPM_ALG_NULL;
23      gc.platformPolicy.t.size = 0;
24      // Flush loaded object in platform hierarchy
25      ObjectFlushHierarchy(TPM_RH_PLATFORM);
26      // Flush platform evict object and index in NV
27      NvFlushHierarchy(TPM_RH_PLATFORM);
28      // Save hierarchy changes to NV
29      NV_SYNC_PERSISTENT(PPSeed);
30      NV_SYNC_PERSISTENT(phProof);
31      // Re-initialize PCR policies
32  #if defined NUM_POLICY_PCR_GROUP && NUM_POLICY_PCR_GROUP > 0
33      for(i = 0; i < NUM_POLICY_PCR_GROUP; i++)
34      {
35          gp.pcrPolicies.hashAlg[i] = TPM_ALG_NULL;
36          gp.pcrPolicies.policy[i].t.size = 0;
37      }
38      NV_SYNC_PERSISTENT(pcrPolicies);
39  #endif
40      // orderly state should be cleared because of the update to state clear data
41      g_clearOrderly = TRUE;
42      return TPM_RC_SUCCESS;
43  }
44  #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 165 — 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 166 — TPM2_ChangeEPS Response

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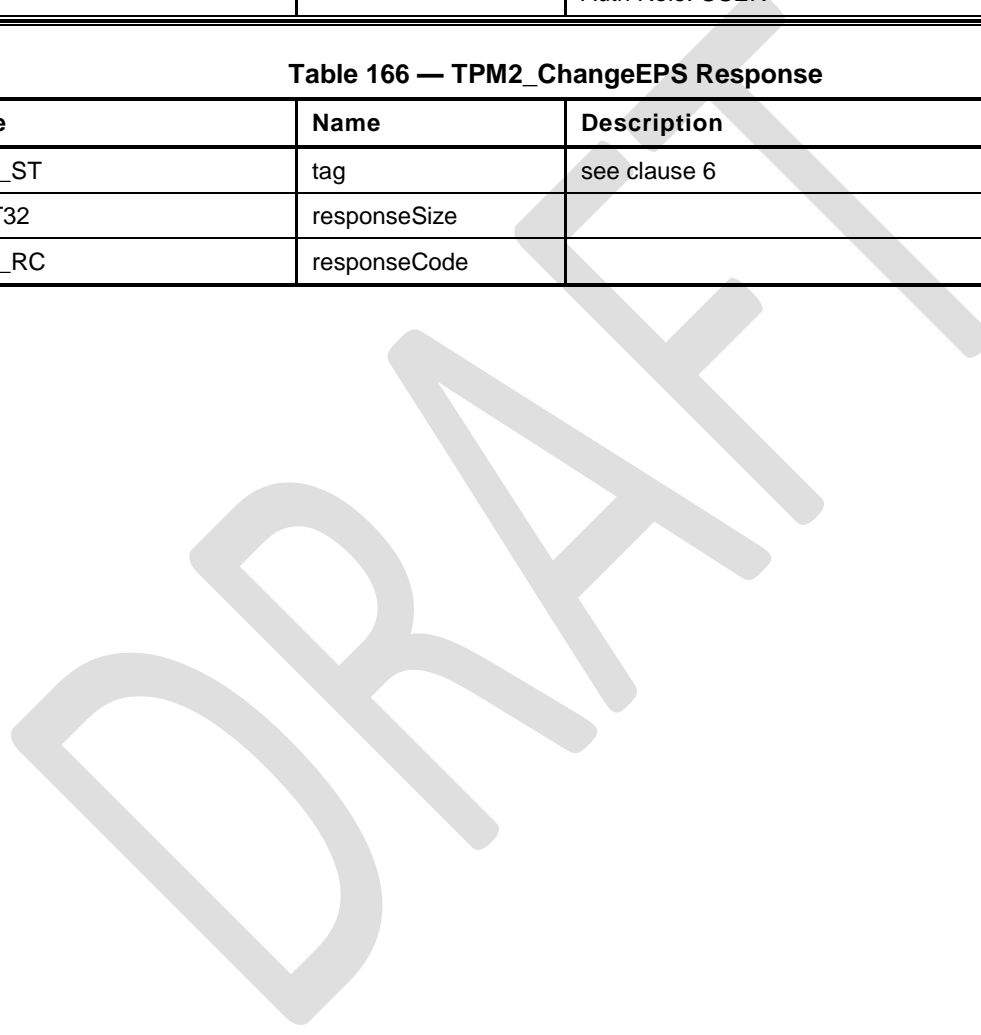
24.5.3   Detailed Actions

```
1    #include "Tpm.h"
2    #include "ChangeEPS_fp.h"
3    #ifdef TPM_CC_ChangeEPS  // Conditional expansion of this file
4    TPM_RC
5    TPM2_ChangeEPS(
6        ChangeEPS_In     *in             // IN: input parameter list
7        )
8    {
9        // The command needs NV update.  Check if NV is available.
10       // A TPM_RC_NV_UNAVAILABLE or TPM_RC_NV_RATE error may be returned at
11       // this point
12       RETURN_IF_NV_IS_NOT_AVAILABLE;
13       // Input parameter is not reference in command action
14       NOT_REFERENCED(in);
15   // Internal Data Update
16       // Reset endorsement hierarchy seed from RNG
17       CryptRandomGenerate(PRIMARY_SEED_SIZE, gp.EPSeed.t.buffer);
18       // Create new ehProof value from RNG
19       CryptRandomGenerate(PROOF_SIZE, gp.ehProof.t.buffer);
20       // Enable endorsement hierarchy
21       gc.ehEnable = TRUE;
22       // set authValue buffer to zeros
23       MemorySet(gp.endorsementAuth.t.buffer, 0, gp.endorsementAuth.t.size);
24       // Set endorsement authValue to null
25       gp.endorsementAuth.t.size = 0;
26       // Set endorsement authPolicy to null
27       gp.endorsementAlg = TPM_ALG_NULL;
28       gp.endorsementPolicy.t.size = 0;
29       // Flush loaded object in endorsement hierarchy
30       ObjectFlushHierarchy(TPM_RH_ENDORSEMENT);
31       // Flush evict object of endorsement hierarchy stored in NV
32       NvFlushHierarchy(TPM_RH_ENDORSEMENT);
33       // Save hierarchy changes to NV
34       NV_SYNC_PERSISTENT(EPSeed);
35       NV_SYNC_PERSISTENT(ehProof);
36       NV_SYNC_PERSISTENT(endorsementAuth);
37       NV_SYNC_PERSISTENT(endorsementAlg);
38       NV_SYNC_PERSISTENT(endorsementPolicy);
39       // orderly state should be cleared because of the update to state clear data
40       g_clearOrderly = TRUE;
41       return TPM_RC_SUCCESS;
42   }
43   #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 167 — 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 168 — TPM2_Clear Response

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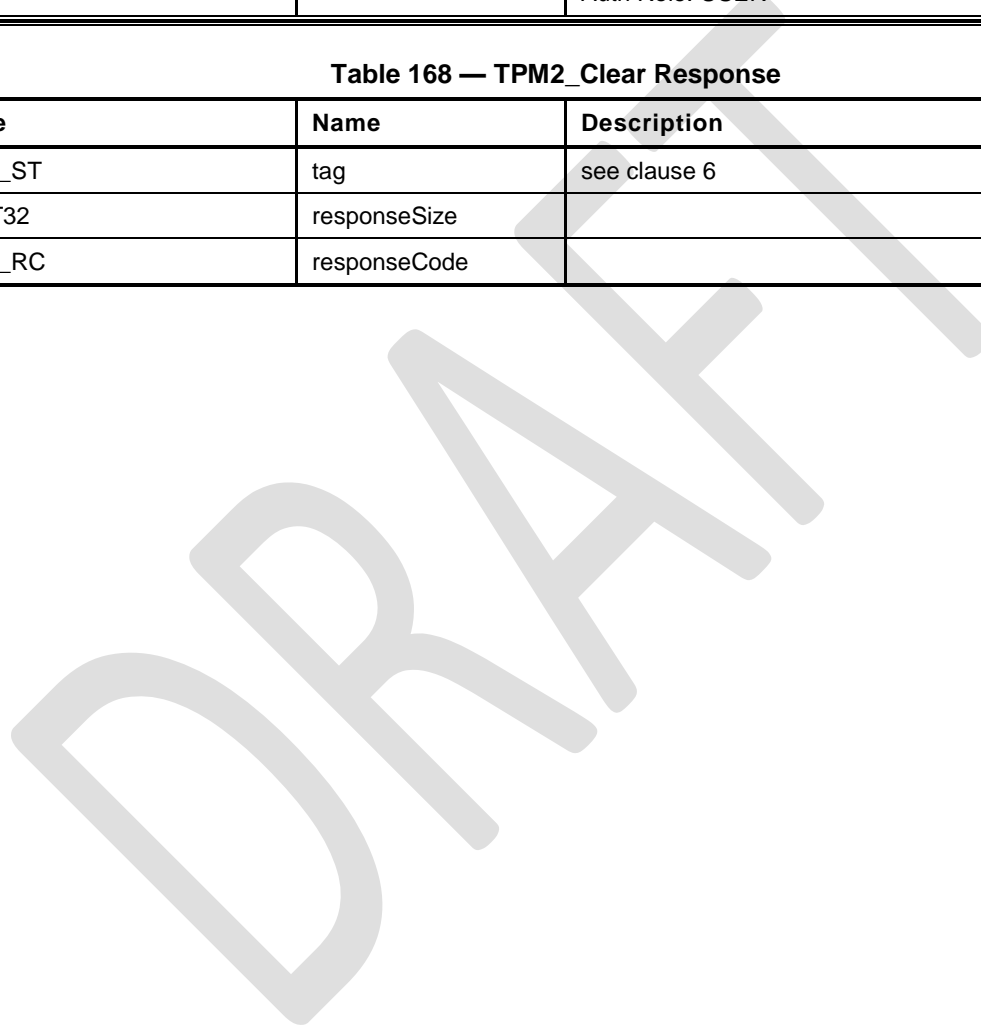
24.6.3   Detailed Actions

```
1    #include "Tpm.h"
2    #include "Clear_fp.h"
3    #ifdef TPM_CC_Clear   // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_DISABLED	Clear command has been disabled

```
4    TPM_RC
5    TPM2_Clear(
6        Clear_In         *in                // IN: input parameter list
7        )
8    {
9        // Input parameter is not reference in command action
10       NOT_REFERENCED(in);
11       // The command needs NV update.  Check if NV is available.
12       // A TPM_RC_NV_UNAVAILABLE or TPM_RC_NV_RATE error may be returned at
13       // this point
14       RETURN_IF_NV_IS_NOT_AVAILABLE;
15   // Input Validation
16       // If Clear command is disabled, return an error
17       if(gp.disableClear)
18           return TPM_RC_DISABLED;
19   // Internal Data Update
20       // Reset storage hierarchy seed from RNG
21       CryptRandomGenerate(PRIMARY_SEED_SIZE, gp.SPSeed.t.buffer);
22       // Create new shProof and ehProof value from RNG
23       CryptRandomGenerate(PROOF_SIZE, gp.shProof.t.buffer);
24       CryptRandomGenerate(PROOF_SIZE, gp.ehProof.t.buffer);
25       // Enable storage and endorsement hierarchy
26       gc.shEnable = gc.ehEnable = TRUE;
27       // set the authValue buffers to zero
28       MemorySet(&gp.ownerAuth, 0, sizeof(gp.ownerAuth));
29       MemorySet(&gp.endorsementAuth, 0, sizeof(gp.endorsementAuth));
30       MemorySet(&gp.lockoutAuth, 0, sizeof(gp.lockoutAuth));
31       // Set storage, endorsement, and lockout authPolicy to null
32       gp.ownerAlg = gp.endorsementAlg = gp.lockoutAlg = TPM_ALG_NULL;
33       MemorySet(&gp.ownerPolicy, 0, sizeof(gp.ownerPolicy));
34       MemorySet(&gp.lockoutPolicy, 0, sizeof(gp.lockoutPolicy));
35       // Flush loaded object in storage and endorsement hierarchy
36       ObjectFlushHierarchy(TPM_RH_OWNER);
37       ObjectFlushHierarchy(TPM_RH_ENDORSEMENT);
38       // Flush owner and endorsement object and owner index in NV
39       NvFlushHierarchy(TPM_RH_OWNER);
40       NvFlushHierarchy(TPM_RH_ENDORSEMENT);
41       // Initialize dictionary attack parameters
42       DAPreInstall_Init();
43       // Reset clock
44       go.clock = 0;
45       go.clockSafe = YES;
46       NvWrite(NV_ORDERLY_DATA, sizeof(ORDERLY_DATA), &go);
47       // Reset counters
48       gp.resetCount = gr.restartCount = gr.clearCount = 0;
49       gp.auditCounter = 0;
50       // Save persistent data changes to NV
51       // Note: since there are so many changes to the persistent data structure, the
52       // entire PERSISTENT_DATA structure is written as a unit
53       NvWrite(NV_PERSISTENT_DATA, sizeof(PERSISTENT_DATA), &gp);
54       // Reset the PCR authValues (this does not change the PCRs)
55       PCR_ClearAuth();
56       // Bump the PCR counter
57       PCRChanged(0);
```

```
58        // orderly state should be cleared because of the update to state clear data
59        g_clearOrderly = TRUE;
60        return TPM_RC_SUCCESS;
61    }
62  #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 169 — 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 170 — TPM2_ClearControl Response

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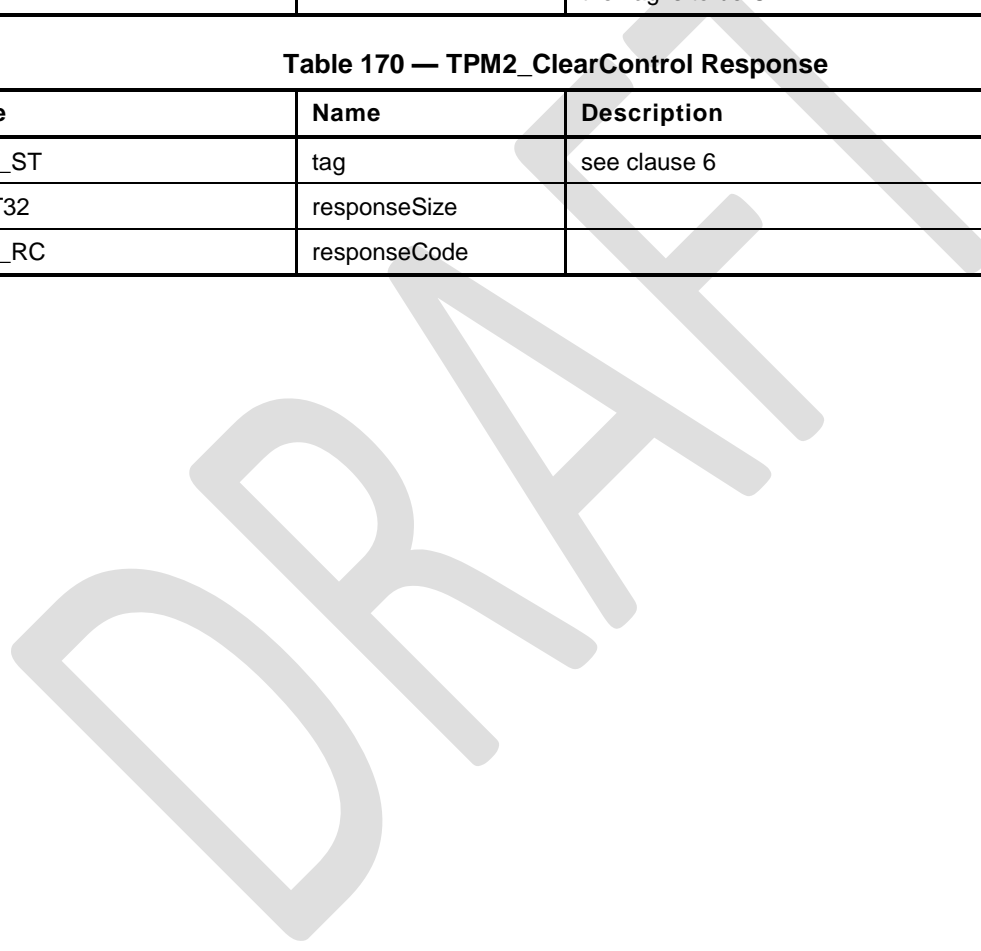
24.7.3   Detailed Actions

```c
1    #include "Tpm.h"
2    #include "ClearControl_fp.h"
3    #ifdef TPM_CC_ClearControl  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_AUTH_FAIL	authorization is not properly given

```c
4    TPM_RC
5    TPM2_ClearControl(
6        ClearControl_In     *in                 // IN: input parameter list
7        )
8    {
9        // The command needs NV update.
10       RETURN_IF_NV_IS_NOT_AVAILABLE;
11   // Input Validation
12       // LockoutAuth may be used to set disableLockoutClear to TRUE but not to FALSE
13       if(in->auth == TPM_RH_LOCKOUT && in->disable == NO)
14           return TPM_RC_AUTH_FAIL;
15   // Internal Data Update
16       if(in->disable == YES)
17           gp.disableClear = TRUE;
18       else
19           gp.disableClear = FALSE;
20       // Record the change to NV
21       NV_SYNC_PERSISTENT(disableClear);
22       return TPM_RC_SUCCESS;
23   }
24   #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 171 — 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 172 — TPM2_HierarchyChangeAuth Response

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

24.8.3   Detailed Actions

```
1    #include "Tpm.h"
2    #include "HierarchyChangeAuth_fp.h"
3    #ifdef TPM_CC_HierarchyChangeAuth  // Conditional expansion of this file
4    #include "Object_spt_fp.h"
```

Error Returns	Meaning
TPM_RC_SIZE	*newAuth* size is greater than that of integrity hash digest

```
5    TPM_RC
6    TPM2_HierarchyChangeAuth(
7        HierarchyChangeAuth_In  *in            // IN: input parameter list
8        )
9    {
10       // The command needs NV update.
11       RETURN_IF_NV_IS_NOT_AVAILABLE;
12       // Make sure that the authorization value is a reasonable size (not larger than
13       // the size of the digest produced by the integrity hash. The integrity
14       // hash is assumed to produce the longest digest of any hash implemented
15       // on the TPM. This will also remove trailing zeros from the authValue.
16       if(MemoryRemoveTrailingZeros(&in->newAuth) > CONTEXT_INTEGRITY_HASH_SIZE)
17           return TPM_RCS_SIZE + RC_HierarchyChangeAuth_newAuth;
18       // Set hierarchy authValue
19       switch(in->authHandle)
20       {
21           case TPM_RH_OWNER:
22               gp.ownerAuth = in->newAuth;
23               NV_SYNC_PERSISTENT(ownerAuth);
24               break;
25           case TPM_RH_ENDORSEMENT:
26               gp.endorsementAuth = in->newAuth;
27               NV_SYNC_PERSISTENT(endorsementAuth);
28               break;
29           case TPM_RH_PLATFORM:
30               gc.platformAuth = in->newAuth;
31               // orderly state should be cleared
32               g_clearOrderly = TRUE;
33               break;
34           case TPM_RH_LOCKOUT:
35               gp.lockoutAuth = in->newAuth;
36               NV_SYNC_PERSISTENT(lockoutAuth);
37               break;
38           default:
39               FAIL(FATAL_ERROR_INTERNAL);
40               break;
41       }
42       return TPM_RC_SUCCESS;
43   }
44   #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_LOCKED 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 173 — 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 174 — TPM2_DictionaryAttackLockReset Response

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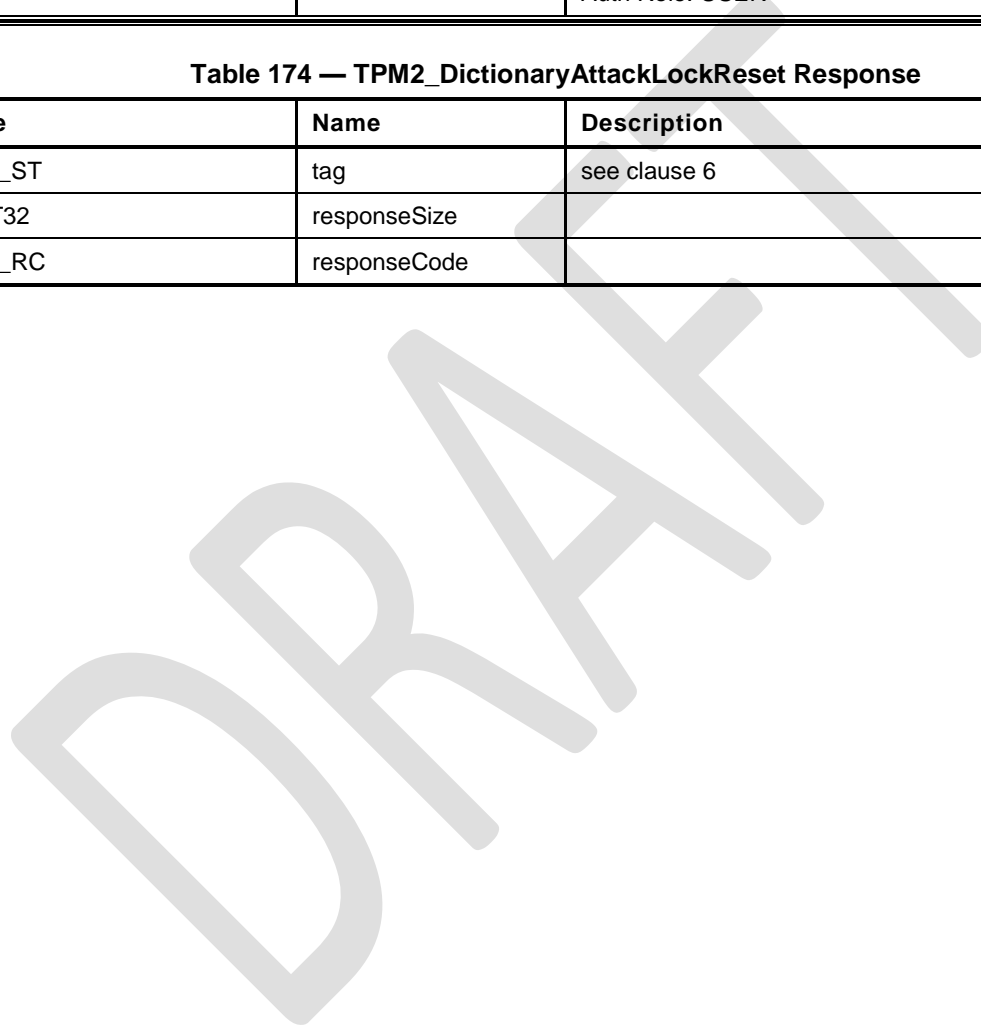
25.2.3   Detailed Actions

```
1   #include "Tpm.h"
2   #include "DictionaryAttackLockReset_fp.h"
3   #ifdef TPM_CC_DictionaryAttackLockReset  // Conditional expansion of this file
4   TPM_RC
5   TPM2_DictionaryAttackLockReset(
6       DictionaryAttackLockReset_In    *in              // IN: input parameter list
7       )
8   {
9       // Input parameter is not reference in command action
10      NOT_REFERENCED(in);
11      // The command needs NV update.
12      RETURN_IF_NV_IS_NOT_AVAILABLE;
13  // Internal Data Update
14      // Set failed tries to 0
15      gp.failedTries = 0;
16      // Record the changes to NV
17      NV_SYNC_PERSISTENT(failedTries);
18      return TPM_RC_SUCCESS;
19  }
20  #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().

This command will set the authorization failure count (*failedTries*) to zero.

Only one *lockoutAuth* authorization failure is allowed for this command during a *lockoutRecovery* interval.

25.3.2 Command and Response

Table 175 — 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 176 — TPM2_DictionaryAttackParameters Response

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

25.3.3   Detailed Actions

```
1    #include "Tpm.h"
2    #include "DictionaryAttackParameters_fp.h"
3    #ifdef TPM_CC_DictionaryAttackParameters   // Conditional expansion of this file
4    TPM_RC
5    TPM2_DictionaryAttackParameters(
6        DictionaryAttackParameters_In    *in            // IN: input parameter list
7        )
8    {
9        // The command needs NV update.
10       RETURN_IF_NV_IS_NOT_AVAILABLE;
11   // Internal Data Update
12       // Set dictionary attack parameters
13       gp.maxTries = in->newMaxTries;
14       gp.recoveryTime = in->newRecoveryTime;
15       gp.lockoutRecovery = in->lockoutRecovery;
16       // Set failed tries to 0
17       gp.failedTries = 0;
18       // Record the changes to NV
19       NV_SYNC_PERSISTENT(failedTries);
20       NV_SYNC_PERSISTENT(maxTries);
21       NV_SYNC_PERSISTENT(recoveryTime);
22       NV_SYNC_PERSISTENT(lockoutRecovery);
23       return TPM_RC_SUCCESS;
24   }
25   #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   Physical Presence may be made a requirement of any policy.

TPM2_PP_Commands() always requires assertion of Physical Presence.

26.2.2   Command and Response

Table 177 — 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 178 — TPM2_PP_Commands Response

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

26.2.3   Detailed Actions

```c
1    #include "Tpm.h"
2    #include "PP_Commands_fp.h"
3    #ifdef TPM_CC_PP_Commands  // Conditional expansion of this file
4    TPM_RC
5    TPM2_PP_Commands(
6        PP_Commands_In  *in              // IN: input parameter list
7        )
8    {
9        UINT32          i;
10       // The command needs NV update.  Check if NV is available.
11       // A TPM_RC_NV_UNAVAILABLE or TPM_RC_NV_RATE error may be returned at
12       // this point
13       RETURN_IF_NV_IS_NOT_AVAILABLE;
14   // Internal Data Update
15       // Process set list
16       for(i = 0; i < in->setList.count; i++)
17           // If command is implemented, set it as PP required.  If the input
18           // command is not a PP command, it will be ignored at
19           // PhysicalPresenceCommandSet().
20           // Note: PhysicalPresenceCommandSet() checks if the command is implemented.
21           PhysicalPresenceCommandSet(in->setList.commandCodes[i]);
22       // Process clear list
23       for(i = 0; i < in->clearList.count; i++)
24           // If command is implemented, clear it as PP required.  If the input
25           // command is not a PP command, it will be ignored at
26           // PhysicalPresenceCommandClear().  If the input command is
27           // TPM2_PP_Commands, it will be ignored as well
28           PhysicalPresenceCommandClear(in->clearList.commandCodes[i]);
29       // Save the change of PP list
30       NV_SYNC_PERSISTENT(ppList);
31       return TPM_RC_SUCCESS;
32   }
33   #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 enter Failure mode.

This command does not change the algorithms available to the platform.

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 179 — 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 180 — TPM2_SetAlgorithmSet Response

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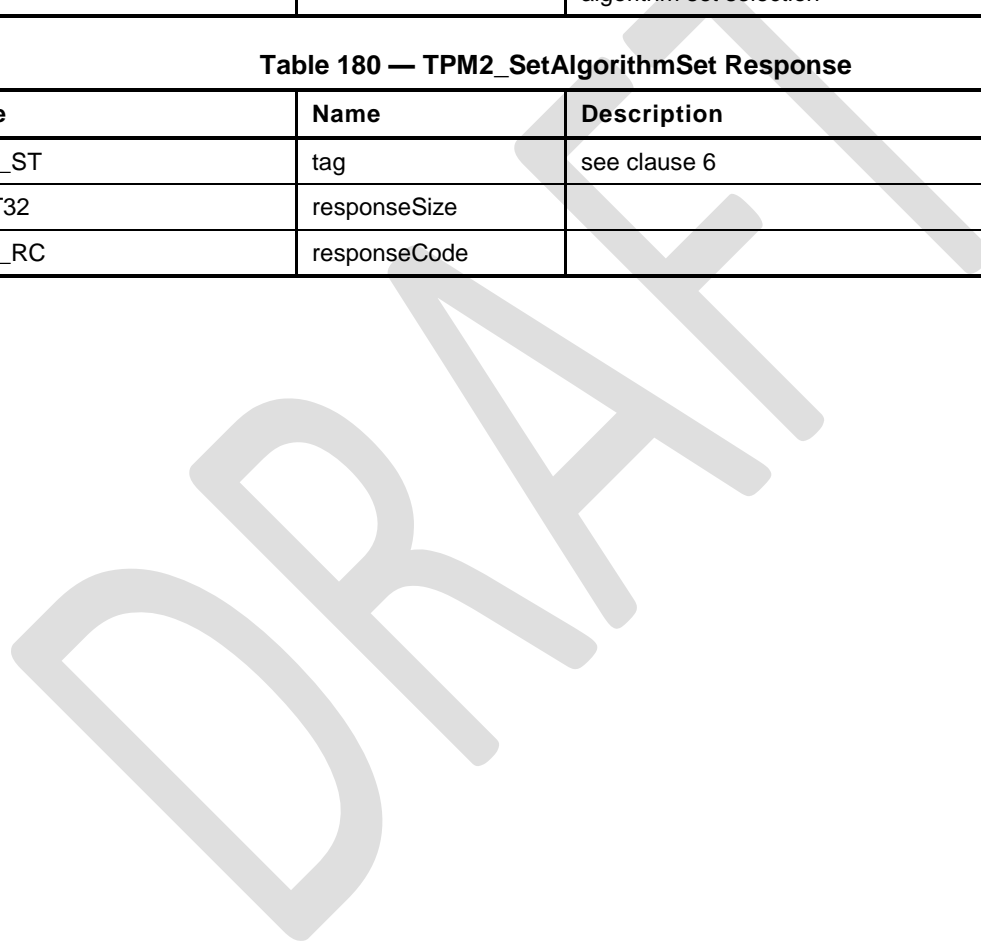
26.3.3   Detailed Actions

```
1    #include "Tpm.h"
2    #include "SetAlgorithmSet_fp.h"
3    #ifdef TPM_CC_SetAlgorithmSet   // Conditional expansion of this file
4    TPM_RC
5    TPM2_SetAlgorithmSet(
6        SetAlgorithmSet_In  *in            // IN: input parameter list
7        )
8    {
9        // The command needs NV update.  Check if NV is available.
10       // A TPM_RC_NV_UNAVAILABLE or TPM_RC_NV_RATE error may be returned at
11       // this point
12       RETURN_IF_NV_IS_NOT_AVAILABLE;
13   // Internal Data Update
14       gp.algorithmSet = in->algorithmSet;
15       // Write the algorithm set changes to NV
16       NV_SYNC_PERSISTENT(algorithmSet);
17       return TPM_RC_SUCCESS;
18   }
19   #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 shall 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 181 — 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 182 — TPM2_FieldUpgradeStart Response

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

27.2.3 Detailed Actions

```
1    #include "Tpm.h"
2    #include "FieldUpgradeStart_fp.h"
3    #ifdef TPM_CC_FieldUpgradeStart  // Conditional expansion of this file
4    TPM_RC
5    TPM2_FieldUpgradeStart(
6        FieldUpgradeStart_In    *in              // IN: input parameter list
7        )
8    {
9        // Not implemented
10       UNUSED_PARAMETER(in);
11       return TPM_RC_SUCCESS;
12   }
13   #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 183 — 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 184 — 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

```
1    #include "Tpm.h"
2    #include "FieldUpgradeData_fp.h"
3    #ifdef TPM_CC_FieldUpgradeData  // Conditional expansion of this file
4    TPM_RC
5    TPM2_FieldUpgradeData(
6        FieldUpgradeData_In     *in,           // IN: input parameter list
7        FieldUpgradeData_Out    *out           // OUT: output parameter list
8        )
9    {
10       // Not implemented
11       UNUSED_PARAMETER(in);
12       UNUSED_PARAMETER(out);
13       return TPM_RC_SUCCESS;
14   }
15   #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 185 — 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 186 — 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

```
1    #include "Tpm.h"
2    #include "FirmwareRead_fp.h"
3    #ifdef TPM_CC_FirmwareRead  // Conditional expansion of this file
4    TPM_RC
5    TPM2_FirmwareRead(
6        FirmwareRead_In     *in,          // IN: input parameter list
7        FirmwareRead_Out    *out          // OUT: output parameter list
8        )
9    {
10       // Not implemented
11       UNUSED_PARAMETER(in);
12       UNUSED_PARAMETER(out);
13       return TPM_RC_SUCCESS;
14   }
15   #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 187 — 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 188 — 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

```
1    #include "Tpm.h"
2    #include "ContextSave_fp.h"
3    #ifdef TPM_CC_ContextSave   // Conditional expansion of this file
4    #include "Context_spt_fp.h"
```

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().

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

```
5    TPM_RC
6    TPM2_ContextSave(
7        ContextSave_In      *in,            // IN: input parameter list
8        ContextSave_Out     *out            // OUT: output parameter list
9        )
10   {
11       TPM_RC          result = TPM_RC_SUCCESS;
12       UINT16          fingerprintSize;    // The size of fingerprint in context
13       // blob.
14       UINT64          contextID = 0;      // session context ID
15       TPM2B_SYM_KEY   symKey;
16       TPM2B_IV        iv;
17       TPM2B_DIGEST    integrity;
18       UINT16          integritySize;
19       BYTE            *buffer;
20       // This command may cause the orderlyState to be cleared due to
21       // the update of state reset data. If the state is orderly and
22       // cannot be changed, exit early.
23       RETURN_IF_ORDERLY;
24   // Internal Data Update
25       // Initialize output handle.  At the end of command action, the output
26       // handle of an object will be replaced, while the output handle
27       // for a session will be the same as input
28       out->context.savedHandle = in->saveHandle;
29       // Get the size of fingerprint in context blob.  The sequence value in
30       // TPMS_CONTEXT structure is used as the fingerprint
31       fingerprintSize = sizeof(out->context.sequence);
32       // Compute the integrity size at the beginning of context blob
33       integritySize = sizeof(integrity.t.size)
34           + CryptHashGetDigestSize(CONTEXT_INTEGRITY_HASH_ALG);
35   // Perform object or session specific context save
36       switch(HandleGetType(in->saveHandle))
37       {
38           case TPM_HT_TRANSIENT:
39           {
40               OBJECT              *object = HandleToObject(in->saveHandle);
41               ANY_OBJECT_BUFFER   *outObject;
42               UINT16               objectSize = ObjectIsSequence(object)
43                   ? sizeof(HASH_OBJECT) : sizeof(OBJECT);
44               outObject = (ANY_OBJECT_BUFFER *)(out->context.contextBlob.t.buffer
45                                       + integritySize + fingerprintSize);
46               // Set size of the context data.  The contents of context blob is vendor
47               // defined.  In this implementation, the size is size of integrity
48               // plus fingerprint plus the whole internal OBJECT structure
```

```
49              out->context.contextBlob.t.size = integritySize +
50                  fingerprintSize + objectSize;
51  // Make sure things fit
52              pAssert(out->context.contextBlob.t.size
53                  <= sizeof(out->context.contextBlob.t.buffer));
54              // Copy the whole internal OBJECT structure to context blob
55              MemoryCopy(outObject, object, objectSize);
56              // Increment object context ID
57              gr.objectContextID++;
58              // If object context ID overflows, TPM should be put in failure mode
59              if(gr.objectContextID == 0)
60                  FAIL(FATAL_ERROR_INTERNAL);
61              // Fill in other return values for an object.
62              out->context.sequence = gr.objectContextID;
63              // For regular object, savedHandle is 0x80000000.  For sequence object,
64              // savedHandle is 0x80000001.  For object with stClear, savedHandle
65              // is 0x80000002
66              if(ObjectIsSequence(object))
67              {
68                  out->context.savedHandle = 0x80000001;
69                  SequenceDataExport((HASH_OBJECT *)object,
70                                     (HASH_OBJECT_BUFFER *)outObject);
71              }
72              else
73                  out->context.savedHandle = (object->attributes.stClear == SET)
74                  ? 0x80000002 : 0x80000000;
75  // Get object hierarchy
76              out->context.hierarchy = ObjectGetHierarchy(object);
77              break;
78          }
79          case TPM_HT_HMAC_SESSION:
80          case TPM_HT_POLICY_SESSION:
81          {
82              SESSION         *session = SessionGet(in->saveHandle);
83              // Set size of the context data.  The contents of context blob is vendor
84              // defined.  In this implementation, the size of context blob is the
85              // size of a internal session structure plus the size of
86              // fingerprint plus the size of integrity
87              out->context.contextBlob.t.size = integritySize +
88                  fingerprintSize + sizeof(*session);
89  // Make sure things fit
90              pAssert(out->context.contextBlob.t.size
91                  < sizeof(out->context.contextBlob.t.buffer));
92              // Copy the whole internal SESSION structure to context blob.
93              // Save space for fingerprint at the beginning of the buffer
94              // This is done before anything else so that the actual context
95              // can be reclaimed after this call
96              pAssert(sizeof(*session) <= sizeof(out->context.contextBlob.t.buffer)
97                  - integritySize - fingerprintSize);
98              MemoryCopy(out->context.contextBlob.t.buffer + integritySize
99                  + fingerprintSize, session, sizeof(*session));
100     // Fill in the other return parameters for a session
101     // Get a context ID and set the session tracking values appropriately
102     // TPM_RC_CONTEXT_GAP is a possible error.
103     // SessionContextSave() will flush the in-memory context
104     // so no additional errors may occur after this call.
105             result = SessionContextSave(out->context.savedHandle, &contextID);
106             if(result != TPM_RC_SUCCESS)
107                 return result;
108             // sequence number is the current session contextID
109             out->context.sequence = contextID;
110             // use TPM_RH_NULL as hierarchy for session context
111             out->context.hierarchy = TPM_RH_NULL;
112             break;
113         }
114         default:
```

```
115                    // SaveContext may only take an object handle or a session handle.
116                    // All the other handle type should be filtered out at unmarshal
117                    FAIL(FATAL_ERROR_INTERNAL);
118                    break;
119            }
120        // Save fingerprint at the beginning of encrypted area of context blob.
121        // Reserve the integrity space
122        pAssert(sizeof(out->context.sequence) <=
123                sizeof(out->context.contextBlob.t.buffer) - integritySize);
124        MemoryCopy(out->context.contextBlob.t.buffer + integritySize,
125                   &out->context.sequence, sizeof(out->context.sequence));
126        // Compute context encryption key
127        ComputeContextProtectionKey(&out->context, &symKey, &iv);
128        // Encrypt context blob
129        CryptSymmetricEncrypt(out->context.contextBlob.t.buffer + integritySize,
130                             CONTEXT_ENCRYPT_ALG, CONTEXT_ENCRYPT_KEY_BITS,
131                             symKey.t.buffer, &iv, TPM_ALG_CFB,
132                             out->context.contextBlob.t.size - integritySize,
133                             out->context.contextBlob.t.buffer + integritySize);
134        // Compute integrity hash for the object
135        // In this implementation, the same routine is used for both sessions
136        // and objects.
137        ComputeContextIntegrity(&out->context, &integrity);
138        // add integrity at the beginning of context blob
139        buffer = out->context.contextBlob.t.buffer;
140        TPM2B_DIGEST_Marshal(&integrity, &buffer, NULL);
141        // orderly state should be cleared because of the update of state reset and
142        // state clear data
143        g_clearOrderly = TRUE;
144        return result;
145    }
146  #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 189 — 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 190 — 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

```c
1   #include "Tpm.h"
2   #include "ContextLoad_fp.h"
3   #ifdef TPM_CC_ContextLoad  // Conditional expansion of this file
4   #include "Context_spt_fp.h"
```

See discussion about the context format in ContextSave.c.

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

```c
5   TPM_RC
6   TPM2_ContextLoad(
7       ContextLoad_In      *in,            // IN: input parameter list
8       ContextLoad_Out     *out            // OUT: output parameter list
9       )
10  {
11  // Local Variables
12      TPM_RC              result;
13      TPM2B_DIGEST        integrityToCompare;
14      TPM2B_DIGEST        integrity;
15      BYTE                *buffer;    // defined to save some typing
16      INT32               size;       // defined to save some typing
17      TPM_HT              handleType;
18      TPM2B_SYM_KEY       symKey;
19      TPM2B_IV            iv;
20  // Input Validation
21      // IF this is a session context, make sure that the sequence number is
22      // consistent with the version in the slot
23      // Check context blob size
24      handleType = HandleGetType(in->context.savedHandle);
25      // Get integrity from context blob
26      buffer = in->context.contextBlob.t.buffer;
27      size = (INT32)in->context.contextBlob.t.size;
28      result = TPM2B_DIGEST_Unmarshal(&integrity, &buffer, &size);
29      if(result != TPM_RC_SUCCESS)
30          return result;
31      // the size of the integrity value has to match the size of digest produced
32      // by the integrity hash
33      if(integrity.t.size != CryptHashGetDigestSize(CONTEXT_INTEGRITY_HASH_ALG))
34          return TPM_RCS_SIZE + RC_ContextLoad_context;
35      // Make sure that the context blob has enough space for the fingerprint. This
36      // is elastic pants to go with the belt and suspenders we already have to make
37      // sure that the context is complete and untampered.
38      if((unsigned)size < sizeof(in->context.sequence))
39          return TPM_RCS_SIZE + RC_ContextLoad_context;
40      // After unmarshaling the integrity value, 'buffer' is pointing at the first
41      // byte of the integrity protected and encrypted buffer and 'size' is the number
42      // of integrity protected and encrypted bytes.
43      // Compute context integrity
44      ComputeContextIntegrity(&in->context, &integrityToCompare);
```

```
45        // Compare integrity
46        if(!MemoryEqual2B(&integrity.b, &integrityToCompare.b))
47            return TPM_RCS_INTEGRITY + RC_ContextLoad_context;
48        // Compute context encryption key
49        ComputeContextProtectionKey(&in->context, &symKey, &iv);
50        // Decrypt context data in place
51        CryptSymmetricDecrypt(buffer, CONTEXT_ENCRYPT_ALG, CONTEXT_ENCRYPT_KEY_BITS,
52                              symKey.t.buffer, &iv, TPM_ALG_CFB, size, buffer);
53        // See if the fingerprint value matches. If not, it is symptomatic of either
54        // a broken TPM or that the TPM is under attack so go into failure mode.
55        if(!MemoryEqual(buffer, &in->context.sequence, sizeof(in->context.sequence)))
56            FAIL(FATAL_ERROR_INTERNAL);
57        // step over fingerprint
58        buffer += sizeof(in->context.sequence);
59        // set the remaining size of the context
60        size -= sizeof(in->context.sequence);
61        // Perform object or session specific input check
62        switch(handleType)
63        {
64            case TPM_HT_TRANSIENT:
65            {
66                OBJECT        *outObject;
67                if(size > (INT32)sizeof(OBJECT))
68                    FAIL(FATAL_ERROR_INTERNAL);
69                // Discard any changes to the handle that the TRM might have made
70                in->context.savedHandle = TRANSIENT_FIRST;
71                // If hierarchy is disabled, no object context can be loaded in this
72                // hierarchy
73                if(!HierarchyIsEnabled(in->context.hierarchy))
74                    return TPM_RCS_HIERARCHY + RC_ContextLoad_context;
75                // Restore object. If there is no empty space, indicate as much
76                outObject = ObjectContextLoad((ANY_OBJECT_BUFFER *)buffer,
77                                              &out->loadedHandle);
78                if(outObject == NULL)
79                    return TPM_RC_OBJECT_MEMORY;
80                break;
81            }
82            case TPM_HT_POLICY_SESSION:
83            case TPM_HT_HMAC_SESSION:
84            {
85                if(size != sizeof(SESSION))
86                    FAIL(FATAL_ERROR_INTERNAL);
87                // This command may cause the orderlyState to be cleared due to
88                // the update of state reset data.  If this is the case, check if NV is
89                // available first
90                RETURN_IF_ORDERLY;
91                // Check if input handle points to a valid saved session and that the
92                // sequence number makes sense
93                if(!SequenceNumbereForSavedContextIsValid(&in->context))
94                    return TPM_RCS_HANDLE + RC_ContextLoad_context;
95                // Restore session.  A TPM_RC_SESSION_MEMORY, TPM_RC_CONTEXT_GAP error
96                // may be returned at this point
97                result = SessionContextLoad((SESSION_BUF *)buffer,
98                                            &in->context.savedHandle);
99                if(result != TPM_RC_SUCCESS)
100                    return result;
101                out->loadedHandle = in->context.savedHandle;
102                // orderly state should be cleared because of the update of state
103                // reset and state clear data
104                g_clearOrderly = TRUE;
105                break;
106            }
107            default:
108                // Context blob may only have an object handle or a session handle.
109                // All the other handle type should be filtered out at unmarshal
110                FAIL(FATAL_ERROR_INTERNAL);
```

```
111                break;
112        }
113        return TPM_RC_SUCCESS;
114 }
115 #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.

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.

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 191 — 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 192 — TPM2_FlushContext Response

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

28.4.3   Detailed Actions

```
1    #include "Tpm.h"
2    #include "FlushContext_fp.h"
3    #ifdef TPM_CC_FlushContext  // Conditional expansion of this file
```

Error Returns	Meaning
TPM_RC_HANDLE	*flushHandle* does not reference a loaded object or session

```
4    TPM_RC
5    TPM2_FlushContext(
6        FlushContext_In    *in               // IN: input parameter list
7        )
8    {
9    // Internal Data Update
10       // Call object or session specific routine to flush
11       switch(HandleGetType(in->flushHandle))
12       {
13           case TPM_HT_TRANSIENT:
14               if(!IsObjectPresent(in->flushHandle))
15                   return TPM_RCS_HANDLE + RC_FlushContext_flushHandle;
16           // Flush object
17           FlushObject(in->flushHandle);
18           break;
19           case TPM_HT_HMAC_SESSION:
20           case TPM_HT_POLICY_SESSION:
21               if(!SessionIsLoaded(in->flushHandle)
22                  && !SessionIsSaved(in->flushHandle)
23                  )
24                   return TPM_RCS_HANDLE + RC_FlushContext_flushHandle;
25           // If the session to be flushed is the exclusive audit session, then
26           // indicate that there is no exclusive audit session any longer.
27           if(in->flushHandle == g_exclusiveAuditSession)
28               g_exclusiveAuditSession = TPM_RH_UNASSIGNED;
29           // Flush session
30           SessionFlush(in->flushHandle);
31           break;
32           default:
33               // This command only takes object or session handle.  Other handles
34               // should be filtered out at handle unmarshal
35               FAIL(FATAL_ERROR_INTERNAL);
36               break;
37       }
38       return TPM_RC_SUCCESS;
39   }
40   #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

    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$_{16}$.

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 3   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 4   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 193 — 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 194 — TPM2_EvictControl Response

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

28.5.3   Detailed Actions

```
1    #include "Tpm.h"
2    #include "EvictControl_fp.h"
3    #ifdef TPM_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*

```
4    TPM_RC
5    TPM2_EvictControl(
6        EvictControl_In    *in            // IN: input parameter list
7        )
8    {
9        TPM_RC        result;
10       OBJECT        *evictObject;
11   // Input Validation
12       // Get internal object pointer
13       evictObject = HandleToObject(in->objectHandle);
14       // Temporary, stClear or public only objects can not be made persistent
15       if(evictObject->attributes.temporary == SET
16          || evictObject->attributes.stClear == SET
17          || evictObject->attributes.publicOnly == SET)
18          return TPM_RCS_ATTRIBUTES + RC_EvictControl_objectHandle;
19       // If objectHandle refers to a persistent object, it should be the same as
20       // input persistentHandle
21       if(evictObject->attributes.evict == SET
22          && evictObject->evictHandle != in->persistentHandle)
23          return TPM_RCS_HANDLE + RC_EvictControl_objectHandle;
24       // Additional authorization validation
25       if(in->auth == TPM_RH_PLATFORM)
26       {
27           // To make persistent
28           if(evictObject->attributes.evict == CLEAR)
29           {
30               // PlatformAuth can not set evict object in storage or endorsement
31               // hierarchy
32               if(evictObject->attributes.ppsHierarchy == CLEAR)
33                   return TPM_RCS_HIERARCHY + RC_EvictControl_objectHandle;
34               // Platform cannot use a handle outside of platform persistent range.
35               if(!NvIsPlatformPersistentHandle(in->persistentHandle))
36                   return TPM_RCS_RANGE + RC_EvictControl_persistentHandle;
37           }
38           // PlatformAuth can delete any persistent object
39       }
40       else if(in->auth == TPM_RH_OWNER)
41       {
42           // OwnerAuth can not set or clear evict object in platform hierarchy
43           if(evictObject->attributes.ppsHierarchy == SET)
44               return TPM_RCS_HIERARCHY + RC_EvictControl_objectHandle;
45           // Owner cannot use a handle outside of owner persistent range.
46           if(evictObject->attributes.evict == CLEAR
```

```
47                && !NvIsOwnerPersistentHandle(in->persistentHandle))
48                    return TPM_RCS_RANGE + RC_EvictControl_persistentHandle;
49        }
50        else
51        {
52            // Other authorization is not allowed in this command and should have been
53            // filtered out in unmarshal process
54            FAIL(FATAL_ERROR_INTERNAL);
55        }
56    // Internal Data Update
57        // Change evict state
58        if(evictObject->attributes.evict == CLEAR)
59        {
60            // Make object persistent
61            if(NvFindHandle(in->persistentHandle) != 0)
62                return TPM_RC_NV_DEFINED;
63            // A TPM_RC_NV_HANDLE or TPM_RC_NV_SPACE error may be returned at this
64            // point
65            result = NvAddEvictObject(in->persistentHandle, evictObject);
66        }
67        else
68        {
69            // Delete the persistent object in NV
70            result = NvDeleteEvict(evictObject->evictHandle);
71        }
72        return result;
73    }
74    #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 195 — TPM2_ReadClock 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_ReadClock

Table 196 — 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

```
1   #include "Tpm.h"
2   #include "ReadClock_fp.h"
3   #ifdef TPM_CC_ReadClock  // Conditional expansion of this file
4   TPM_RC
5   TPM2_ReadClock(
6       ReadClock_Out    *out              // OUT: output parameter list
7       )
8   {
9   // Command Output
10      out->currentTime.time = g_time;
11      TimeFillInfo(&out->currentTime.clockInfo);
12      return TPM_RC_SUCCESS;
13  }
14  #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 197 — 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 198 — TPM2_ClockSet Response

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

29.2.3   Detailed Actions

```
1    #include "Tpm.h"
2    #include "ClockSet_fp.h"
3    #ifdef TPM_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

```
4    TPM_RC
5    TPM2_ClockSet(
6        ClockSet_In      *in               // IN: input parameter list
7        )
8    {
9    #define CLOCK_UPDATE_MASK  ~((1ULL << NV_CLOCK_UPDATE_INTERVAL)- 1)
10   // Input Validation
11       // new time can not be bigger than 0xFFFF000000000000 or smaller than
12       // current clock
13       if(in->newTime > 0xFFFF000000000000ULL
14          || in->newTime < go.clock)
15           return TPM_RCS_VALUE + RC_ClockSet_newTime;
16   // Internal Data Update
17       return TimeClockUpdate(in->newTime);
18   }
19   #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 199 — 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 200 — TPM2_ClockRateAdjust Response

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

29.3.3   Detailed Actions

```c
1    #include "Tpm.h"
2    #include "ClockRateAdjust_fp.h"
3    #ifdef TPM_CC_ClockRateAdjust  // Conditional expansion of this file
4    TPM_RC
5    TPM2_ClockRateAdjust(
6        ClockRateAdjust_In  *in              // IN: input parameter list
7        )
8    {
9    // Internal Data Update
10       TimeSetAdjustRate(in->rateAdjust);
11       return TPM_RC_SUCCESS;
12   }
13   #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 set to TPML_TAGGED_PCR_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 will return a zero length list with the moreData parameter set to NO. 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_CURVE	TPM_ECC_CURVE[(1)]	TPML_ECC_CURVE
TPM_CAP_VENDOR_PROPERTY	manufacturer specific	manufacturer-specific values
NOTES: (1) The TPM_ALG_ID or TPM_ECC_CURVE is cast to a UINT32		

- 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.

- 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.

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 201 — 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 202 — 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

```
1    #include "Tpm.h"
2    #include "GetCapability_fp.h"
3    #ifdef TPM_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

```
4    TPM_RC
5    TPM2_GetCapability(
6        GetCapability_In    *in,            // IN: input parameter list
7        GetCapability_Out   *out            // OUT: output parameter list
8        )
9    {
10       TPMU_CAPABILITIES   *data = &out->capabilityData.data;
11   // Command Output
12       // Set output capability type the same as input type
13       out->capabilityData.capability = in->capability;
14       switch(in->capability)
15       {
16           case TPM_CAP_ALGS:
17               out->moreData = AlgorithmCapGetImplemented((TPM_ALG_ID)in->property,
18                                                     in->propertyCount,
19                                                     &data->algorithms);
20               break;
21           case TPM_CAP_HANDLES:
22               switch(HandleGetType((TPM_HANDLE)in->property))
23               {
24                   case TPM_HT_TRANSIENT:
25                       // Get list of handles of loaded transient objects
26                       out->moreData = ObjectCapGetLoaded((TPM_HANDLE)in->property,
27                                                     in->propertyCount,
28                                                     &data->handles);
29                       break;
30                   case TPM_HT_PERSISTENT:
31                       // Get list of handles of persistent objects
32                       out->moreData = NvCapGetPersistent((TPM_HANDLE)in->property,
33                                                     in->propertyCount,
34                                                     &data->handles);
35                       break;
36                   case TPM_HT_NV_INDEX:
37                       // Get list of defined NV index
38                       out->moreData = NvCapGetIndex((TPM_HANDLE)in->property,
39                                                 in->propertyCount,
40                                                 &data->handles);
41                       break;
42                   case TPM_HT_LOADED_SESSION:
43                       // Get list of handles of loaded sessions
44                       out->moreData = SessionCapGetLoaded((TPM_HANDLE)in->property,
45                                                     in->propertyCount,
46                                                     &data->handles);
47                       break;
48   #ifdef TPM_HT_SAVED_SESSION
49                   case TPM_HT_SAVED_SESSION:
50   #else
51                   case TPM_HT_ACTIVE_SESSION:
52   #endif
53                       // Get list of handles of
54                       out->moreData = SessionCapGetSaved((TPM_HANDLE)in->property,
```

```
55                                              in->propertyCount,
56                                              &data->handles);
57                       break;
58                   case TPM_HT_PCR:
59                       // Get list of handles of PCR
60                       out->moreData = PCRCapGetHandles((TPM_HANDLE)in->property,
61                                              in->propertyCount,
62                                              &data->handles);
63                       break;
64                   case TPM_HT_PERMANENT:
65                       // Get list of permanent handles
66                       out->moreData = PermanentCapGetHandles((TPM_HANDLE)in->property,
67                                                  in->propertyCount,
68                                                  &data->handles);
69                       break;
70                   default:
71                       // Unsupported input handle type
72                       return TPM_RCS_HANDLE + RC_GetCapability_property;
73                       break;
74               }
75               break;
76           case TPM_CAP_COMMANDS:
77               out->moreData = CommandCapGetCCList((TPM_CC)in->property,
78                                          in->propertyCount,
79                                          &data->command);
80               break;
81           case TPM_CAP_PP_COMMANDS:
82               out->moreData = PhysicalPresenceCapGetCCList((TPM_CC)in->property,
83                                              in->propertyCount,
84                                              &data->ppCommands);
85               break;
86           case TPM_CAP_AUDIT_COMMANDS:
87               out->moreData = CommandAuditCapGetCCList((TPM_CC)in->property,
88                                              in->propertyCount,
89                                              &data->auditCommands);
90               break;
91           case TPM_CAP_PCRS:
92               // Input property must be 0
93               if(in->property != 0)
94                   return TPM_RCS_VALUE + RC_GetCapability_property;
95               out->moreData = PCRCapGetAllocation(in->propertyCount,
96                                          &data->assignedPCR);
97               break;
98           case TPM_CAP_PCR_PROPERTIES:
99               out->moreData = PCRCapGetProperties((TPM_PT_PCR)in->property,
100                                              in->propertyCount,
101                                              &data->pcrProperties);
102               break;
103          case TPM_CAP_TPM_PROPERTIES:
104               out->moreData = TPMCapGetProperties((TPM_PT)in->property,
105                                              in->propertyCount,
106                                              &data->tpmProperties);
107               break;
108  #ifdef TPM_ALG_ECC
109          case TPM_CAP_ECC_CURVES:
110               out->moreData = CryptCapGetECCCurve((TPM_ECC_CURVE)in->property,
111                                              in->propertyCount,
112                                              &data->eccCurves);
113               break;
114  #endif // TPM_ALG_ECC
115          case TPM_CAP_AUTH_POLICIES:
116               if(HandleGetType((TPM_HANDLE)in->property) != TPM_HT_PERMANENT)
117                   return TPM_RCS_VALUE + RC_GetCapability_property;
118               out->moreData = PermanentHandleGetPolicy((TPM_HANDLE)in->property,
119                                              in->propertyCount,
120                                              &data->authPolicies);
```

```
121                  break;
122             case TPM_CAP_VENDOR_PROPERTY:
123                  // vendor property is not implemented
124             default:
125                  // Unsupported TPM_CAP value
126                  return TPM_RC_VALUE + RC_GetCapability_capability;
127                  break;
128         }
129     return TPM_RC_SUCCESS;
130     }
131  #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

<p style="text-align:center">Table 203 — TPM2_TestParms Command</p>

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

<p style="text-align:center">Table 204 — TPM2_TestParms Response</p>

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

30.3.3 Detailed Actions

```c
1   #include "Tpm.h"
2   #include "TestParms_fp.h"
3   #ifdef TPM_CC_TestParms  // Conditional expansion of this file
4   TPM_RC
5   TPM2_TestParms(
6       TestParms_In    *in             // IN: input parameter list
7       )
8   {
9       // Input parameter is not reference in command action
10      NOT_REFERENCED(in);
11      // The parameters are tested at unmarshal process.  We do nothing in command
12      // action
13      return TPM_RC_SUCCESS;
14  }
15  #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 205 — 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 206 — TPM2_NV_DefineSpace Response

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

31.3.3 Detailed Actions

```
1   #include "Tpm.h"
2   #include "NV_DefineSpace_fp.h"
3   #ifdef TPM_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_NV_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)

```
4   TPM_RC
5   TPM2_NV_DefineSpace(
6       NV_DefineSpace_In   *in                // IN: input parameter list
7       )
8   {
9       TPMA_NV         attributes;
10      UINT16          nameSize;
11      nameSize = CryptHashGetDigestSize(in->publicInfo.nvPublic.nameAlg);
12  // Input Validation
13      // Checks not specific to type
14      // If the UndefineSpaceSpecial command is not implemented, then can't have
15      // an index that can only be deleted with policy
16  #if CC_NV_UndefineSpaceSpecial == NO
17      if(IsNv_TPMA_NV_POLICY_DELETE(attributes))
18          return TPM_RCS_ATTRIBUTES + RC_NV_DefineSpace_publicInfo;
19  #endif
20      // check that the authPolicy consistent with hash algorithm
21      if(in->publicInfo.nvPublic.authPolicy.t.size != 0
22         && in->publicInfo.nvPublic.authPolicy.t.size != nameSize)
23          return TPM_RCS_SIZE + RC_NV_DefineSpace_publicInfo;
24      // make sure that the authValue is not too large
25      if(MemoryRemoveTrailingZeros(&in->auth)
26         > CryptHashGetDigestSize(in->publicInfo.nvPublic.nameAlg))
27          return TPM_RCS_SIZE + RC_NV_DefineSpace_auth;
28      // If an index is being created by the owner and shEnable is
29      // clear, then we would not reach this point because ownerAuth
30      // can't be given when shEnable is CLEAR. However, if phEnable
31      // is SET but phEnableNV is CLEAR, we have to check here
32      if(in->authHandle == TPM_RH_PLATFORM && gc.phEnableNV == CLEAR)
33          return TPM_RCS_HIERARCHY + RC_NV_DefineSpace_authHandle;
34      attributes = in->publicInfo.nvPublic.attributes;
35      // Attribute checks
36      // Eliminate the unsupported types
37      switch(NV_ATTRIBUTES_TO_TYPE(attributes))
38      {
39  #if CC_NV_Increment == YES
40          case TPM_NT_COUNTER:
41  #endif
42  #if CC_NV_SetBits == YES
43          case TPM_NT_BITS:
44  #endif
45  #if CC_NV_Extend == YES
46          case TPM_NT_EXTEND:
```

Family "2.0"

TCG Public Review

Page 398

Level 00 Revision 01.36

Copyright © TCG 2006-2016

September 16, 2016

```
47      #endif
48      #if CC_PolicySigned == YES && defined TPM_NT_PIN_PASS
49              case TPM_NT_PIN_PASS:
50              case TPM_NT_PIN_FAIL:
51      #endif
52              case TPM_NT_ORDINARY:
53                  break;
54              default:
55                  return TPM_RCS_ATTRIBUTES + RC_NV_DefineSpace_publicInfo;
56                  break;
57          }
58      // Check that the sizes are OK based on the type
59      switch(NV_ATTRIBUTES_TO_TYPE(attributes))
60          {
61              case TPM_NT_ORDINARY:
62                  // Can't exceed the allowed size for the implementation
63                  if(in->publicInfo.nvPublic.dataSize > MAX_NV_INDEX_SIZE)
64                      return TPM_RCS_SIZE + RC_NV_DefineSpace_publicInfo;
65                  break;
66              case TPM_NT_EXTEND:
67                  if(in->publicInfo.nvPublic.dataSize != nameSize)
68                      return TPM_RCS_SIZE + RC_NV_DefineSpace_publicInfo;
69                  break;
70              default:
71                  // Everything else needs a size of 8
72                  if(in->publicInfo.nvPublic.dataSize != 8)
73                      return TPM_RCS_SIZE + RC_NV_DefineSpace_publicInfo;
74                  break;
75          }
76      // Handle other specifics
77      switch(NV_ATTRIBUTES_TO_TYPE(attributes))
78          {
79              case TPM_NT_COUNTER:
80                  // Counter can't have TPMA_NV_CLEAR_STCLEAR SET (don't clear counters)
81                  if(IsNv_TPMA_NV_CLEAR_STCLEAR(attributes))
82                      return TPM_RCS_ATTRIBUTES + RC_NV_DefineSpace_publicInfo;
83                  break;
84      #ifdef TPM_NT_PIN_FAIL
85              case TPM_NT_PIN_FAIL:
86                  // NV_NO_DA must be SET and AUTHWRITE must be CLEAR
87                  // NOTE: As with a PIN_PASS index, the authValue of the index is not
88                  // available until the index is written. If AUTHWRITE is the only way to
89                  // write then index, it could never be written. Rather than go through
90                  // all of the other possible ways to write the Index, it is simply
91                  // prohibited to write the index with the authValue. Other checks
92                  // below will insure that there seems to be a way to write the index
93                  // (i.e., with platform authorization , owner authorization,
94                  // or with policyAuth.)
95                  // It is not allowed to create a PIN Index that can't be modified.
96                  if(!IsNv_TPMA_NV_NO_DA(attributes))
97                      return TPM_RCS_ATTRIBUTES + RC_NV_DefineSpace_publicInfo;
98      #endif
99      #ifdef TPM_NT_PIN_PASS
100             case TPM_NT_PIN_PASS:
101                 // AUTHWRITE must be CLEAR (see note above to TPM_NT_PIN_FAIL)
102                 if(IsNv_TPMA_NV_AUTHWRITE(attributes)
103                     || IsNv_TPMA_NV_GLOBALLOCK(attributes)
104                     || IsNv_TPMA_NV_WRITEDEFINE(attributes))
105                     return TPM_RCS_ATTRIBUTES + RC_NV_DefineSpace_publicInfo;
106     #endif  // this comes before break because PIN_FAIL falls through
107                 break;
108             default:
109                 break;
110         }
111     // Locks may not be SET and written cannot be SET
112     if(IsNv_TPMA_NV_WRITTEN(attributes)
```

```
113              || IsNv_TPMA_NV_WRITELOCKED(attributes)
114              || IsNv_TPMA_NV_READLOCKED(attributes))
115              return TPM_RCS_ATTRIBUTES + RC_NV_DefineSpace_publicInfo;
116          // There must be a way to read the index.
117          if(!IsNv_TPMA_NV_OWNERREAD(attributes)
118              && !IsNv_TPMA_NV_PPREAD(attributes)
119              && !IsNv_TPMA_NV_AUTHREAD(attributes)
120              && !IsNv_TPMA_NV_POLICYREAD(attributes))
121              return TPM_RCS_ATTRIBUTES + RC_NV_DefineSpace_publicInfo;
122          // There must be a way to write the index
123          if(!IsNv_TPMA_NV_OWNERWRITE(attributes)
124              && !IsNv_TPMA_NV_PPWRITE(attributes)
125              && !IsNv_TPMA_NV_AUTHWRITE(attributes)
126              && !IsNv_TPMA_NV_POLICYWRITE(attributes))
127              return TPM_RCS_ATTRIBUTES + RC_NV_DefineSpace_publicInfo;
128          // An index with TPMA_NV_CLEAR_STCLEAR can't have TPMA_NV_WRITEDEFINE SET
129          if(IsNv_TPMA_NV_CLEAR_STCLEAR(attributes)
130              &&  IsNv_TPMA_NV_WRITEDEFINE(attributes))
131              return TPM_RCS_ATTRIBUTES + RC_NV_DefineSpace_publicInfo;
132          // Make sure that the creator of the index can delete the index
133          if((IsNv_TPMA_NV_PLATFORMCREATE(attributes)
134              && in->authHandle == TPM_RH_OWNER)
135              || (!IsNv_TPMA_NV_PLATFORMCREATE(attributes)
136                  && in->authHandle == TPM_RH_PLATFORM))
137              return TPM_RCS_ATTRIBUTES + RC_NV_DefineSpace_authHandle;
138          // If TPMA_NV_POLICY_DELETE is SET, then the index must be defined by
139          // the platform
140          if(IsNv_TPMA_NV_POLICY_DELETE(attributes)
141              &&  TPM_RH_PLATFORM != in->authHandle)
142              return TPM_RCS_ATTRIBUTES + RC_NV_DefineSpace_publicInfo;
143          // Make sure that the TPMA_NV_WRITEALL is not set if the index size is larger
144          // than the allowed NV buffer size.
145          if(in->publicInfo.nvPublic.dataSize > MAX_NV_BUFFER_SIZE
146              &&  IsNv_TPMA_NV_WRITEALL(attributes))
147              return TPM_RCS_SIZE + RC_NV_DefineSpace_publicInfo;
148          // And finally, see if the index is already defined.
149          if(NvIndexIsDefined(in->publicInfo.nvPublic.nvIndex))
150              return TPM_RC_NV_DEFINED;
151      // Internal Data Update
152          // define the space.  A TPM_RC_NV_SPACE error may be returned at this point
153          return NvDefineIndex(&in->publicInfo.nvPublic, &in->auth);
154      }
155  #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 207 — 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 208 — TPM2_NV_UndefineSpace Response

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

31.4.3 Detailed Actions

```
1   #include "Tpm.h"
2   #include "NV_UndefineSpace_fp.h"
3   #ifdef TPM_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

```
4   TPM_RC
5   TPM2_NV_UndefineSpace(
6       NV_UndefineSpace_In     *in              // IN: input parameter list
7       )
8   {
9       NV_REF            locator;
10      NV_INDEX          *nvIndex = NvGetIndexInfo(in->nvIndex, &locator);
11  // Input Validation
12      // This command can't be used to delete an index with TPMA_NV_POLICY_DELETE SET
13      if(IsNv_TPMA_NV_POLICY_DELETE(nvIndex->publicArea.attributes))
14          return TPM_RCS_ATTRIBUTES + RC_NV_UndefineSpace_nvIndex;
15      // The owner may only delete an index that was defined with ownerAuth. The
16      // platform may delete an index that was created with either authorization.
17      if(in->authHandle == TPM_RH_OWNER
18          && IsNv_TPMA_NV_PLATFORMCREATE(nvIndex->publicArea.attributes))
19          return TPM_RC_NV_AUTHORIZATION;
20  // Internal Data Update
21      // Call implementation dependent internal routine to delete NV index
22      return NvDeleteIndex(nvIndex, locator);
23  }
24  #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 209 — 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 210 — TPM2_NV_UndefineSpaceSpecial Response

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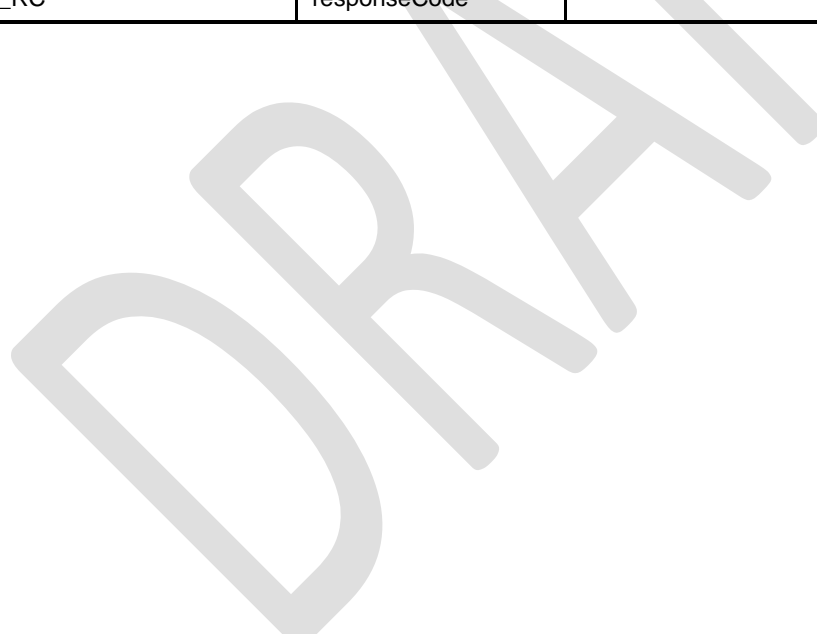
31.5.3   Detailed Actions

```
1    #include "Tpm.h"
2    #include "NV_UndefineSpaceSpecial_fp.h"
3    #include "SessionProcess_fp.h"
4    #ifdef TPM_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*

```
5    TPM_RC
6    TPM2_NV_UndefineSpaceSpecial(
7        NV_UndefineSpaceSpecial_In  *in            // IN: input parameter list
8        )
9    {
10       TPM_RC           result;
11       NV_REF           locator;
12       NV_INDEX         *nvIndex = NvGetIndexInfo(in->nvIndex, &locator);
13   // Input Validation
14       // This operation only applies when the TPMA_NV_POLICY_DELETE attribute is SET
15       if(!IsNv_TPMA_NV_POLICY_DELETE(nvIndex->publicArea.attributes))
16           return TPM_RCS_ATTRIBUTES + RC_NV_UndefineSpaceSpecial_nvIndex;
17   // Internal Data Update
18       // Call implementation dependent internal routine to delete NV index
19       result = NvDeleteIndex(nvIndex, locator);
20       // If we just removed the index providing the authorization, make sure that the
21       // authorization session computation is modified so that it doesn't try to
22       // access the authValue of the just deleted index
23       if(result == TPM_RC_SUCCESS)
24           SessionRemoveAssociationToHandle(in->nvIndex);
25       return result;
26   }
27   #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 211 — 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 212 — 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

```
1   #include "Tpm.h"
2   #include "NV_ReadPublic_fp.h"
3   #ifdef TPM_CC_NV_ReadPublic  // Conditional expansion of this file
4   TPM_RC
5   TPM2_NV_ReadPublic(
6       NV_ReadPublic_In    *in,            // IN: input parameter list
7       NV_ReadPublic_Out   *out            // OUT: output parameter list
8       )
9   {
10      NV_INDEX        *nvIndex = NvGetIndexInfo(in->nvIndex, NULL);
11  // Command Output
12      // Copy index public data to output
13      out->nvPublic.nvPublic = nvIndex->publicArea;
14      // Compute NV name
15      NvGetIndexName(nvIndex, &out->nvName);
16      return TPM_RC_SUCCESS;
17  }
18  #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 213 — 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 offset into the NV Area

Table 214 — TPM2_NV_Write Response

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

31.7.3 Detailed Actions

```
1   #include "Tpm.h"
2   #include "NV_Write_fp.h"
3   #ifdef TPM_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

```
4   TPM_RC
5   TPM2_NV_Write(
6       NV_Write_In     *in             // IN: input parameter list
7       )
8   {
9       NV_INDEX        *nvIndex = NvGetIndexInfo(in->nvIndex, NULL);
10      TPMA_NV          attributes = nvIndex->publicArea.attributes;
11      TPM_RC           result;
12  // Input Validation
13      // Common access checks, NvWriteAccessCheck() may return TPM_RC_NV_AUTHORIZATION
14      // or TPM_RC_NV_LOCKED
15      result = NvWriteAccessChecks(in->authHandle,
16                                  in->nvIndex,
17                                  attributes);
18      if(result != TPM_RC_SUCCESS)
19          return result;
20      // Bits index, extend index or counter index may not be updated by
21      // TPM2_NV_Write
22      if(IsNvCounterIndex(attributes)
23         || IsNvBitsIndex(attributes)
24         || IsNvExtendIndex(attributes))
25          return TPM_RC_ATTRIBUTES;
26      // Make sure that the offset is not too large
27      if(in->offset > nvIndex->publicArea.dataSize)
28          return TPM_RCS_VALUE + RC_NV_Write_offset;
29      // Make sure that the selection is within the range of the Index
30      if(in->data.t.size > (nvIndex->publicArea.dataSize - in->offset))
31          return TPM_RC_NV_RANGE;
32      // If this index requires a full sized write, make sure that input range is
33      // full sized.
34      // Note: if the requested size is the same as the Index data size, then offset
35      // will have to be zero. Otherwise, the range check above would have failed.
36      if(IsNv_TPMA_NV_WRITEALL(attributes)
37         && in->data.t.size < nvIndex->publicArea.dataSize)
38          return TPM_RC_NV_RANGE;
39  // Internal Data Update
40      // Perform the write.  This called routine will SET the TPMA_NV_WRITTEN
41      // attribute if it has not already been SET. If NV isn't available, an error
42      // will be returned.
43      return NvWriteIndexData(nvIndex, in->offset, in->data.t.size,
44                              in->data.t.buffer);
45  }
46  #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 215 — 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 216 — TPM2_NV_Increment Response

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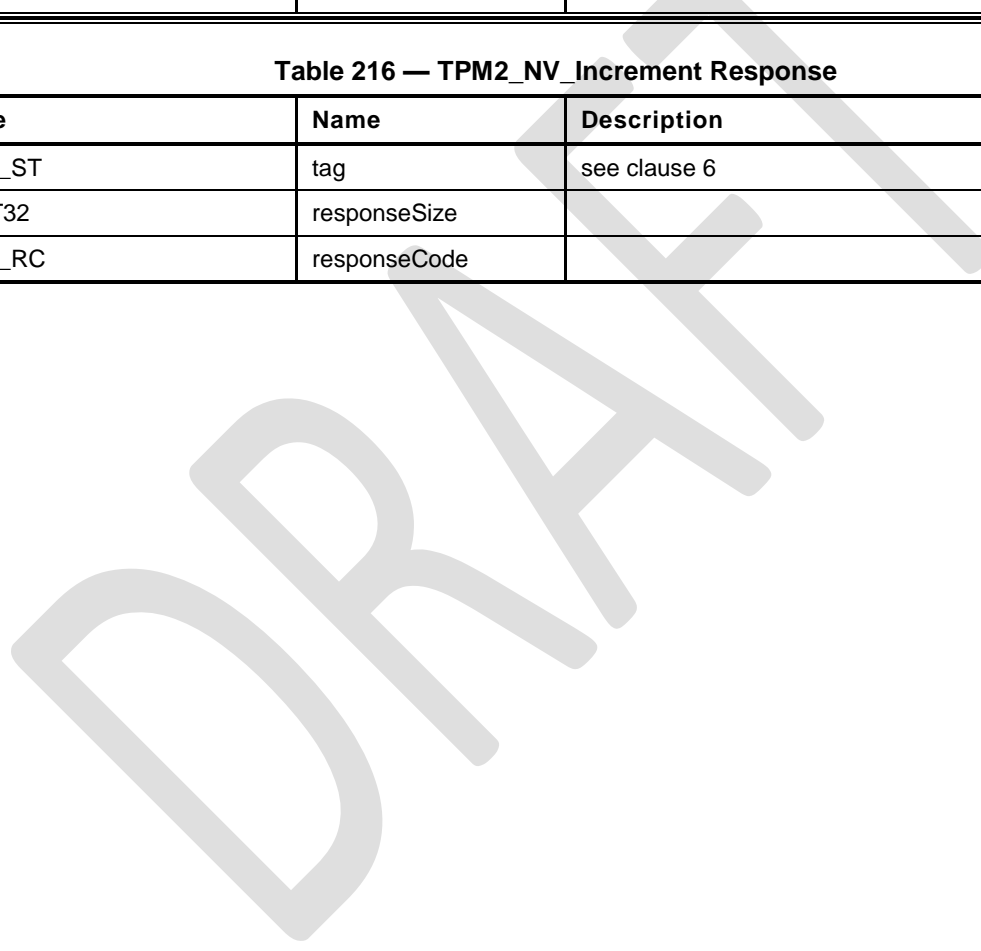
31.8.3   Detailed Actions

```
1    #include "Tpm.h"
2    #include "NV_Increment_fp.h"
3    #ifdef TPM_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

```
4    TPM_RC
5    TPM2_NV_Increment(
6        NV_Increment_In     *in                 // IN: input parameter list
7        )
8    {
9        TPM_RC            result;
10       NV_REF            locator;
11       NV_INDEX          *nvIndex = NvGetIndexInfo(in->nvIndex, &locator);
12       UINT64            countValue;
13   // Input Validation
14       // Common access checks, NvWriteAccessCheck() may return TPM_RC_NV_AUTHORIZATION
15       // or TPM_RC_NV_LOCKED
16       result = NvWriteAccessChecks(in->authHandle,
17                                    in->nvIndex,
18                                    nvIndex->publicArea.attributes);
19       if(result != TPM_RC_SUCCESS)
20           return result;
21       // Make sure that this is a counter
22       if(!IsNvCounterIndex(nvIndex->publicArea.attributes))
23           return TPM_RCS_ATTRIBUTES + RC_NV_Increment_nvIndex;
24   // Internal Data Update
25       // If counter index is not been written, initialize it
26       if(!IsNv_TPMA_NV_WRITTEN(nvIndex->publicArea.attributes))
27           countValue = NvReadMaxCount();
28       else
29           // Read NV data in native format for TPM CPU.
30           countValue = NvGetUINT64Data(nvIndex, locator);
31       // Do the increment
32       countValue++;
33       // Write NV data back. A TPM_RC_NV_UNAVAILABLE or TPM_RC_NV_RATE error may
34       // be returned at this point. If necessary, this function will set the
35       // TPMA_NV_WRITTEN attribute
36       result = NvWriteUINT64Data(nvIndex, countValue);
37       if(result == TPM_RC_SUCCESS)
38       {
39           // If a counter just rolled over, then force the NV update.
40           // Note, if this is an orderly counter, then the write-back needs to be
41           // forced, for other counters, the write-back will happen anyway
42           if(IsNv_TPMA_NV_ORDERLY(nvIndex->publicArea.attributes)
43              && (countValue & MAX_ORDERLY_COUNT) == 0 )
44           {
45               // Need to force an NV update of orderly data
46               SET_NV_UPDATE(UT_ORDERLY);
47           }
48       }
49       return result;
50   }
51   #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:

$$nvIndex{\rightarrow}data_{new} := \mathbf{H}_{nameAkg}(nvIndex{\rightarrow}data_{old} \,||\, data.buffer) \tag{41}$$

where

$\mathbf{H}_{nameAkg}()$ the hash algorithm indicated in *nvIndex→nameAlg*

nvIndex→data   the value of the data field in the NV Index

data.buffer   the data buffer of the command parameter

NOTE 3   If TPMA_NV_WRITTEN is CLEAR, then *nvIndex→data* is a Zero Digest.

31.9.2   Command and Response

Table 217 — 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 218 — TPM2_NV_Extend Response

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

31.9.3 Detailed Actions

```
1   #include "Tpm.h"
2   #include "NV_Extend_fp.h"
3   #ifdef TPM_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

```
4   TPM_RC
5   TPM2_NV_Extend(
6       NV_Extend_In    *in              // IN: input parameter list
7       )
8   {
9       TPM_RC                 result;
10      NV_REF                 locator;
11      NV_INDEX               *nvIndex = NvGetIndexInfo(in->nvIndex, &locator);
12      TPM2B_DIGEST           oldDigest;
13      TPM2B_DIGEST           newDigest;
14      HASH_STATE             hashState;
15  // Input Validation
16      // Common access checks, NvWriteAccessCheck() may return TPM_RC_NV_AUTHORIZATION
17      // or TPM_RC_NV_LOCKED
18      result = NvWriteAccessChecks(in->authHandle,
19                                   in->nvIndex,
20                                   nvIndex->publicArea.attributes);
21      if(result != TPM_RC_SUCCESS)
22          return result;
23      // Make sure that this is an extend index
24      if(!IsNvExtendIndex(nvIndex->publicArea.attributes))
25          return TPM_RCS_ATTRIBUTES + RC_NV_Extend_nvIndex;
26  // Internal Data Update
27      // Perform the write.
28      oldDigest.t.size = CryptHashGetDigestSize(nvIndex->publicArea.nameAlg);
29      pAssert(oldDigest.t.size <= sizeof(oldDigest.t.buffer));
30      if(IsNv_TPMA_NV_WRITTEN(nvIndex->publicArea.attributes))
31      {
32          NvGetIndexData(nvIndex, locator, 0, oldDigest.t.size, oldDigest.t.buffer);
33      }
34      else
35      {
36          MemorySet(oldDigest.t.buffer, 0, oldDigest.t.size);
37      }
38      // Start hash
39      newDigest.t.size = CryptHashStart(&hashState, nvIndex->publicArea.nameAlg);
40      // Adding old digest
41      CryptDigestUpdate2B(&hashState, &oldDigest.b);
42      // Adding new data
43      CryptDigestUpdate2B(&hashState, &in->data.b);
44      // Complete hash
45      CryptHashEnd2B(&hashState, &newDigest.b);
46      // Write extended hash back.
47      // Note, this routine will SET the TPMA_NV_WRITTEN attribute if necessary
48      return NvWriteIndexData(nvIndex, 0, newDigest.t.size, newDigest.t.buffer);
49  }
50  #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 219 — 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 220 — TPM2_NV_SetBits Response

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

31.10.3 Detailed Actions

```
1   #include "Tpm.h"
2   #include "NV_SetBits_fp.h"
3   #ifdef TPM_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

```
4    TPM_RC
5    TPM2_NV_SetBits(
6        NV_SetBits_In   *in                 // IN: input parameter list
7        )
8    {
9        TPM_RC             result;
10       NV_REF             locator;
11       NV_INDEX          *nvIndex = NvGetIndexInfo(in->nvIndex, &locator);
12       UINT64             oldValue;
13       UINT64             newValue;
14   // Input Validation
15       // Common access checks, NvWriteAccessCheck() may return TPM_RC_NV_AUTHORIZATION
16       // or TPM_RC_NV_LOCKED
17       result = NvWriteAccessChecks(in->authHandle,
18                                    in->nvIndex,
19                                    nvIndex->publicArea.attributes);
20       if(result != TPM_RC_SUCCESS)
21           return result;
22       // Make sure that this is a bit field
23       if(!IsNvBitsIndex(nvIndex->publicArea.attributes))
24           return TPM_RCS_ATTRIBUTES + RC_NV_SetBits_nvIndex;
25       // If index is not been written, initialize it
26       if(!IsNv_TPMA_NV_WRITTEN(nvIndex->publicArea.attributes))
27           oldValue = 0;
28       else
29           // Read index data
30           oldValue = NvGetUINT64Data(nvIndex, locator);
31       // Figure out what the new value is going to be
32       newValue = oldValue | in->bits;
33   // Internal Data Update
34       return  NvWriteUINT64Data(nvIndex, newValue);
35   }
36   #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) unless TPMA_NV_WRITEDEFINE is SET or if TPMA_NV_WRITTEN is CLEAR.

### 31.11.2 Command and Response

Table 221 — 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 222 — TPM2_NV_WriteLock Response

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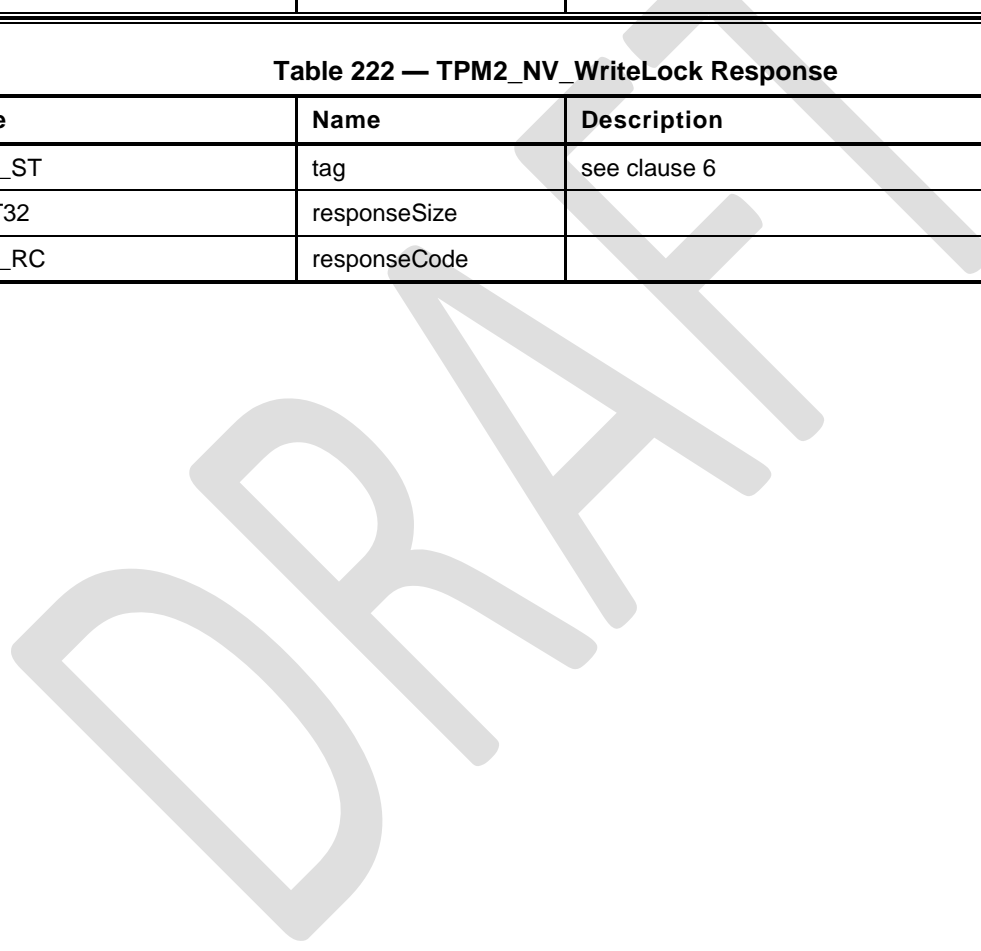
31.11.3 Detailed Actions

```
1    #include "Tpm.h"
2    #include "NV_WriteLock_fp.h"
3    #ifdef TPM_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*

```
4    TPM_RC
5    TPM2_NV_WriteLock(
6        NV_WriteLock_In    *in                // IN: input parameter list
7        )
8    {
9        TPM_RC            result;
10       NV_REF            locator;
11       NV_INDEX          *nvIndex = NvGetIndexInfo(in->nvIndex, &locator);
12       TPMA_NV           nvAttributes = nvIndex->publicArea.attributes;
13   // Input Validation:
14       // Common access checks, NvWriteAccessCheck() may return TPM_RC_NV_AUTHORIZATION
15       // or TPM_RC_NV_LOCKED
16       result = NvWriteAccessChecks(in->authHandle, in->nvIndex, nvAttributes);
17       if(result != TPM_RC_SUCCESS)
18       {
19           if(result == TPM_RC_NV_AUTHORIZATION)
20               return result;
21           // If write access failed because the index is already locked, then it is
22           // no error.
23           return TPM_RC_SUCCESS;
24       }
25       // if neither TPMA_NV_WRITEDEFINE nor TPMA_NV_WRITE_STCLEAR is set, the index
26       // can not be write-locked
27       if(nvAttributes.TPMA_NV_WRITEDEFINE == CLEAR
28          && nvAttributes.TPMA_NV_WRITE_STCLEAR == CLEAR)
29           return TPM_RCS_ATTRIBUTES + RC_NV_WriteLock_nvIndex;
30   // Internal Data Update
31       // Set the WRITELOCK attribute.
32       // Note: if TPMA_NV_WRITELOCKED were already SET, then the write access check
33       // above would have failed and this code isn't executed.
34       nvAttributes.TPMA_NV_WRITELOCKED = SET;
35       // Write index info back
36       return NvWriteIndexAttributes(nvIndex->publicArea.nvIndex, locator,
37                                     nvAttributes);
38   }
39   #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_WRITELOCKED 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 223 — TPM2_NV_GlobalWriteLock Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_NV_GlobalWriteLock
TPMI_RH_PROVISION	@authHandle	TPM_RH_OWNER or TPM_RH_PLATFORM+{PP} Auth Index: 1 Auth Role: USER

Table 224 — TPM2_NV_GlobalWriteLock Response

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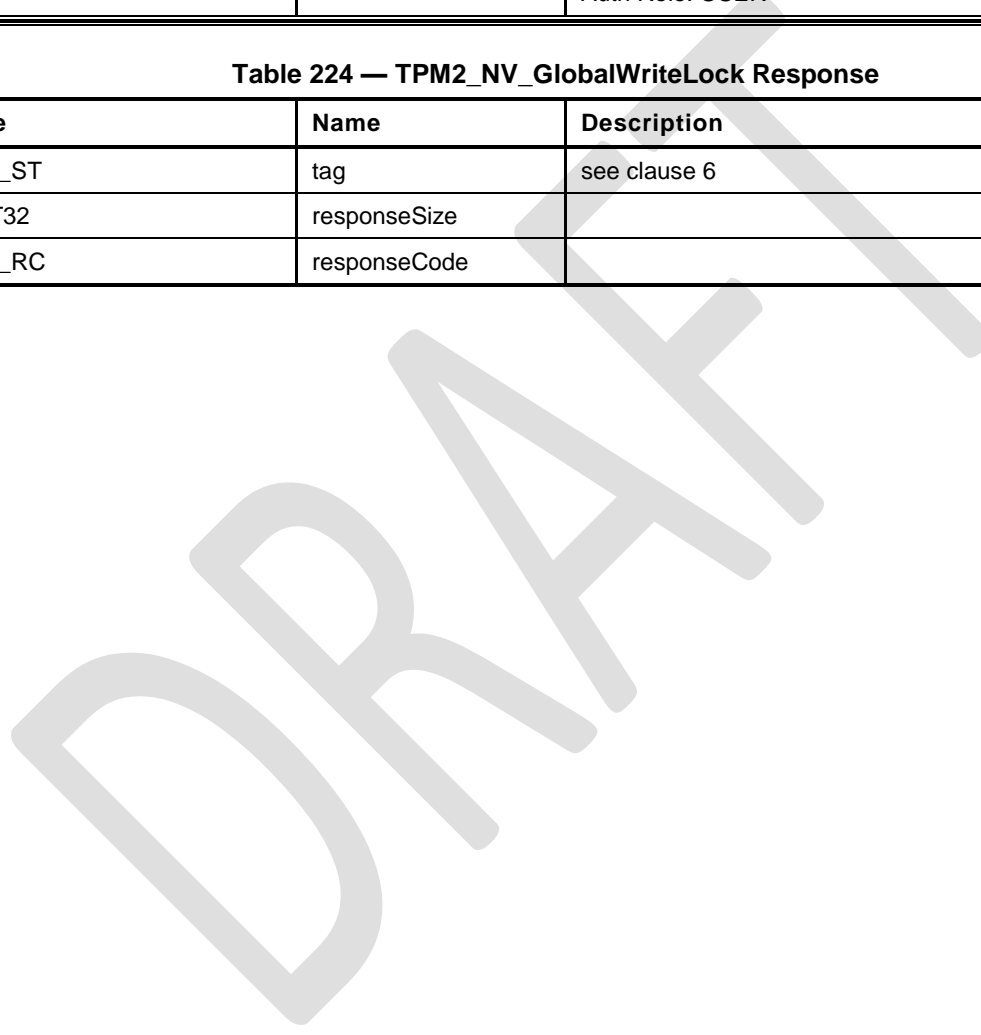
31.12.3 Detailed Actions

```
1    #include "Tpm.h"
2    #include "NV_GlobalWriteLock_fp.h"
3    #ifdef TPM_CC_NV_GlobalWriteLock  // Conditional expansion of this file
4    TPM_RC
5    TPM2_NV_GlobalWriteLock(
6        NV_GlobalWriteLock_In   *in             // IN: input parameter list
7        )
8    {
9        // Input parameter (the authorization handle) is not reference in command action.
10       NOT_REFERENCED(in);
11   // Internal Data Update
12       // Implementation dependent method of setting the global lock
13       return NvSetGlobalLock();
14   }
15   #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, *offset* is ignored and the data is read starting at offset 0.

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 225 — 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 area This value shall be less than or equal to the size of the *nvIndex* data.

Table 226 — 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

```
1    #include "Tpm.h"
2    #include "NV_Read_fp.h"
3    #ifdef TPM_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_NV_VALUE	the read size is larger than the MAX_NV_BUFFER_SIZE

```
4    TPM_RC
5    TPM2_NV_Read(
6        NV_Read_In      *in,             // IN: input parameter list
7        NV_Read_Out     *out             // OUT: output parameter list
8        )
9    {
10       NV_REF          locator;
11       NV_INDEX        *nvIndex = NvGetIndexInfo(in->nvIndex, &locator);
12       TPM_RC          result;
13   // Input Validation
14       // Common read access checks. NvReadAccessChecks() may return
15       // TPM_RC_NV_AUTHORIZATION, TPM_RC_NV_LOCKED, or TPM_RC_NV_UNINITIALIZED
16       result = NvReadAccessChecks(in->authHandle, in->nvIndex,
17                                   nvIndex->publicArea.attributes);
18       if(result != TPM_RC_SUCCESS)
19           return result;
20       // Make sure the data will fit the return buffer
21       if(in->size > MAX_NV_BUFFER_SIZE)
22           return TPM_RCS_VALUE + RC_NV_Read_size;
23       // Verify that the offset is not too large
24       if(in->offset > nvIndex->publicArea.dataSize)
25           return TPM_RCS_VALUE + RC_NV_Read_offset;
26       // Make sure that the selection is within the range of the Index
27       if(in->size > (nvIndex->publicArea.dataSize - in->offset))
28           return TPM_RC_NV_RANGE;
29   // Command Output
30       // Set the return size
31       out->data.t.size = in->size;
32       // Perform the read
33       NvGetIndexData(nvIndex, locator, in->offset, in->size, out->data.t.buffer);
34       return TPM_RC_SUCCESS;
35   }
36   #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 227 — TPM2_NV_ReadLock Command

Type	Name	Description
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS
UINT32	commandSize	
TPM_CC	commandCode	TPM_CC_NV_ReadLock
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 228 — TPM2_NV_ReadLock Response

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

31.14.3 Detailed Actions

```
1    #include "Tpm.h"
2    #include "NV_ReadLock_fp.h"
3    #ifdef TPM_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*

```
4    TPM_RC
5    TPM2_NV_ReadLock(
6        NV_ReadLock_In  *in                  // IN: input parameter list
7        )
8    {
9        TPM_RC            result;
10       NV_REF            locator;
11       // The referenced index has been checked multiple times before this is called
12       // so it must be present and will be loaded into cache
13       NV_INDEX          *nvIndex = NvGetIndexInfo(in->nvIndex, &locator);
14       TPMA_NV           nvAttributes = nvIndex->publicArea.attributes;
15   // Input Validation
16       // Common read access checks. NvReadAccessChecks() may return
17       // TPM_RC_NV_AUTHORIZATION, TPM_RC_NV_LOCKED, or TPM_RC_NV_UNINITIALIZED
18       result = NvReadAccessChecks(in->authHandle,
19                                   in->nvIndex,
20                                   nvAttributes);
21       if(result == TPM_RC_NV_AUTHORIZATION)
22           return TPM_RC_NV_AUTHORIZATION;
23       // Index is already locked for write
24       else if(result == TPM_RC_NV_LOCKED)
25             return TPM_RC_SUCCESS;
26       // If NvReadAccessChecks return TPM_RC_NV_UNINITALIZED, then continue.
27       // It is not an error to read lock an uninitialized Index.
28       // if TPMA_NV_READ_STCLEAR is not set, the index can not be read-locked
29       if(nvAttributes.TPMA_NV_READ_STCLEAR == CLEAR)
30           return TPM_RCS_ATTRIBUTES + RC_NV_ReadLock_nvIndex;
31   // Internal Data Update
32       // Set the READLOCK attribute
33       nvAttributes.TPMA_NV_READLOCKED = SET;
34       // Write NV info back
35       return NvWriteIndexAttributes(nvIndex->publicArea.nvIndex,
36                                     locator,
37                                     nvAttributes);
38   }
39   #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 authorization indicated when the NV Index was defined.

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 229 — 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 230 — TPM2_NV_ChangeAuth Response

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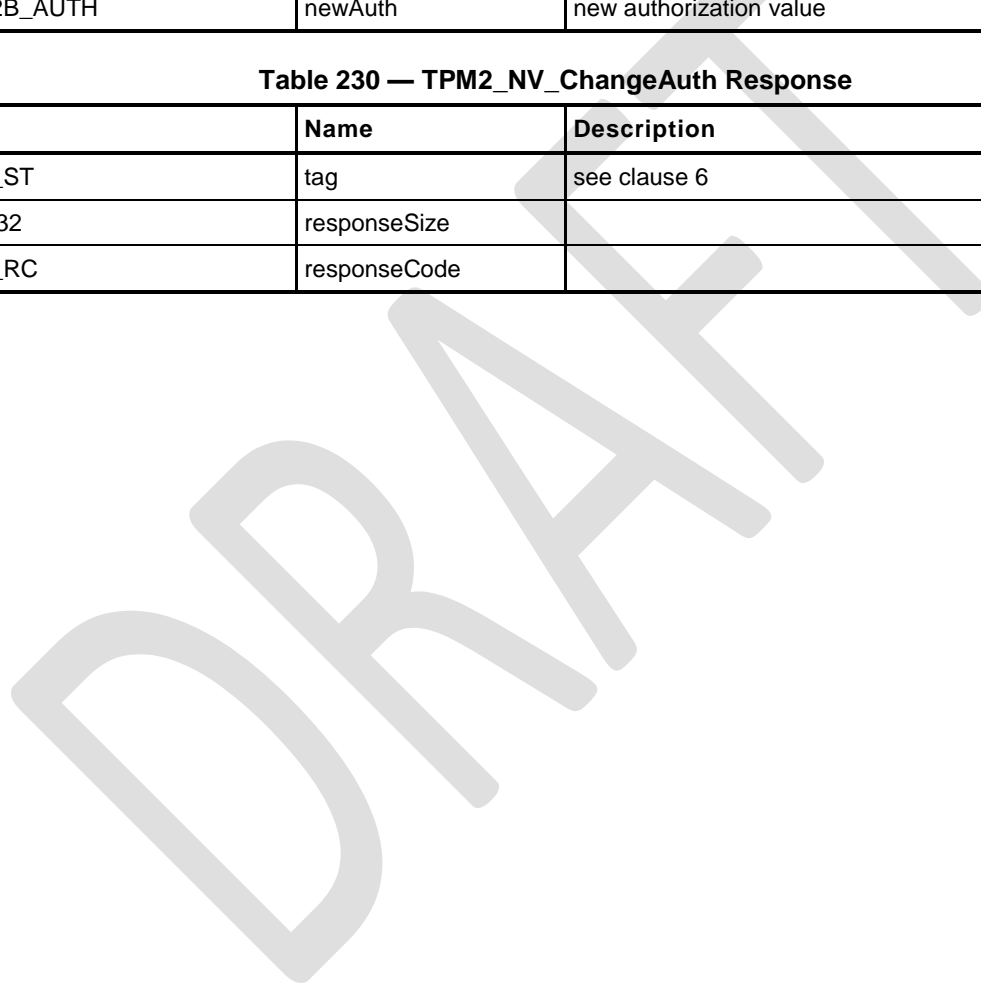
31.15.3 Detailed Actions

```
1    #include "Tpm.h"
2    #include "NV_ChangeAuth_fp.h"
3    #ifdef TPM_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*

```
4    TPM_RC
5    TPM2_NV_ChangeAuth(
6        NV_ChangeAuth_In    *in              // IN: input parameter list
7        )
8    {
9        NV_REF          locator;
10       NV_INDEX        *nvIndex = NvGetIndexInfo(in->nvIndex, &locator);
11   // Input Validation
12       // Remove trailing zeros and make sure that the result is not larger than the
13       // digest of the nameAlg.
14       if(MemoryRemoveTrailingZeros(&in->newAuth)
15          > CryptHashGetDigestSize(nvIndex->publicArea.nameAlg))
16           return TPM_RCS_SIZE + RC_NV_ChangeAuth_newAuth;
17   // Internal Data Update
18       // Change authValue
19       return NvWriteIndexAuth(locator, &in->newAuth);
20   }
21   #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, *offset* is ignored and an offset of 0 is used.

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.

31.16.2 Command and Response

Table 231 — 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 area This value shall be less than or equal to the size of the *nvIndex* data.

Table 232 — 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

```
1    #include "Tpm.h"
2    #include "Attest_spt_fp.h"
3    #include "NV_Certify_fp.h"
4    #ifdef TPM_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

```
5    TPM_RC
6    TPM2_NV_Certify(
7        NV_Certify_In    *in,              // IN: input parameter list
8        NV_Certify_Out   *out              // OUT: output parameter list
9        )
10   {
11       TPM_RC                    result;
12       NV_REF                     locator;
13       NV_INDEX                  *nvIndex = NvGetIndexInfo(in->nvIndex, &locator);
14       TPMS_ATTEST               certifyInfo;
15       OBJECT                    *signObject = HandleToObject(in->signHandle);
16   // Input Validation
17       if(!IsSigningObject(signObject))
18           return TPM_RCS_KEY + RC_NV_Certify_signHandle;
19       if(!CryptSelectSignScheme(signObject, &in->inScheme))
20           return TPM_RC_SCHEME + RC_NV_Certify_inScheme;
21       // Common access checks, NvWriteAccessCheck() may return TPM_RC_NV_AUTHORIZATION
22       // or TPM_RC_NV_LOCKED
23       result = NvReadAccessChecks(in->authHandle, in->nvIndex,
24                                   nvIndex->publicArea.attributes);
25       if(result != TPM_RC_SUCCESS)
26           return result;
27       // make sure that the selection is within the range of the Index (cast to avoid
28       // any wrap issues with addition)
29       if((UINT32)in->size + (UINT32)in->offset > (UINT32)nvIndex->publicArea.dataSize)
30           return TPM_RC_NV_RANGE;
31   // Command Output
32       // Fill in attest information common fields
33       FillInAttestInfo(in->signHandle, &in->inScheme, &in->qualifyingData,
34                       &certifyInfo);
35       // NV certify specific fields
36       // Attestation type
37       certifyInfo.type = TPM_ST_ATTEST_NV;
38       // Get the name of the index
39       NvGetIndexName(nvIndex, &certifyInfo.attested.nv.indexName);
40       // Set the return size
41       certifyInfo.attested.nv.nvContents.t.size = in->size;
42       // Set the offset
43       certifyInfo.attested.nv.offset = in->offset;
44       // Perform the read
45       NvGetIndexData(nvIndex, locator, in->offset, in->size,
46                     certifyInfo.attested.nv.nvContents.t.buffer);
47       // Sign attestation structure.  A NULL signature will be returned if
```

Family "2.0"

TCG Public Review

Page 439

Level 00 Revision 01.36

Copyright © TCG 2006-2016

September 16, 2016

```
48        // signObject is NULL.
49        return SignAttestInfo(signObject, &in->inScheme, &certifyInfo,
50                              &in->qualifyingData, &out->certifyInfo, &out->signature);
51    }
52    #endif // CC_NV_Certify
```

32 Vendor Specific

32.1 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 insure the proper operation of the command dispatch code when processing a vendor-specific command.

32.2 TPM2_Vendor_TCG_Test

32.2.1 General Description

This is a placeholder to allow testing of the dispatch code.

32.2.2 Command and Response

Table 233 — 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 234 — 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

32.2.3   Detailed Actions

```
1   #include "Tpm.h"
2   #include "Vendor_TCG_Test_fp.h"
3   #include "Marshal_fp.h"
4   #if defined TPM_CC_Vendor_TCG_Test       // Conditional expansion of this file
```

A dummy function for testing.

```
5   TPM_RC
6   TPM2_Vendor_TCG_Test(
7       Vendor_TCG_Test_In          *in,        // IN: input parameter list
8       Vendor_TCG_Test_Out         *out        // OUT: output parameter list
9       )
10  {
11      out->outputData = in->inputData;
12      return TPM_RC_SUCCESS;
13  }
14  #endif // CC_Vendor_TCG_Test
```