# TPM Main
# Part 1 Design Principles

**Specification Version 1.2**
**Revision 116**
**1 March 2011**
**TCG Published**

Contact: admin@trustedcomputinggroup.com

## TCG Published

## Disclaimers, Notices, and License Terms

THIS SPECIFICATION IS PROVIDED "AS IS" WITH NO WARRANTIES WHATSOEVER, INCLUDING ANY WARRANTY OF MERCHANTABILITY, NONINFRINGEMENT, FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION OR SAMPLE.

Without limitation, TCG disclaims 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.

Contact the Trusted Computing Group at http://www.trustedcomputinggroup.org/">www.trustedcomputinggroup.org for information on specification licensing through membership agreements.

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

# Change History

| Version | Date | Description |
|---|---|---|
| Rev 50 | Jun 2003 | Started 30 Jun 2003 by David Grawrock<br>First cut at the design principles |
| Rev 52 | Jul 2003 | Started 15 Jul 2003 by David Grawrock<br>Moved |
| Rev 58 | Aug 2003 | Started 27 Aug 2003 by David Grawrock<br>All emails through 28 August 2003<br>New delegation from Graeme merged |
| Rev 62 | Oct 2003 | Approved by WG, TC and Board as public release of 1.2 |
| Rev 63 | Oct 2003 | Started 2 Oct 2003 by David Grawrock<br>Kerry email 7 Oct “Various items in rev62”<br>kerry email 10 Oct “Other issues in rev 62”<br>Changes to audit generation |
| Rev 64 | Oct 2003 | Started 12 Oct 2003 by David Grawrock<br>Removed PCRWRITE usage in the NV write commands<br>Added locality to transport_out log<br>Disable readpubek now set in takeownership. DisableReadpubek now deprecated, as the functionality is moot.<br>Oshrats email regarding DSAP/OSAP sessions and the invalidation of them on delegation changes<br>Changes for CMK commands.<br>Oshrats email with minor 63 comments |
| Rev 65 | Nov 2003 | Action in NV_DefineSpace to ignore the Booleans in the input structure (Kerry email of 10/30<br>Transport changes from markus 11/6 email<br>Set rules for encryption of parameters for OIAP,OSAP and DSAP<br>Rewrote section on debug PCR to specify that the platform spec must indicate which register is the debug PCR<br>Orlando FtF decisions<br>CMK changes from Graeme |
| Rev 66 | Nov 2003 | Comment that OSAP tied to owner delegation needs to be treated internally in the TPM as a DSAP session<br>Minor edits from Monty<br>Added new GetCapability as requested by PC Specific WG<br>Added new DP section that shows mandatory and optional<br>Oshrat email of 11/27<br>Change PCR attributes to use locality selection instead of an array of BOOL’s<br>Removed transport sessions as something to invalidate when a resource type is flushed.<br>Oshrat email of 12/3<br>added checks for NV_Locked in the NV commands<br>Additional emails from the WG for minor editing fixes |
| Rev 67 | Dec 2003 | Made locality_modifier always a 1 size<br>Changed NV index values to add the reserved bit. Also noticed that the previous NV index values were 10 bytes not 8. Edited them to correct size.<br>Audit changes to ensure audit listed as optional and the previous commands properly deleted<br>Added new OSAP authorization encryption. Changes made with new entity types, new section in DP (bottom of doc) and all command rewritten to check for the new encryption |
| Rev 68 | Jan 2004 | Added new section to identify all changes made for FIPS. Made some FIPS changes on creating and loading of keys<br>Added change that OSAP encryption IV creation always uses both odd and even nonces<br>Added SEALX ordinal and changes to TPM_STORED_DATA12 and seal/unseal to support this |

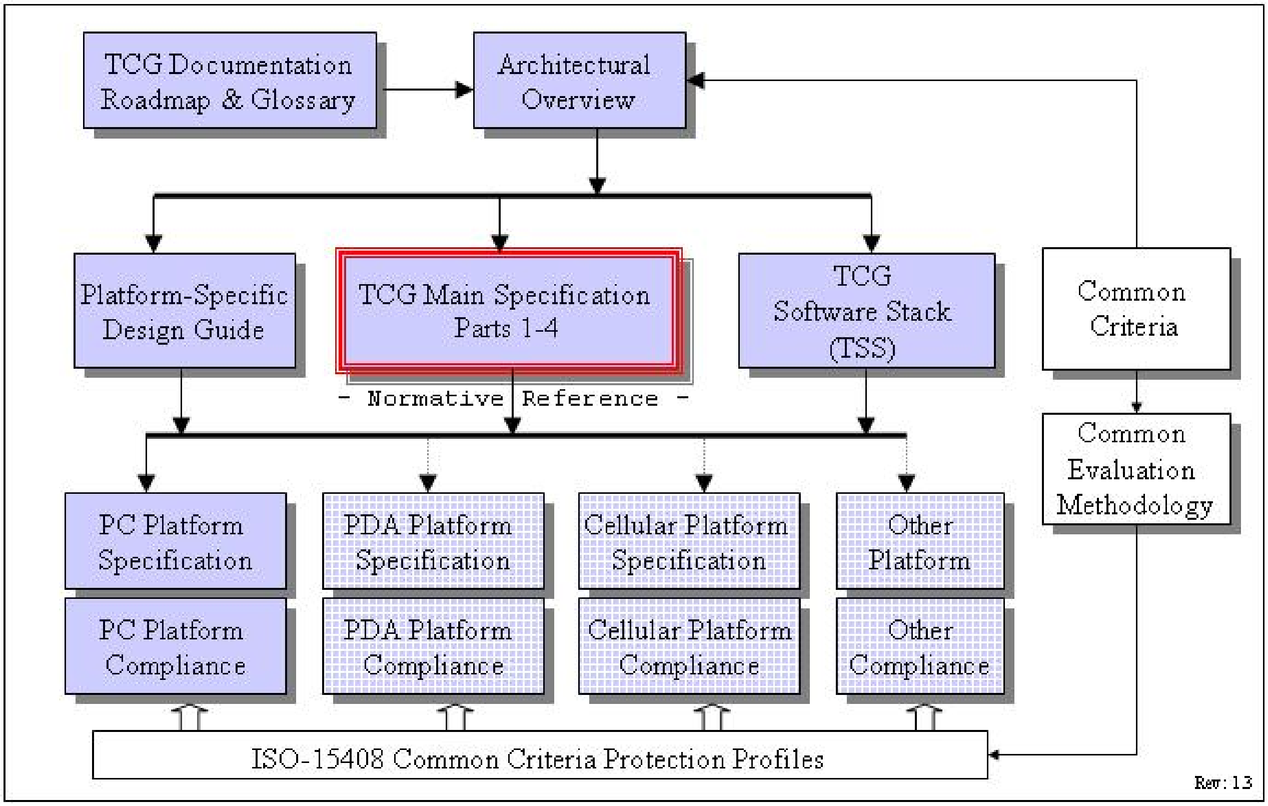
TCG Published

| | | |
|---|---|---|
| Rev 69 | Feb 2004 | Fixup on stored_data12.<br>Removed magic4 from the GPIO<br>Added in section 34 of DP further discussion of versioning and getcap<br>DP todo section cleaned up<br>Changed store_privkey in migrate_asymkey<br>Moved text for getcapabilities – hopefully it is easier to read and follow through on now. |
| Rev 70 | Mar 2004 | Rewrite structure doc on PCR selection usage.<br>New getcap to answer questions regarding TPM support for pcr selection size |
| Rev 71 | Mar 2004 | Change terms from authorization data to AuthData. |
| Rev 72 | Mar 2004 | Zimmermann's changes for DAA<br>Added TPM_Quote2, this includes new structure and ordinal<br>Updated key usage table to include the 1.2 commands<br>Added security properties section that links the main spec to the conformance WG guidelines (in section 1) |
| Rev 73 | Apr 2004 | Changed CMK_MigrateKey to use TPM_KEY12 and removed two input parameters<br>Allowed TPM_Getcapability and TPM_GetTestResult to execute prior to TPM_Startup when in failure mode |
| Rev 74 | May 2004 | Minor editing to reflect comments on web site.<br>Locked spec and submitted for IP review |
| Rev 76 | Aug 2004 | All comments from the WG<br>Included new SetValue command and all of the indexes to make that work |
| Rev 77 | Aug 2004 | All comments from the WG |
| Rev 78 | Oct 2004 | Comments from WG. Added new getcaps to report and query current TPM version |
| Rev 82 | Jan 2005 | All changes from emails and minutes (I think). |
| Rev 84 | Feb 2005 | Final changes for 1.2 level 2 |
| Rev 88 | Aug 2005 | Eratta level 2 release candidate |
| Rev 91 | Sept. 2005 | Update to Figure 9 (b) in section 9.2 by Tasneem Brutch |
| Rev 100 | May 2006 | Clarified CTR mode |
| Rev 101 | Aug 2006 | Added deactivated rationale. Clarified number of sessions. Changed "set to NULL" to "set to zero". Added NV index D bit rationale. Added _INFO key rationale and clarified cases where _INFO keys act as _SHA1 keys. |
| Rev 102 | Sept 2006 | Minor typos only. No functional changes. |
| Rev 103 | Oct 2006 | Note that blobs encrypted in blocks must have integrity chaining. Merged two AIK sections. Self-test checks EK using encryption, not signing. |
| Rev 104 | Nov 2006 | RNG must be NIST approved in FIPS mode. When ordinal says OSAP for ADIP, it means OSAP or DSAP. |
| Rev 105 | Feb 2007 | Removed informative reference to getting signed counter values. |
| Rev 106 | April 2007 | Updated TPM_TakeOwnership state diagram and text to remove deactivated. Indicated that _INFO ordinals are examples, not a complete list. |
| Rev 107 | July 2007 | Explained that PCRs are still extended when disabled and deactivated. Removed restriction that TPM_CreateEndorsementKeyPair is not allowed after TPM_CreateRevokableEK or TPM_RevokeTrust. |
| Rev 108 | Aug 2007 | Field upgrade should not affect shielded locations. TPM MUST support two key slots. |
| Rev 109 | Oct 2007 | Cleaned up Opt-in physical presence wording. Changed some physical presence terms to agree with Part 2. |
| Rev 110 | May 2008 | TPM_AUTH_PRIV_USE_ONLY name change and indication that it refers to reading the public key. Warning that ADIP with a well known secret may require transport. |
| Rev 111 | July 2008 | TSC_ ordinals must be tested early. AddedMUST to self tests, but indicated that test methods are examples. Key contexts cannot be flushed. Deleted unbind payload normative to agreee with part 3. |
| Rev 112 | Jan 2009 | Default exponent clarified, self-test failure cleared by TPM_Init, self-test failure must delete saved state |
| Rev 113 | Jan 2009 | No changes. |
| Rev 114 | Jan 2009 | No changes |

| | | |
|---|---|---|
| Rev 116 | Aug 2009 | Audit happens at the end of the command.  Audit occurs when deactivated to remove a corner case. |

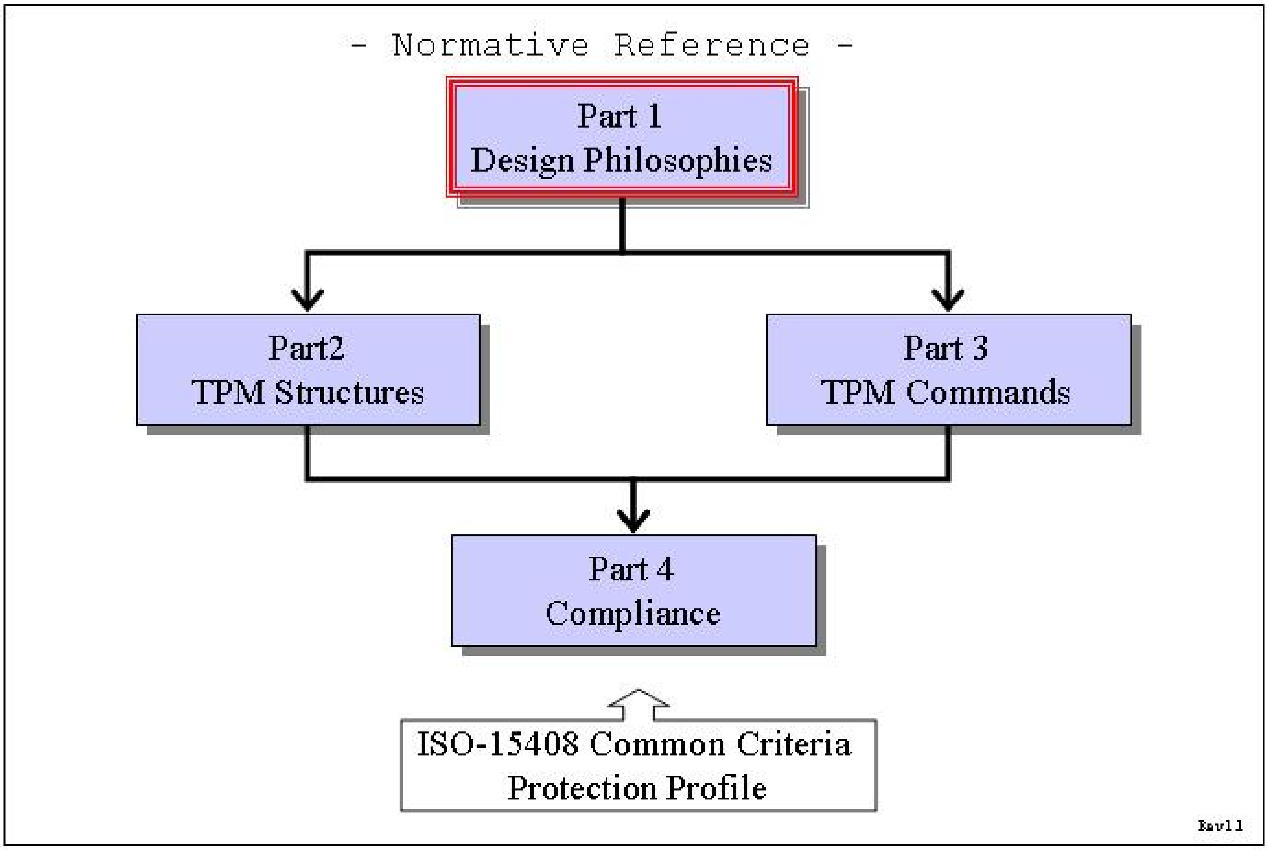
TCG Published

# TCG Doc Roadmap – Main Spec


TCG Documentation Roadmap & Glossary
Architectural Overview
Platform-Specific Design Guide
TCG Main Specification Parts 1-4
TCG Software Stack (TSS)
Common Criteria
- Normative Reference -
Common Evaluation Methodology
PC Platform Specification
PDA Platform Specification
Cellular Platform Specification
Other Platform
PC Platform Compliance
PDA Platform Compliance
Cellular Platform Compliance
Other Compliance
ISO-15408 Common Criteria Protection Profiles
Rev: 13

60

# TCG Main Spec Roadmap



- Normative Reference -

Part 1
Design Philosophies

Part2
TPM Structures

Part 3
TPM Commands

Part 4
Compliance

ISO-15408 Common Criteria
Protection Profile

TCG Published

# Table of Contents

**TCG Published**

TCG Published

# 1. Scope and Audience

The TPM main specification is an industry specification that enables trust in computing platforms in general. The main specification is broken into parts to make the role of each document clear. A version of the specification (like 1.2) requires all parts to be a complete specification.

A TPM designer MUST be aware that for a complete definition of all requirements necessary to build a TPM, the designer MUST use the appropriate platform specific specification for all TPM requirements.

## 1.1 Key words

The key words “MUST,” “MUST NOT,” “REQUIRED,” “SHALL,” “SHALL NOT,” “SHOULD,” “SHOULD NOT,” “RECOMMENDED,” “MAY,” and “OPTIONAL” in the chapters 2-10 normative statements are to be interpreted as described in [RFC-2119].

## 1.2 Statement Type

Please note a very important distinction between different sections of text throughout this document. You will encounter two distinctive kinds of text: informative comment and normative statements. Because most of the text in this specification will be of the kind normative statements, the authors have informally defined it as the default and, as such, have specifically called out text of the kind informative comment They have done this by flagging the beginning and end of each informative comment and highlighting its text in gray. This means that unless text is specifically marked as of the kind informative comment, you can consider it of the kind normative statements.

For example:

**Start of informative comment**

This is the first paragraph of several paragraphs containing text of the kind *informative comment …*

This is the second paragraph of text of the kind *informative comment …*

This is the nth paragraph of text of the kind *informative comment …*

To understand the TCG specification the user must read the specification. (This use of MUST does not require any action).

**End of informative comment**

This is the first paragraph of one or more paragraphs (and/or sections) containing the text of the kind normative statements …

To understand the TCG specification the user MUST read the specification. (This use of MUST indicates a keyword usage and requires an action).

# 2. Description

The design principles give the basic concepts of the TPM and generic information relative to TPM functionality.

A TPM designer MUST review and implement the information in the TPM Main specification (parts 1-4) and review the platform specific document for the intended platform. The platform specific document will contain normative statements that affect the design and implementation of a TPM.

A TPM designer MUST review and implement the requirements, including testing and evaluation, as set by the TCG Conformance Workgroup. The TPM MUST comply with the requirements and pass any evaluations set by the Conformance Workgroup. The TPM MAY undergo more stringent testing and evaluation.

The question section keeps track of questions throughout the development of the specification and hence can have information that is no longer current or moot. The purpose of the questions is to track the history of various decisions in the specification to allow those following behind to gain some insight into the committees thinking on various points.

## 2.1 TODO (notes to keep the editor on track)

## 2.2 Questions

**Start of informative comment**

How to version the flag structures?

> I suggest that we simply put the version into the structure and pass it back in the structure. Add the version information into the persistent and volatile flag structures.

When using the encryption transport failures are easy to see. Also the watcher on the line can tell where the error occurred. If the failure occurs at the transport level the response is an error (small packet) and it is in the clear. If the error occurs during execution of the command then the response is a small encrypted packet. Should we expand the packet size or simply let this go through?

> Not an issue.

Do we restrict the loading of a counter to once per TPM_Startup(Clear)?

> Yes once a counter is set it must remain the same until the next successful startup.

Does the time stamp work as a change on the tag or as a wrapped command like the transport protection.

> While possibly easier at the HW level the tag mechanism seems to be harder at the SW level as to what commands are sent to the TPM. The issue of how the SW presents the TS session to the SW writer is not an issue. This is due to the fact that however the session is presented to the SW writer the writer must take into account which commands are being time stamped and how to manage the log etc. So accepting a mechanism that is easy for the HW developer and having the SW manage the interface is a sufficient direction.

When returning time information do we return the entire time structure or just the time and have the caller obtain all the information with a GetCap call?

All time returns will use the entire structure with all the details.

Do we want to return a real clock value or a value with some additional bits (like a monotonic value with a time value)?

Add a count value into the time structure.

Do we need NTP or is SNTP sufficient?

The TPM will not run the time protocol itself. What the TPM will do is accept a value from outside software and a hash of the protocols that produced the value. This allows the platform to use whatever they want to set the value from secure time to the local PC clock.

Can an owner destroy a TPM by issuing repeated CreateCounter commands?

A TPM may place a throttle on this command to avoid burn issues. It MUST not be possible to burn out the TPM counter under normal operating conditions. The CreateCounter command is limited to only once per successful TPM_Startup(ST_CLEAR).

This answer is now somewhat moot as the command to createcounter is now owner authorized. This allows the owner to decide when to authorize the counter creation. As there are only 4 counters available it is not an issue with having the owner continue to authorize counters.

What happens to a transport session (log etc.) on an S3?

Should these be the same as the authorization sessions? The saving of a transport session across S3 is not a security concern but is a memory concern. The TPM MUST clear the transport session on TPM_Startup(CLEAR) and MAY clear the session on TPM_Startup(any).

While you can’t increment or create a new counter after startup can you read a counter other than the active one?

You may read other counters

When we audit a command that is not authorized should we hash the parameters and provide that as part of the audit event, currently they are set to null.

We should hash parameters of non-authorized commands

There is a fundamental problem with the encryption of commands in the transport and auditing. If we cover a command we have no way to audit, if we show the command then it isn’t protected. Can we expose the command (ordinal) and not the parameters?

If the owner has requested that a function be audited then the execute transport return will include sufficient information to produce the audit entry.

How to set the time in the audit structure and tell the log what is going on.

The time in the audit structure is set to nulls except when audit occurs as part of a transport session. In that case the audit command is set from the time value in the TPM.

Is there a limit to the number of locality modifiers?

Yes, the TPM need only support a maximum of 4 modifiers. The definition of the modifiers is always a platform specific issue.

How do we evict various resources?

There are numerous eviction routines in the current spec. We will deprecate the various types and move to TPM_Flushxxx for all resource types.

Can you flush a saved context?

Yes, you must be able to invalidate saved contexts. This would be done by making sure that the TPM could not load any saved context.

What is the value of maintaining the clock value when the time is not incrementing? Can this be due to the fact that the time is now known to be at least after the indicated time?

Moot point now as we don't keep the clock value at

Should we change the current structures and add the tag?

TODO

Can we have a bank of bits (change bit locality) for each of the 4 levels of locality?

Now

How do we find out what sessions are active? Do we care?

I would say yes we care and we should use the same mechanism that we do for the keys. A GetCap that will return the handles.

Can we limit the transport sessions to only one?

No, we should have as a minimum 2 sessions. One gets into deadlocks and such so the minimum should be 2.

Changed: The deadlock is with authorization sessions, not transport.

Does the TPM need to keep the audit structure or can it simply keep a hash?

The TPM just keeps the audit digest and no other information.

What happens to an OSAP session if the key associated with it is taken off chip with a "SaveContext"? What happens if the key saveContext occurs after an OSAP auth context that is already off chip? How do you later connect the key to the auth session (without having to store all sorts of things on chip)? Are we really honestly convinced that we've thought of all the possible ramifications of saving and restoring auth sessions? And is it really true that all the things we say about a saved auth session do/should apply to a saved key (which is to say is there really a single loadContext command and a single context structure)?

Saved context a reliable indication of the linkage between the OSAP and the key. When saving save auth then key, on load key then auth. Auth session checks for the key and if not found fails.

Why is addNonce an output of 16.5 loadContext?

If it's wrong, it's a little late to find out now - why not have it as an input and have the TPM return an error if the encrypted addNonce doesn't match the input? The thought was that the nonce area might not be a nonce but was information that the caller could put in. If they use it as a nonce fine, but they could also use it as a label or sequence number or … any value the caller wanted

Is there a memory endurance problem with contextNonceSession?

contextNonceSession does not have to be saved across S3 states so there is no endurance problem.

Is there a memory endurance problem with contextNonceKey?

contextNonceKey only changes on TPM_Startup(ST_Clear) so it's endurance is the same as a PCR.

The debate continues about restoring a resource's handle during TPM_LoadContext.

Debate ends by having the load context be informed of what the loaders opinion is about the handle. The requestor can indicate that it wishes the same handle and if the TPM can perform that task it does, if it cannot then the load fails.

Interesting attack is now available with the new audit close flag on get audit signed. Anyone with access to a signing key can close the audit log. The only requirement on the command is that the key be authorized. While there is no loss of information (as the attacker can always destroy the external log) does the closing of a log make things look different. This does enable a burn out attack. The ability to closeAudit enables a new DenialOfService attack.

Resolution: The TPM Owner owns the audit process, so the TPM Owner should have exclusive control over closeAudit. Hence the signing key used to closeAudit must be an AIK. Note that the owner can choose to give this AIK's AuthData value to the OS, so that the OS can automatically close an audit session during platform power down. But such operations are outside this specification.

Should we keep the E function in the tick counter?

From Graeme, I would prefer to see these calculations deleted. The calculation starts with one assertion and derives a contradictory assertion. Generally, there seems little value in trying to derive an equality relationship when nothing is known about the path to and from the Time Authority.

What is the difference between DIR_Quote and DirReadSigned?

Appears to be none so DIR_Quote deleted

The tickRate parameter associates tick with seconds and has no way to indicate that the rate is greater than one second. Is this OK?

Do we need to allow for tick rates that are slower than once per second. We report in nanoseconds.

The TPM MUST support a minimum of 2 authorization sessions. Where do we put this requirement in the spec?

Can we find a use for the DIR and BIT areas for locality 0?

They have no protections so in many ways they are just extra. We leave this as it is as locality 0 may mean something else on a platform other than a PC.

How do we send back the transport log information on each execute transport?

It is 64 byes in length and would make things very difficult to include on every command. Change wrappedaudit to be input params, add output parms and the caller has all information necessary to create the structure to add into the digest.

The transport log structure is a single structure used both for input and output with the only difference being the setting of ticks to 0 on input and a real value on output, do we need two structures.

I believe that a single structure is fine

For TPM_Startup(ST_Clear) I added that all keys would be flushed. Is this right?

Yes

Why have 2 auths for release transport signed? It is an easy attack to simply kill the session.

The reason is that an attacker can close the session and get a signature of the session log. We are currently not sure of the level of this attack but by having the creator of the session authorize the signing of the log it is completely avoided.

19.3 Action 3 (startup/state) doesn't reference the situation where there is no saved state. My presumption is that you can still run startup/clear, but maybe you have to do a hardware reset?

DWG I don't think so. This could be an attack and a way to get the wrong PCR values into the system. The BIOS is taking one path and may not set PCR values. Hence the response is to go into failed selftest mode.

What happens to a transport session if a command clears the TPM like revokeTrust

This is fine. The transport session is not complete but the session protected the information till the command that changed the TPM. It is impossible to get a log from the session or to sign the session but that is what the caller wanted.

**End of informative comment**

### 2.2.1 Delegation Questions

**Start of informative comment**

Is loading the table by untrusted process ok? Does this cause a problem when the new table is loaded and permissions change?

Yes, the fill table can be done by any process. A TPM Owner wishing to validate the table can perform the operations necessary to gain assurance of the table entries.

Are the permissions for a table row sensitive?

Currently we believe not but there are some attack models that knowing the permissions makes the start of the attack easier. It does not make the success of the attack any easier. Example if I know that a single process is the only process in the table that has the CreateAIK capability then the attacker only attempts to break into the single process and not all others.

What software is in use to modify the table?

The table can be updated by any software or process given the capability to manage the table. Three likely sources of the software would be a BIOS process, an applet of a trusted process and a standalone self-booting (from CD-ROM) management application.

Who holds the TPM Owner password?

There is no change to the holding of the TPM Owner token. The permissions do allow the creation of an application that sets the TPM Owner token to a random value and then seals the value to the application.

How are these changes created such that there is minimal change to the current TPM?

This works by using the current authorization process and only making changes in the authorization and not for each and every command.

What about S3 and other events?

Permissions, once granted, are non-volatile.

The permission bit to changeOwnerAuth (bit 11) gives rise to the functionality that the SW that has this bit can control the TPM completely. This includes removing control from the TPM Owner as the TPM Owner value will now be a random value only known to SW. There are use models where this is good and bad, do we want this functionality?

Pros and cons of physical enable table when TPM Owner is present – Pro physically present user can make SW play fair. Con – physically present user can override the desires of a TPM Owner.

Do we need to reset TPM_PERMISSION_KEY at some time?

We know that the key is NOT reset on TPM_ClearOwner.

What is the meaning of using permission table in an OIAP and OSAP mode?

Delegate table can be used in either OIAP or OSAP mode.

Can you grant permissions without assigning the permissions to a specific process?

Yes, do a SetRow with a PCR_SELECTION of null and the permissions are available to any process.

Do we need a ClearTableOwner?

I would assert that we do not need this command. The TPM Owner can perform SetRow with NULLS four times and creates the exact same thing. Not having this command lowers the number of ordinals the TPM is required to support.

There are some issues with the currently defined behavior of familyID and the verificationCount.

Talked to David for 30 mins. We decided that maxFamilyID is set to zero at manufacture, and incremented for every FamTable_SetRow

It is the responsibility of DelTable_SetRow to set the appropriate familyID

DelTable_SetRow fails if the provided familyID is not active and present somewhere in the FamTable

FillTable works differently. It effectively resets the family table (invalidating all active rows) and sets up as many rows as are needed based on the number of families specified in FillTable

TCG Published

This still needs a bit of work. Presumably the caller of FillTable uses a “fake” familyID, and this is changed to the actual familyID when the fill happens

There are some issues with the verificationCount.

Uber-issue. If none of the rows in the table are allowed to create other rows and export them, then the “sign” of the table is meaningful

If one of the rows is allowed to create and export new rows, is there any real meaning to “the current set of exported rows?” (i.e. SW can just up and make new rows).

Should section 4.4, TPM_DelTable_ClearTable), section 4.5 (TPM_DelTable_SetEnable), and section 4.7 (TPM_DelTable_Set_Admin) all say “there must be UNAMBIGUOUS evidence of the presence of physical access…” Is this okay?

Answer: No, group agreed to change UNAMBIGUOUS to BEST EFFORT in all three sections.

Is FamilyID a sensitive value?

If so, why? Agreement: FamilyID is not a sensitive value.

Should TPM_TakeOwnership be included in permissions bits (see bit 12 in section 3.1)?

Enables a better administrative monitor and may enable user to take ownership easier. Agreement leave it in and change informative comments to reflect the reasons.

[From the TPM_DelTable_SetRow command informative comments]: Note that there are two types of rights: family rights (you can either edit your family’s rows or grab new rows) and administrative rights.

This is really just an editor’s note, not a question to be resolved.

[From the TPM_DelTable_ExportRow command informational comments]:

Does not effect content of exported row left behind in the table;

Valid for all rows in the table;

Does not need to be OwnerAuth’d;

Family Rights are that family can only export a row from rows 0-3 if row belongs to the family, but rows 4 and upwards can be exported by any Trusted Process, without any family checking being done. This is really just an editor’s note, not a question to be resolved.

When a Family Table row is set, the verificationCount is set to 1, make sure that is consistently used in all other command actions.

Done.

SetEnable and SetEnableOwner enable and disable all rows in a table, not just the rows belong to the family of the process that used the SetEnable and/or SetEnableOwner commands. This is also true for SetAdmin and SetAdminOwner. Can anybody come up with a use scenario where that causes any problems?

In command actions where the TPM must walk the delegation table looking for a configuration that matches the command input parameters (PCRinfo and/or authValues) and there are rows in the table with duplicate values, what does the TPM do? Is there any reason not to use the rule “the TPM starts walking the table starting with the first row and use the first row it finds with matching values”?

> Answer to this question may mean change to pseudo code in section 2.3, Using the AuthData Value, which currently shows the TPM walking the delegation table, starting with the first row, and using the first row it finds with matching values.

What familyID value signals a family table row that is not in use/contains invalid values?

> To get consistency in all the command Actions that use this, that FamilyID value has been edited in all places to be NULL, instead of 0. Yes, FamilyID value of NULL signals a family table row that is not in use or contains invalid values.

From section 2.4, Delegate Table Fill and Enablement: “The changing of a TPM Owner does not automatically clear the delegate table. Changing a TPM Owner does disable all current delegations, including exported rows, and requires the new TPM Owner to re-enable the delegations in the table. The table entry values like trusted process identification and delegations to that process are not effected by a change in owner. THE AUTHDATA VALUES DO NOT SURVIVE THE OWNERSHIP CHANGE.” Question: If this is true, no delegations work after a change of owner. How does the new owner set new AuthData values?

> The simple way of handling this is to get AdminMonitor to own backing up delegations at first owner install and then be run by new owner, and AdminMonitor uses FillTable, to handle “Owner migration.” Or, for another use option, is for second owner to pick-up PCR-ID’s and delegations bits from previous owner – what is the most straight-forward way to do this?

In section 3.1 (Delegate Definitions bit map table), several commands that do not require owner authorization are in the table and can be delegated: TPM_SetTempDeactivated (bit 15), TPM_ReadPubek (bit 7), and TPM_LoadManuMaintPub (bit 3), Why?

In section 3.3 it is stated, “The Family ID resets to NULL on each change of TPM Owner.” This invalidates all delegations. Is this what we want?

> You don’t have to blow away FamilyID to blow away the blobs, because key is gone. So this is not required – can eliminate these actions.

In section 3.12, why is TPM_DELEGATE_LABEL included in the table?

In section 4.2 (TPM_DelTable_FillTable), is it okay to delete requirement that delegate table be empty? Also, in Action 14, now that we have both persistent and volatile tableAdmin flags, should this command set volatile tableAdmin flag to FALSE upon completion?

> The delegate table does not need to be empty to use the TPM_DelTable_FillTable command, Also, a paragraph has been added to Informative comment for TPM_DelTable_FillTable that points out usefulness of immediately following TPM_DelTable_FillTable with TPM_Delegate_TempSetAdmin, to stop table administration in the current boot cycle.

In section 4.15 (TPM_FamTable_IncrementCount), why does this command require TPMOwner authorization, as currently documented in section 4.15?

> IncrementCount is gated by tableAdmin, which seems sufficient, and use of ownerauth makes it difficult to automatically verify a table using a CDROM.

In section 4.3 (TPM_DelTable_FillTableOwner), in the Action 3d, use OTP[80] = MFG(x1) in place of oneTimePad[n] = SHA1(x1 || seed[n]))?, yes.

In section 4.9 (TPM_DelTable_SetRow), is invalidateRow input parameter really needed?

It is only used in action 5. Couldn’t action 5 simply read “Set N1 -> familyID = NULL”?

There is no easy way to generate a blob that can be used to delegate migration authority for a user key.

This is because the TPM does not store the migration authority on the chip as the migration command involves an encrypted key, not a loaded one. One could invent a ‘CreateMigrationDelegationBlob’ that took the encrypted key as input and generated the encrypted delegation blob as output, but it would not be pretty. Sorry Dave.

If a delegate row in NV memory (nominally 4 rows) is to refer to a user key (instead of owner auth), then it needs to include a hash of the public key. It could be that the NV table is restricted to owner auth delegations, this would save 80 bytes of NV store and also simplify the LoadBlob command.

Maybe would simplify other things. I would definitely NOT permit user keys in the table to be run with the legacy OSAP and OIAP ordinals.

A few more GetCapability values are also required, the usual constants that we discussed and also the two readTable caps.

TBD Verify that Delegate Table Management commands (see section 2.8) cover all the functionality of obsolete or updated commands.

Redefine bits 16 and above in Delegation Definitions table (section 3.1). In particular, can new command set (with TPM_FAMILY_OPERATION options as defined in section 3.20) be delegated individually and appropriately. Also, how many user key authorized commands will be delegated?

Is new TPM_FAMILY_FLAGS field of family table (defined in section 3.5) sensitive data?

DSAP informative comment needs to be completed (section 4.1). In particular, does the statement “The DSAP command works like OSAP except it takes an encrypted blob – an encrypted delegate table row -- as input” sufficient? Or do some particular differences between DSAP and OSAP have to be pointed out in this informative comment??

The TPM_Delegate_LoadBlob[Owner] commands cannot be used to load key delegation blobs into the TPM. Is another ordinal required to do that?

Is it okay for TPM_Delegate_LoadBlob[Owner] commands to ignore enable/disable use/admin flags in family table rows?

Is it wise to delegate TPM_DelTable_ConvertBlob command (defined in section 4.11)? Does current definition of this command support section 2.7 scenarios?

Is there a privacy problem with DelTable_ReadRow since the contents may not be identical from TPM to TPM?

Are DSAP sessions being pooled with the other sessions? if so, can one save\load them by context functions? if not, then there should be a restriction in saveContext.

DSAP are "normal" authorization sessions and would save/load with OIAP and OSAP sessions

**End of informative comment**

### 2.2.2 NV Questions

**Start of informative comment**

You would set this by using a new ordinal that is unauthorized and only turns the flag on to lock everything. Yet another ordinal? Do we need it? Is this an important functionality for the uses we see?

> Yes this allows us to have "close" to write once functionality. What the functionality would be is that the RTM would assure that the proper information is present in the TPM and then "lock" the area. One could create this functionality by having the RTM change the authorization each time but then you would need to eat more NV store so save the sealed AuthData value. I think that is easier to have an ordinal than eat the NV space and require a much more complex programming model.

Is it OK to have an element partially written?

> Given that we have chunks there has to be a mechanism to allow partial writes.

If an element is partially written, how does a caller know that more needs to be written?

> I would say the use model that provides the ability to write – read, in a loop is just not supported. Get it all written and then do the read.

Usage of the lock bit: as you wrote, the RTM would assure that the proper information is present in the TPM and then "lock" the area. so why in action #4 we should also check bWritten when the lock bit is set? should be as action #3b of TPM_NV_DefineSpace, if lock is set - return error

> [Grawrock, David] Not quite, the use model I was trying to create was the one where the TPM was locked and the user was attempting to add a new area. If the locked bit doesn't allow for writing once to a new area, one must reboot to perform the write and also tell the RTM what the value to write must be. So this allows the creator of an area to write it once and then it flows with the locked bit.

Can you delete a NV value with only physical presence?

> [Grawrock, David] You can't delete with physical presence, you must use owner authorization. This I think is a reasonable restriction to avoid burn problems.

Why is there no check on the writes for a TPM Owner?

> The check for an owner occurred during the TPM_NV_DefineSpace. It is imperative that the TPM_NV_DefineSpace set in place the appropriate restrictions to limit the potential for attacks on the NV storage area.

Description of maxNVBufSize is confusing to me. Why is this value related to the input size? And since there is no longer any 'written' bits, why is there a maximum area size at all?

> [Grawrock, David] This is a fixed size and set by the TPM manufacturer. I would see values like the input buffer, transport sessions etc all coming up with the max size the TPM can handle. This does NOT indicate what is available on the TPM right now. The TPM could have 4k of space but max size would be 782 and would always report that number. If the available space fell to 20 bytes this value would still be 782.

If the storage area is an opaque area to the TPM (as described), then how does the TPM know what PCR registers have been used to seal a blob?

The VALUES of the area are opaque, the attributes to control access are not. So if the attributes indicate that PCR restrictions are in place the TPM keeps those PCR values as part of the index attributes. This in reality seals the value as there is no need for tpmProof since the value never leaves the TPM.

**End of informative comment**

# 3. Protection

## 3.1 Introduction

**Start of informative comment**

The Protection Profile in the Conformance part of the specification defines the threats that are resisted by a platform. This section, "Protection," describes the properties of selected capabilities and selected data locations within a TPM that has a Protection Profile and has not been modified by physical means.

This section introduces the concept of protected capabilities and the concept of shielded locations for data. The ordinal set defined in part II and III is the set of protected capabilities. The data structures in part II define the shielded locations.

- A protected capability is one whose correct operation is necessary in order for the operation of the TCG Subsystem to be trusted.

- A shielded location is an area where data is protected against interference and prying, independent of its form.

This specification uses the concept of protected capabilities so as to distinguish platform capabilities that must be trustworthy. Trust in the TPM depends critically on the protected capabilities. Platform capabilities that are not protected capabilities must (of course) work properly if the TCG Subsystem is to function properly.

This specification uses the concept of shielded locations, rather than the concept of "shielded data." While the concept of shielded data is intuitive, it is extraordinarily difficult to define because of the imprecise meaning of the word "data." For example, consider data that is produced in a safe location and then moved into ordinary storage. It is the same data in both locations, but in one it is shielded data and in the other it is not. Also, data may not always exist in the same form. For example, it may exist as vulnerable plaintext, but also may sometimes be transformed into a logically protected form. This data continues to exist, but doesn't always need to be shielded data - the vulnerable form needs to be shielded data, but the logically protected form does not. If a specific form of data requires protection against interference or prying, it is therefore necessary to say "if the data-D exists, it must exist only in a shielded location." A more concise expression is "the data-D must be extant only in a shielded location."

Hence, if trust in the TCG Subsystem depends critically on access to certain data, that data should be extant only in a shielded location and accessible only to protected capabilities. When not in use, such data could be erased after conversion (using a protected capability) into another data structure. Unless the other data structure was defined as one that must be held in a shielded location, it need not be held in a shielded location.

**End of informative comment**

1. The data structures described in part II of the TPM specifications MUST NOT be instantiated in a TPM, except as data in TPM-shielded-locations.

2. The ordinal set defined in part II and III of the TPM specifications MUST NOT be instantiated in a TPM, except as TPM-protected-capabilities.

TCG Published

3. Functions MUST NOT be instantiated in a TPM as TPM-protected-capabilities if they do not appear in the ordinal set defined in part II and III of the TPM specifications.

## 3.2 Threat

**Start of informative comment**

This section, “Threat,” defines the scope of the threats that must be considered when considering whether a platform facilitates subversion of capabilities and data in a platform.

The design and implementation of a platform determines the extent to which the platform facilitates subversion of capabilities and data within that platform. It is necessary to define the attacks that must be resisted by TPM-shielded locations and TPM-protected capabilities in that platform.

The TCG specifications define all attacks that are resisted by the TPM. These attacks must be considered when determining whether the integrity of TPM-protected capabilities and data in TPM-shielded locations can be damaged. These attacks must be considered when determining whether there is a backdoor method of obtaining access to TPM-protected capabilities and data in TPM-shielded locations. These attacks must be considered when determining whether TPM-protected capabilities have undesirable side effects.

**End of informative comment**

1. For the purposes of the “Protection” section of the specification, the threats that MUST be considered when determining whether the TPM facilitates subversion of TPM-protected-capabilities or data in TPM-shielded-locations SHALL include
   a. The methods inherent in physical attacks that fail if the TPM complies with the “physical protection” requirements specified by TCG
   b. All methods that require execution of instructions in a computing engine in the platform

## 3.3 Protection of functions

**Start of informative comment**

A TPM-protected-capability must be used to modify TPM-protected capabilities. Other methods must not be allowed to modify TPM-protected capabilities. Otherwise, the integrity of TPM-protected capabilities is unknown.

**End of informative comment**

1. A TPM SHALL NOT facilitate the alteration of TPM-protected-capabilities, except by TPM-protected capabilities.

## 3.4 Protection of information

**Start of informative comment**

TPM-protected capabilities must provide the only means from outside the TPM to access information represented by data in TPM-shielded-locations. Otherwise, a rogue can reveal data in TPM-shielded-locations, or create a derivative of data from TPM-shielded-locations (in a way that maintains some or all of the information content of the data) and reveal the derivative.

**End of informative comment**

1. A TPM SHALL NOT export data that is dependent upon data structures described in part II of the TPM specifications, other than via a TPM-Protected-Capability.

## 3.5 Side effects

**Start of informative comment**

An implementation of a TPM-protected capability must not disclose the contents of TPM-shielded locations. The only exceptions are when such disclosure is inherent in the definition of the capability or in the methods used by the capability. For example, a capability might be designed specifically to reveal hidden data or might use cryptography and hence always be vulnerable to cryptanalysis. In such cases, some disclosure or risk of disclosure is inherent and cannot be avoided. Other forms of disclosure (by side effects, for example) must always be avoided.

**End of informative comment**

1. The implementation of a TPM-protected-capability in a TPM SHALL NOT facilitate the disclosure or the exposure of information represented by data in TPM-shielded–locations, except by means unavoidably inherent in the TPM definition.

## 3.6 Exceptions and clarifications

**Start of informative comment**

These exceptions to the blanket statements in the generic “protection” requirements (above) are fully compatible with the intended effect of those statements. These exceptions affect TCG-data that is available as plain-text outside the TPM and TCG-data that can be used without violating security or privacy. These exceptions are valuable because they approve use of TPM resources by vendor-specific commands in particular circumstances.

These clarifications to the blanket statements of the generic “protection” requirements (above) do not materially change the effect of those statements, but serve to approve specific legitimate interpretations of the requirements.

**End of informative comment**

1. A Shielded Location is a place (memory, register, etc.) where data is protected against interference and exposure, independent of its form
2. A TPM-Protected-Capability is an operation defined in and restricted to those identified in part II and III of the TPM specifications.
3. A vendor specific command or capability MAY use the standard TCG owner/operator authorization mechanism
4. A vendor specific command or capability MAY utilize a TPM_PUBKEY structure stored on the TPM so long as the usage of that TPM_PUBKEY structure is authorized using the standard TCG authorization mechanism.
5. A vendor specific command or capability MAY use a sequence of standard TCG commands. The command MUST propagate the locality used for the call to the used TCG commands or capabilities, or set locality to 0.

6. A vendor specific command or capability that takes advantage of exceptions and clarifications to the “protection” requirements MUST be defined as part of the security target of the TPM. Such a vendor specific command or capability MUST be evaluated to meet the Platform Specific TPM and System Security Targets.

7. If a TPM employs vendor-specific cipher-text that is protected against subversion to the same or greater extent as internal TPM-resources stored outside the TPM with TCG-defined methods, that vendor-specific cipher-text does not necessarily require protection from physical attack. If a TPM location stores only vendor-specific cipher-text that does not require protection from physical attack, that location can be ignored when determining whether the TPM complies with the "physical protection" requirements specified by TCG.

TCG Published

# 4. TPM Architecture

## 4.1 Interoperability

**Start of informative comment**

The TPM must support a minimum set of algorithms and operations to meet TCG specifications.

Algorithms

RSA, SHA-1, HMAC

The algorithms and protocols are the minimum that the TPM must support. Additional algorithms and protocols may be available to the TPM. All algorithms and protocols available in the TPM must be included in the TPM and platform credential.

The reason to specify these algorithms is two fold. The first is to know and understand the security properties of selected algorithms; identify appropriate key sizes and ensure appropriate use in protocols. The second reason is to define a base level of algorithms for interoperability.

**End of informative comment**

## 4.2 Components

**Start of informative comment**

The following is a block diagram Figure 4:a shows the major components of a TPM.

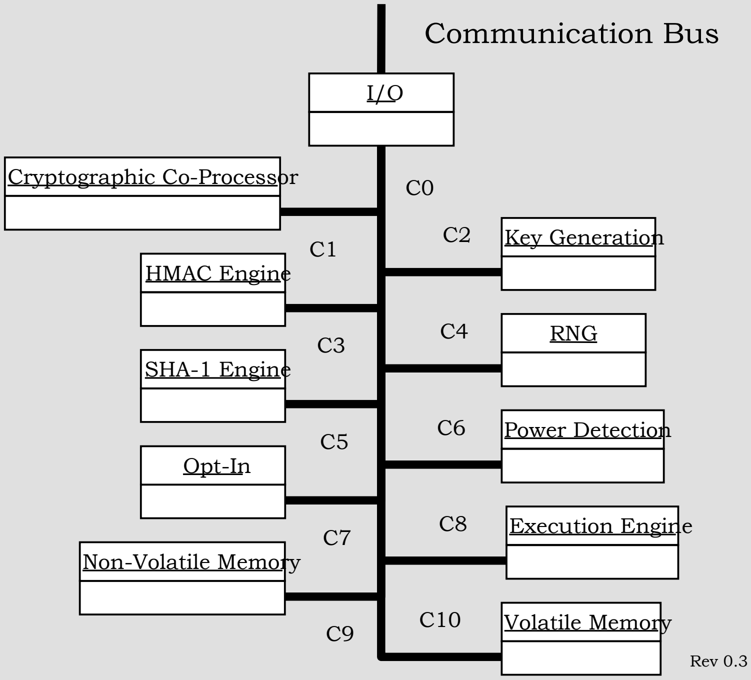


Figure 4:a - TPM Component Architecture

**End of informative comment**

### 4.2.1 Input and Output

**Start of informative comment**

The I/O component, Figure 4:a C0, manages information flow over the communications bus. It performs protocol encoding/decoding suitable for communication over external and internal buses. It routes messages to appropriate components. The I/O component enforces access policies associated with the Opt-In component as well as other TPM functions requiring access control.

The main specification does not require a specific I/O bus. Issues around a particular I/O bus are the purview of a platform specific specification.

**End of informative comment**

1. The number of incoming operand parameter bytes must exactly match the requirements of the command ordinal. If the command contains more or fewer bytes than required, the TPM MUST return TPM_BAD_PARAMETER.

### 4.2.2 Cryptographic Co-Processor

**Start of informative comment**

The cryptographic co-processor, Figure 4:a C1, implements cryptographic operations within the TPM. The TPM employs conventional cryptographic operations in conventional ways. Those operations include the following:

Asymmetric key generation (RSA)

Asymmetric encryption/decryption (RSA)

Hashing (SHA-1)

Random number generation (RNG)

The TPM uses these capabilities to perform generation of random data, generation of asymmetric keys, signing and confidentiality of stored data.

The TPM may symmetric encryption for internal TPM use but does not expose any symmetric algorithm functions to general users of the TPM.

The TPM may implement additional asymmetric algorithms. TPM devices that implement different algorithms may have different algorithms perform the signing and wrapping.

If the TPM uses RSA with the required key length (2048 bits for storage keys), the output of all commands for key or data blob generation (e.g., TPM_CreateWrapKey, TPM_Seal, TPM_Sealx, TPM_MakeIdentity) consists of only one block.  However, if the TPM uses other asymmetric algorithms that result in more than one output block for these commands, the integrity of the blobs must be protected by the TPM (by means of appropriate chaining mechanisms).

**End of informative comment**

1. The TPM MAY implement other asymmetric algorithms such as DSA or elliptic curve.

a. These algorithms may be in use for wrapping, signatures and other operations. There is no guarantee that these keys can migrate to other TPM devices or that other TPM devices will accept signatures from these additional algorithms.

b. If the output key or data blob generated with a storage key consists of more than one block, the TPM MUST protect the integrity of the blob by means of appropriate chaining mechanisms.

2. All storage keys MUST be of strength equivalent to a 2048 bits RSA key or greater. The TPM SHALL NOT load a storage key whose strength less than that of a 2048 bits RSA key.

3. All AIK MUST be of strength equivalent to a 2048 bits RSA key, or greater.

#### 4.2.2.1 RSA Engine

**Start of informative comment**

The RSA asymmetric algorithm is used for digital signatures and for encryption.

For RSA keys the PKCS #1 standard provides the implementation details for digital signature, encryption and data formats.

There is no requirement concerning how the RSA algorithm is to be implemented. TPM manufacturers may use Chinese Remainder Theorem (CRT) implementations or any other method. Designers should review P1363 for guidance on RSA implementations.

For keys that are required to be 2048-bit RSA keys, the default $2^{16}+1$ exponent will be required. This guarantees the strength of the key without walking a hierarchy that cannot necessarily be walked reliably.

**End of informative comment**

1. The TPM MUST support RSA.

2. The TPM MUST use the RSA algorithm for encryption and digital signatures.

3. The TPM MUST support key sizes of 512, 1024, and 2048 bits. The TPM MAY support other key sizes.

   a. The minimum RECOMMENDED key size is 2048 bits.

   b. In FIPS mode, the minimum key size MUST be 1024.

4. The TPM MUST support an RSA public exponent of $2^{16}+1$. The TPM MAY support other exponent values.

5. TPM devices that use CRT as the RSA implementation MUST provide protection and detection of failures during the CRT process to avoid attacks on the private key.

#### 4.2.2.2 Signature Operations

**Start of informative comment**

The TPM performs signatures on both internal items and on requested external blobs. The rules for signatures apply to both operations.

**End of informative comment**

1. The TPM MUST use the RSA algorithm for signature operations where signed data is verified by entities other than the TPM that performed the sign operation.
2. The TPM MAY use other asymmetric algorithms for signatures; however, there is no requirement that other TPM devices either accept or verify those signatures.
3. The TPM MUST use P1363 for the format and design of the signature output.

### 4.2.2.3 Symmetric Encryption Engine

**Start of informative comment**

The TPM uses symmetric encryption to encrypt authentication information, provide confidentiality in transport sessions and provide internal encryption of blobs stored off the TPM.

For authentication and transport sessions, the mandatory mechanism is a Vernam one-time-pad with XOR. The mechanism to generate the one-time-pad is MGF1 and the nonces from the session protocol. When encrypting authorization data, the authorization data and the nonces are the same size, 20 bytes, so a direct XOR is possible.

For transport sessions the size of data is larger than the nonces so there needs to be a mechanism to expand the entropy to the size of the data. The mechanism to expand the entropy is the MGF1 function from PKCS#1. This function provides a known mechanism that does not lower the entropy of the nonces.

AES may be supported as an alternate symmetric key encryption algorithm.

Internal protection of information can use any symmetric algorithm that the TPM designer feels provides the proper level of protection.

The TPM does not expose any of the symmetric operations for general message encryption.

**End of informative comment**

### 4.2.2.4 Using Keys

**Start of Informative comments:**

Keys can be symmetric or asymmetric.

As the TPM does not have an exposed symmetric algorithm, the TPM is only a generator, storage device and protector of symmetric keys. Generation of the symmetric key would use the TPM RNG. Storage and protection would be provided by the BIND and SEAL capabilities of the TPM. If the caller wants to ensure that the release of a symmetric key is not exposed after UNBIND/UNSEAL on delivery to the caller, the caller should use a transport session with confidentiality set.

For asymmetric algorithms, the TPM generates and operates on RSA keys. The keys can be held only by the TPM or in conjunction with the caller of the TPM. If the private portion of a key is in use outside of the TPM it is the responsibility of the caller and user of that key to ensure the protections of the key.

The TPM has provisions to indicate if a key is held exclusively for the TPM or can be shared with entities off of the TPM.

**End of informative comments.**

1. A secret key is a key that is a private asymmetric key or a symmetric key.
2. Data SHOULD NOT be used as a secret key by a TCG protected capability unless that data has been extant only in a shielded location.
3. A key generated by a TCG protected capability SHALL NOT be used as a secret key unless that key has been extant only in a shielded location.
4. A secret key obtained by a TCG protected capability from a Protected Storage blob SHALL be extant only in a shielded location.

### 4.2.3 Key Generation

**Start of informative comment**

The Key Generation component, Figure 4:a C2, creates RSA key pairs and symmetric keys. TCG places no minimum requirements on key generation times for asymmetric or symmetric keys.

**End of informative comment**

#### 4.2.3.1 Asymmetric – RSA

The TPM MUST generate asymmetric key pairs. The generate function is a protected capability and the private key is held in a shielded location. The implementation of the generate function MUST be in accordance with P1363.

The prime-number testing for the RSA algorithm MUST use the definitions of P1363. If additional asymmetric algorithms are available, they MUST use the definitions from P1363 for the underlying basis of the asymmetric key (for example, elliptic curve fitting).

#### 4.2.3.2 Nonce Creation

The creation of all nonce values MUST use the next n bits from the TPM RNG.

### 4.2.4 HMAC Engine

**Start of informative comment**

The HMAC engine, Figure 4:a C3, provides two pieces of information to the TPM: proof of knowledge of the AuthData and proof that the request arriving is authorized and has no modifications made to the command in transit.

The HMAC definition is for the HMAC calculation only. It does not specify the order or mechanism that transports the data from caller to actual TPM.

The creation of the HMAC is order dependent. Each command has specific items that are portions of the HMAC calculation. The actual calculation starts with the definition from RFC 2104.

RFC 2104 requires the selection of two parameters to properly define the HMAC in use. These values are the key length and the block size. This specification will use a key length of 20 bytes and a block size of 64 bytes. These values are known in the RFC as K for the key length and B as the block size.

The basic construct is

H(K XOR opad, H(K XOR ipad, text))

where

H = the SHA1 hash operation

K = the key or the AuthData

XOR = the xor operation

opad = the byte 0x5C repeated B times

B = the block length

ipad = the byte 0x36 repeated B times

text = the message information and any parameters from the command

**End of informative comment**

The TPM MUST support the calculation of an HMAC according to RFC 2104.

The size of the key (K in RFC 2104) MUST be 20 bytes. The block size (B in RFC 2104) MUST be 64 bytes.

The order of the parameters is critical to the TPM's ability to recreate the HMAC. Not all of the fields are sent on the wire for each command for instance only one of the nonce values travels on the wire. Each command interface definition indicates what parameters are involved in the HMAC calculation.

### 4.2.5 Random Number Generator

**Start of informative comment**

The Random Number Generator (RNG) component, Figure 6:a C4 is the source of randomness in the TPM. The TPM uses these random values for nonces, key generation, and randomness in signatures.

The RNG consists of a state-machine that accepts and mixes unpredictable data and a post-processor that has a one-way function (e.g. SHA-1). The idea behind the design is that a TPM can be good source of randomness without having to require a genuine source of hardware entropy.

The state-machine can have a non-volatile state initialized with unpredictable random data during TPM manufacturing before delivery of the TPM to the customers. The state-machine can accept, at any time, further (unpredictable) data, or entropy, to salt the random number. Such data comes from hardware or software sources – for example; from thermal noise, or by monitoring random keyboard strokes or mouse movements. The RNG requires a reseeding after each reset of the TPM. A true hardware source of entropy is likely to supply entropy at a higher baud rate than a software source.

When adding entropy to the state-machine, the process must ensure that after the addition, no outside source can gain any visibility into the new state of the state-machine. Neither the Owner of the TPM nor the manufacturer of the TPM can deduce the state of the state-machine after shipment of the TPM. The RNG post-processor condenses the output of the state-machine into data that has sufficient and uniform entropy. The one-way function should use more bits of input data than it produces as output.

Our definition of the RNG allows implementation of a Pseudo Random Number Generator (PRNG) algorithm. However, on devices where a hardware source of entropy is available, a PRNG need not be implemented. This specification refers to both RNG and PRNG implementations as the RNG mechanism. There is no need to distinguish between the two at the TCG specification level.

The TPM should be able to provide 32 bytes of randomness on each call. Larger requests may fail with not enough randomness being available.

**End of informative comment**

1. The RNG for the TPM will consist of the following components:
   a. Entropy source and collector
   b. State register
   c. Mixing function
2. The RNG capability is a TPM-protected capability with no access control.
3. The RNG output may or may not be shielded data. When the data is for internal use by the TPM (e.g., generation of tpmProof or an asymmetric key), the data MUST be held in a shielded location. The RNG output for internal use MUST not be known outside the TPM. In particular, it MUST not be known by the TPM manufacturer. When the data is for use by the TSS or another external caller, the data is not shielded.
4. In FIPS mode, the RNG MUST be a NIST approved RNG. The NIST self-test requirements MUST be satisfied.

### 4.2.5.1 Entropy Source and Collector

**Start of informative comment**

The entropy source is the process or processes that provide entropy. These types of sources could include noise, clock variations, air movement, and other types of events.

The entropy collector is the process that collects the entropy, removes bias, and smoothes the output. The collector differs from the mixing function in that the collector may have special code to handle any bias or skewing of the raw entropy data. For instance, if the entropy source has a bias of creating 60 percent 1s and only 40 percent 0s, then the collector design takes that bias into account before sending the information to the state register.

**End of informative comment**

1. The entropy source MUST provide entropy to the state register in a manner that provides entropy that is not visible to an outside process.
   a. For compliance purposes, the entropy source MAY be outside of the TPM; however, attention MUST be paid to the reporting mechanism.
2. The entropy source MUST provide the information only to the state register.
   a. The entropy source may provide information that has a bias, so the entropy collector must remove the bias before updating the state register. The bias removal could use the mixing function or a function specifically designed to handle the bias of the entropy source.

b. The entropy source can be a single device (such as hardware noise) or a combination of events (such as disk timings). It is the responsibility of the entropy collector to update the state register whenever the collector has additional entropy.

#### 4.2.5.2 State Register

**Start of informative comment**

The state register implementation may use two registers: a non-volatile register rngState and a volatile register. The TPM loads the volatile register from the non-volatile register on startup. Each subsequent change to the state register from either the entropy source or the mixing function affects the volatile state register. The TPM saves the current value of the volatile state register to the non-volatile register on TPM power-down. The TPM may update the non-volatile register at any other time. The reasons for using two registers are:

To handle an implementation in which the non-volatile register is in a flash device;

To avoid overuse of the flash, as the number of writes to a flash device are limited.

**End of informative comment**

1. The state register is in a TPM shielded-location.
   a. The state register MUST be non-volatile.
   b. The update function to the state register is a TPM protected-capability.
   c. The primary input to the update function SHOULD be the entropy collector.
2. If the current value of the state register is unknown, calls made to the update function with known data MUST NOT result in the state register ending up in a state that an attacker could know.
   a. This requirement implies that the addition of known data MUST NOT result in a decrease in the entropy of the state register.
3. The TPM MUST NOT export the state register.

#### 4.2.5.3 Mixing Function

**Start of informative comment**

The mixing function takes the state register and produces output. The mixing function is a TPM protected-capability. The mixing function takes the value from a state register and creates the RNG output. If the entropy source has a bias, then the collector takes that bias into account before sending the information to the state register.

**End of informative comment**

1. Each use of the mixing function MUST affect the state register.
   a. This requirement is to affect the volatile register and does not need to affect the non-volatile state register.

#### 4.2.5.4 RNG Reset

**Start of informative comment**

The resetting of the RNG occurs at least in response to a loss of power to the device.

These tests prove only that the RNG is still operating properly; they do not prove how much entropy is in the state register. This is why the self-test checks only after the load of previous state and may occur before the addition of more entropy.

**End of informative comment**

1. The RNG MUST NOT output any bits after a system reset until the following occurs:
   a. The entropy collector performs an update on the state register. This does not include the adding of the previous state but requires at least one bit of entropy.
   b. The mixing function performs a self-test. This self-test MUST occur after the loading of the previous state. It MAY occur before the entropy collector performs the first update.

### 4.2.6 SHA-1 Engine

**Start of informative comment**

The SHA-1, Figure 4:a C5, hash capability is primarily used by the TPM, as it is a trusted implementation of a hash algorithm. The hash interfaces are exposed outside the TPM to support Measurement taking during platform boot phases and to allow environments that have limited capabilities access to a hash functions. The TPM is not a cryptographic accelerator. TCG does not specify minimum throughput requirements for TPM hash services.

**End of informative comment**

1. The TPM MUST implement the SHA-1 hash algorithm as defined by FIPS-180-1.
2. The output of SHA-1 is 160 bits and all areas that expect a hash value are REQUIRED to support the full 160 bits.
3. The only commands that SHALL be presented to the TPM in-between a TPM_SHA1Start command and a TPM_SHA1Complete command SHALL be a variable number (possibly 0) of TPM_SHA1Update commands.
   a. The TPM_SHA1Update commands can occur in a transport session.
4. Throughout all parts of the specification the characters x1 || x2 imply the concatenation of x1 and x2

### 4.2.7 Power Detection

**Start of informative comment**

The power detection component, Figure 4:a C6, manages the TPM power states in conjunction with platform power states. TCG requires that the TPM be notified of all power state changes.

Power detection also supports physical presence assertions. The TPM may restrict command-execution during periods when the operation of the platform is physically constrained. In a PC, operational constraints occur during the power-on self-test (POST) and require Operator input via the keyboard. The TPM might allow access to certain commands while in a constrained execution mode or boot state. At some critical point in the POST process, the TPM may be notified of state changes that affect TPM command processing modes.

**End of informative comment**

### 4.2.8 Opt-In

**Start of informative comment**

The Opt-In component, Figure 4:a C7, provides mechanisms and protections to allow the TPM to be turned on/off, enabled/disabled, activated/deactivated. The Opt-In component maintains the state of persistent and volatile flags and enforces the semantics associated with these flags.

The setting of flags requires either authorization by the TPM Owner or the assertion of physical presence at the platform. The platform's manufacturer determines the techniques used to represent physical-presence. The guiding principle is that no remote entity should be able to change TPM status without either knowledge of the TPM Owner or the Operator is physically present at the platform. Physical presence may be asserted during a period when platform operation is constrained such as power-up.

Non-Volatile Flags:

physicalPresenceLifetimeLock

physicalPresenceHWEnable

physicalPresenceCMDEnable

Volatile Flags:

physicalPresenceLock

physicalPresence

The notation physicalPresenceV indicates the physical presence state that ordinals refer to when they say, for example, "if physical presence is asserted".

The following truth table explains the conditions in which the physicalPresenceV flag may be altered:

| Persistent / Volatile | P | P | P | V | |
|---|---|---|---|---|---|
| Control Flags | physicalPresenceLifetimeLock | physicalPresenceHWEnable | physicalPresenceCMDEnable | physicalPresenceLock | |
| Volatile Access Semantics to Physical Presence Flag | - | F | F | - | No access to physicalPresenceV flag. |
| | - | F | - | T | |
| | - | - | T | F | Access to physicalPresenceV flag through TCS_PhysicalPresence command enabled. |
| | - | T | - | - | Access to physicalPresenceV flag through hardware signal enabled. |
| | - | T | T | F | Access to physicalPresenceV flag through hardware signal or TCS_PhysicalPresence command enabled. |
| | | | | | |
| Persistent Access Semantics to Physical Presence Flag | T | F | F | - | Access to physicalPresenceV flag permanently disabled. |
| | T | F | T | T | Access to physicalPresenceV flag disabled until TPM_Startup(ST_CLEAR). |
| | T | F | T | F | Exclusive access to physicalPresenceV flag through TCS_PhysicalPresence command permanently enabled. |
| | T | T | F | - | Exclusive access to physicalPresenceV flag through hardware signal permanently enabled. |
| | T | T | T | F | Access to physicalPresenceV flag through hardware signal or TCS_PhysicalPresence command permanently enabled. |

Table 4:a - Physical Presence Semantics

TCG also recognizes the concept of unambiguous physical presence. Conceptually, the use of dedicated electrical hardware providing a trusted path to the Operator has higher precedence than the physicalPresenceV flag value. Unambiguous physical presence may be used to override physicalPresenceV flag value under conditions specified by platform specific design considerations.

Additional details relating to physical presence can be found in sections on Volatile and Non-volatile memory.

**End of informative comment**

### 4.2.9 Execution Engine

**Start of informative comment**

The execution engine, Figure 4:a C8, runs program code to execute the TPM commands received from the I/O port. The execution engine is a vital component in ensuring that operations are properly segregated and shield locations are protected.

**End of informative comment**

### 4.2.10 Non-Volatile Memory

**Start of informative comment**

Non-volatile memory component, Figure 4:a C9, is used to store persistent identity and state associated with the TPM. The NV area has set items (like the EK) and also is available for allocation and use by entities authorized by the TPM Owner.

The TPM designer should consider the use model of the TPM and if the use of NV storage is a concern. NV storage does have a limited life and using the NV storage in a high volume use model may prematurely wear out the TPM.

There is no requirement for the TPM to protect against wear out by detecting that a write of the same value need not be performed. Applications should avoid frequent writes of the same value. For example, precede a TPM_SetCapability with a TPM_GetCapability and skip the write if the TPM already holds the desired value.

**End of informative comment**

## 4.3 Data Integrity Register (DIR)

**Start of informative comment**

The DIR were a version 1.1 function. They provided a place to store information using the TPM NV storage.

In 1.2 the DIR are deprecated and the use of the DIR should move to the general purpose NV storage area.

The TPM must still support the functionality of the DIR register in the NV storage area.

**End of informative comment**

1. A TPM MUST provide one Data Integrity Register (DIR)
   a. The TPM DIR commands are deprecated in 1.2
   b. The TPM MUST reserve the space for one DIR in the NV storage area
   c. The TPM MAY have more than 1 DIR.
2. The DIR MUST be 160-bit values and MUST be held in TPM shielded-locations.
3. The DIR MUST be non-volatile (values are maintained during the power-off state).
   a. A TPM implementation need not provide the same number of DIRs as PCRs.

## 4.4 Platform Configuration Register (PCR)

**Start of informative comment**

A Platform Configuration Register (PCR) is a 160-bit storage location for discrete integrity measurements. There are a minimum of 16 PCR registers. All PCR registers are shielded-locations and are inside of the TPM. The decision of whether a PCR contains a standard measurement or if the PCR is available for general use is deferred to the platform specific specification.

A large number of integrity metrics may be measured in a platform, and a particular integrity metric may change with time and a new value may need to be stored. It is difficult to authenticate the source of measurement of integrity metrics, and as a result a new value of an integrity metric cannot be permitted to simply overwrite an existing value. (A rogue could erase an existing value that indicates subversion and replace it with a benign value.) Thus, if values of integrity metrics are individually stored, and updates of integrity metrics must be individually stored, it is difficult to place an upper bound on the size of memory that is required to store integrity metrics.

The PCR is designed to hold an unlimited number of measurements in the register. It does this by using a cryptographic hash and hashing all updates to a PCR. The pseudo code for this is:

```
PCRi New = HASH ( PCRi Old value || value to add)
```

There are two salient properties of cryptographic hash that relate to PCR construction. Ordering – meaning updates to PCRs are not commutative. For example, measuring (A then B) is not the same as measuring (B then A).

The other hash property is one-way-ness. This property means it should be computationally infeasible for an attacker to determine the input message given a PCR value. Furthermore, subsequent updates to a PCR cannot be determined without knowledge of the previous PCR values or all previous input messages provided to a PCR register since the last reset.

If the TPM is disabled or deactivated, commands that extend a PCR (e.g., TPM_Extend, TPM_SHA1CompleteExtend) return a PCR value of all zeros, and commands that use the PCRs (e.g., TPM_PCRRead) are not available. However, the commands that extend a PCR still update the PCR correctly and return success. For disabled, this is because the TPM may become enabled later, and so must not miss a measurement. For deactivated, this is because resource limited code like the CRTM will perform extends and may not be able to handle a deactivated error case.

**End of informative comment**

1. The PCR MUST be a 160-bit field that holds a cumulatively updated hash value
2. The PCR MUST have a status field associated with it
3. The PCR MUST be in the RTS and should be in volatile storage
4. The PCR MUST allow for an unlimited number of measurements to be stored in the PCR
5. The PCR MUST preserve the ordering of measurements presented to it
6. A PCR MUST be set to the default value as specified by the PCRReset attribute
7. A TPM implementation MUST provide 16 or more independent PCRs. These PCRs are identified by index and MUST be numbered from 0 (that is, PCR0 through PCR15 are required for TCG compliance). Vendors MAY implement more registers for general-purpose use. Extra registers MUST be numbered contiguously from 16 up to max – 1, where max is the maximum offered by the TPM.
8. The TCG-protected capabilities that expose and modify the PCRs use a 32-bit index, indicating the maximum usable PCR index. However, TCG reserves register indices 230 and higher for later versions of the specification. A TPM implementation MUST NOT provide registers with indices greater than or equal to 230. In this specification, the following terminology is used (although this internal format is not mandated).

9. The PSS MUST define at least define one measurement that the RTM MUST make and the PCR where the measurement is stored.

10. A TCG measurement agent MAY discard a duplicate event instead of incorporating it in a PCR, provided that:

11. A relevant TCG platform specification explicitly permits duplicates of this type of event to be discarded

12. The PCR already incorporates at least one event of this type

13. An event of this type previously incorporated into the PCR included a statement that duplicate such events may be discarded. This option could be used where frequent recording of sleep states will adversely affect the lifetime of a TPM, for example.

14. PCRs and the protected capabilities that operate upon them MAY NOT be used until power-on self-test (TPM POST) has completed. If TPM POST fails, the TPM_Extend operation will fail; and, of greater importance, the TPM_Quote operation and TPM_Seal operations that respectively report and examine the PCR contents MUST fail. At the successful completion of TPM POST, all PCRs MUST be set to their default value (either 0x00…00 or 0xFF…FF). Additionally, the UINT32 flags MUST be set to zero.

# 5. Endorsement Key Creation

**Start of informative comment**

The TPM contains a 2048-bit RSA key pair called the endorsement key (EK). The public portion of the key is the PUBEK and the private portion the PRIVEK. Due to the nature of this key pair, both the PUBEK and the PRIVEK have privacy and security concerns.

The TPM has the EK generated before the end customer receives the platform. The Trusted Platform Module Entity (TPME) that causes EK generation is also the entity that will create and sign the EK credential attesting to the validity of the TPM and the EK. The TPME is typically the TPM manufacturer.

The TPM can generate the EK internally using the TPM_CreateEndorsementKey or by using an outside key generator. The EK needs to indicate the genealogy of the EK generation.

Subsequent attempts to either generate an EK or insert an EK must fail.

If the data structure TPM_ENDORSEMENT_CREDENTIAL is stored on a platform after an Owner has taken ownership of that platform, it SHALL exist only in storage to which access is controlled and is available to authorized entities.

**End of informative comment**

1. The EK MUST be a 2048-bit RSA key
   a. The public portion of the key is the PUBEK
   b. The private portion of the key is the PRIVEK
   c. The PRIVEK SHALL exist only in a TPM-shielded location.
2. Access to the PRIVEK and PUBEK MUST only be via TPM protected capabilities
   a. The protected capabilities MUST require TPM Owner authentication or operator physical presence
3. The generation of the EK may use a process external to the TPM and TPM_CreateEndorsementKeyPair
   a. The external generation MUST result in an EK that has the same properties as an internally generated EK
   b. The external generation process MUST protect the EK from exposure during the generation and insertion of the EK
   c. After insertion of the EK the TPM state MUST be the same as the result of the TPM_CreateEndorsementKeyPair execution
   d. The process MUST guarantee correct generation, cryptographic strength, uniqueness, privacy, and installation into a genuine TPM, of the EK
   e. The entity that signs the EK credential MUST be satisfied that the generation process properly generated the EK and inserted it into the TPM
   f. The process MUST be defined in the target of evaluation (TOE) of the security target in use to evaluate the TPM

## 5.1 Controlling Access to PRIVEK

**Start of informative comment**

Exposure of the PRIVEK is a security concern.

The TPM must ensure that the PRIVEK is not exposed outside of the TPM

**End of informative comment**

1. The PRIVEK MUST never be out of the control of a TPM shielded location

## 5.2 Controlling Access to PUBEK

**Start of informative comment**

There are no security concerns with exposure or use of the PUBEK.

Privacy guidelines suggest that PUBEK could be considered personally identifiable information (PII) if it were associated in some way with personal information (PI) or associated with other PII, but PUBEK alone cannot be considered PII. Arbitrary random numbers do not represent a threat to privacy unless further associated with PI or PII. The PUBEK is an arbitrary random number that may be associated with aggregate platform information, but not personally identifiable information.

An EK may become associated with personally identifiable information when an alias platform identifier (AIK) is also associated with PI. The attestation service could include personal information in the AIK credential, thereby making the AIK-PUBEK association PII – but not before.

The association of PUBEK with AIK therefore is important to protect via privacy guidelines. The owner/user of the TPM should be able to control whether PUBEK is disclosed along with AIK. The owner/user should be notified of personal information that might be added to an AIK credential, which could result in AIK being considered PII. The owner/user should be able to evaluate the mechanisms used by an attestation entity to protect PUBEK-AIK associations before disclosure occurs. No other entity should be privy to owner/user authorized disclosure besides the intended attestation entity.

Several commands may be used to negotiate the conditions of PUBEK-AIK disclosure. TPM_MakeIdentity discloses PUBEK-AIK in the context of requesting an AIK credential. TPM_ActivateIdentity ensures the owner/user has not been spoofed by an interloper. These interfaces allow the owner/user to choose whether disclosure is acceptable and control the circumstances under which disclosure takes place. They do not allow the owner/user the ability to retain control of PUBEK-AIK subsequent to disclosure except by traditional means of trusting the attestation entity to abide by an acceptable privacy policy. The owner/user is able to associate the accepted privacy policy with the disclosure operation (e.g. TPM_MakeIdentity).

A persistent flag called readPubek can be set to TRUE to permit reading of PUBEK via TPM_ReadPubek. Reporting the PUBEK value is not considered privacy sensitive because it cannot be associated with any of the AIK keys managed by the TPM without using TPM protected-capabilities. Keys are encrypted with a nonce when flushed from TPM shielded-locations, Cryptanalysis of flushed keys will not reveal an association of EK to any AIK.

The command that manipulates the readPubek flag is TPM_DisablePubekRead.

**End of informative comment**

# 6. Attestation Identity Keys

**Start of informative comment**

See 11.4 Attestation Identity Keys.

**End of informative comment**

TCG Published

# 7. TPM Ownership

**Start of informative comment**

Taking ownership of a TPM is the process of inserting a shared secret into a TPM shielded-location. Any entity that knows the shared secret is a TPM Owner. Proof of ownership occurs when an entity, in response to a challenge, proves knowledge of the shared secret. Certain operations in the TPM require authentication from a TPM Owner.

Certain operations also allow the human, with physical possession of the platform, to assert TPM Ownership rights. When asserting TPM Ownership, using physical presence, the operations must not expose any secrets protected by the TPM.

The platform owner controls insertion of the shared secret into the TPM. The platform owner sets the NV persistent flag ownershipEnabled that allows the execution of the TPM_TakeOwnership command. The TPM_SetOwnerInstall, the command that controls the value ownershipEnabled, requires the assertion of physical presence.

Attempting to execute TPM_TakeOwnership fails when a TPM already has an owner. To remove an owner when the current TPM Owner is unable to remove themselves, the human that is in possession of the platform asserts physical presence and executes TPM_ForceClear which removes the shared secret.

The insertion protocol that supplies the shared secret has the following requirements: confidentiality, integrity, remoteness and verifiability.

To provide confidentiality the proposed TPM Owner encrypts the shared secret using the PUBEK. This requires the PRIVEK to decrypt the value. As the PRIVEK is only available in the TPM the encrypted shared secret is only available to the intended TPM.

The integrity of the process occurs by the TPM providing proof of the value of the shared secret inserted into the TPM.

By using the confidentiality and integrity, the protocol is useable by TPM Owners that are remote to the platform.

The new TPM Owner validates the insertion of the shared secret by using integrity response.

**End of informative comment**

The TPM MUST ship with no Owner installed. The TPM MUST use the ownership-control protocol (OIAP or OSAP)

## 7.1 Platform Ownership and Root of Trust for Storage

**Start of informative comment**

The semantics of platform ownership are tied to the Root-of-trust-for-storage (RTS). The TPM_TakeOwnership command creates a new Storage Root Key (SRK) and new tpmProof value whenever a new owner is established. It follows that objects owned by a previous owner will not be inherited by the new owner. Objects that should be inherited must be transferred by deliberate data migration actions.

**End of informative comment**

# 8. Authentication and Authorization Data

**Start of informative comment**

Using security vernacular the terms below apply to the TPM for this discussion:

Authentication: The process of providing proof of claimed ownership of an object or a subject's claimed identity.

Authorization: Granting a subject appropriate access to an object.

Each TPM object that does not allow "public" access contains a 160-bit shared secret. This shared secret is enveloped within the object itself. The TPM grants use of TPM objects based on the presentation of the matching 160-bits using protocols designed to provide protection of the shared secret. This shared secret is called the AuthData.

Neither the TPM, nor its objects (such as keys), contain access controls for its objects (the exception to this is what is provided by the delegation mechanism). If an subject presents the AuthData, that subject is granted full use of the object based on the object's capabilities, not a set of rights or permissions of the subject. This apparent overloading of the concepts of authentication and authorization has caused some confusion. This is caused by having two similarly rooted but distinct perspectives.

From the perspective of the TPM looking out, this AuthData is its sole mechanism for authenticating the owner of its objects, thus from its perspective it is authentication data. However, from the application's perspective this data is typically the result of other functions that might perform authentications or authorizations of subjects using higher level mechanisms such as OS login, file system access, etc. Here, AuthData is a result of these functions so in this usage, it authorizes access to the TPM's objects. From this perspective, i.e., the application looking in on the TPM and its objects, the AuthData is authorization data. For this reason, and thanks to a common root within the English language, the term for this data is chosen to be AuthData and is to be interpreted or expanded as either authentication data or authorization data depending on context and perspective.

The term AuthData refers to the 160-bit value used to either prove ownership of, or authorization to use, an object. This is also called the object's shared secret. The term authorization will be used when referring the combined action of verifying the AuthData and allowing access to the object or function. The term authorization session applies to a state where the AuthData has been authentication and a session handle established that is associated with that authentication.

A wide-range of objects use AuthData. It is used to establish platform ownership, key use restrictions, object migration and to apply access control to opaque objects protected by the TPM.

AuthData is a 160-bit shared-secret plus high-entropy random number. The assumption is the shared-secret and random number are mixed using SHA-1 digesting, but no specific function for generating AuthData is specified by TCG.

TCG command processing sessions (e.g. OSAP, ADIP) may use AuthData as an initialization vector when creating a one-time pad. Session encryption is used to encrypt portions of command messages exchanged between TPM and a caller.

The TPM stores AuthData with TPM controlled-objects and in shielded-locations. AuthData is never in the clear, when managed by the TPM except in shielded-locations. Only TPM protected-capabilities may access AuthData (contained in the TPM). AuthData objects may not be used for any other purpose besides authentication and authorization of TPM operations on controlled-objects.

Outside the TPM, a reference monitor of some kind is responsible for protecting AuthData. AuthData should be regarded as a controlled data item (CDI) in the context of the security model governing the reference monitor. TCG expects this entity to preserve the interests of the platform Owner.

There is no requirement that instances of AuthData be unique.

**End of informative comment**

The TPM MUST reserve 160 bits for the AuthData. The TPM treats the AuthData as a blob. The TPM MUST keep AuthData in a shielded-location.

The TPM MUST enforce that the only usage in the TPM of the AuthData is to perform authorizations.

## 8.1 Dictionary Attack Considerations

**Start of informative comment**

The decision to provide protections against dictionary attacks is due to the inability of the TPM to guarantee that an authorization value has high entropy. While the creation and authorization protocols could change to support the assurance of high entropy values, the changes would be drastic and would totally invalidate any 1.x TPM version.

Version 1.1 explicitly avoided any requirements for dictionary attack mitigation.

Version 1.2 adds the requirement that the TPM vendor provide some assistance against dictionary attacks. The internal mechanism is vendor specific. The TPM designer should review the requirements for dictionary attack mitigation in the Common Criteria.

The 1.2 specification does not provide any functions to turn on the dictionary attack prevention. The specification does provide a way to reset from the TPM response to an attack.

By way of example, the following is a way to implement the dictionary attack mitigation.

The TPM keeps a count of failed authorization attempts. The vendor allows the TPM Owner to set a threshold of failed authorizations. When the count exceeds the threshold, the TPM locks up and does not respond to any requests for a time out period. The time out period doubles each time the count exceeds the threshold. If the TPM resets during a time out period, the time out period starts over after TPM_Init, or TPM_Startup. To reset the count and the time out period the TPM Owner executes TPM_ResetLockValue. If the authorization for TPM_ResetLockValue fails, the TPM must lock up for the entire time out period and no additional attempts at unlocking will be successful. Executing TPM_ResetLockValue when outside of a time out period still results in the resetting of the count and time out period.

**End of informative comment**

The TPM SHALL incorporate mechanism(s) that will provide some protection against exhaustive or dictionary attacks on the authorization values stored within the TPM.

This version of the TPM specification does NOT specify the particular strategy to be used. Some examples might include locking out the TPM after a certain number of failures, forcing a reboot under some combination of failures, or requiring specific actions on the part of some actors after an attack has been detected. The mechanisms to manage these strategies are vendor specific at this time.

If the TPM in response to the attacks locks up for some time period or requires a special operation to restart, the TPM MUST prevent any authorized TPM command and MAY prevent any TPM command from executing until the mitigation mechanism completes. The TPM Owner can reset the mechanism using the TPM_ResetLockValue command. TPM_ResetLockValue MUST be allowed to run exactly once while the TPM is locked up.

# 9. TPM Operation

**Start of informative comment**

Through the course of TPM operation, it may enter several operational modes that include power-up, self-test, administrative modes and full operation. This section describes TPM operational states and state transition criteria. Where applicable, the TPM commands used to facilitate state transition or function are included in diagrams and descriptions.

The TPM keeps the information relative to the TPM operational state in a combination of persistent and volatile flags. For ease of reading the persistent flags are prefixed by pFlags and the volatile flags prefixed by vFlags.

The following state diagram describes TPM operational states at a high level. Subsequent state diagrams drill-down to finer detail that describes fundamental operations, protections on operations and the transitions between them.

The state diagrams use the following notation:

- Signifies a state.

→ - Transitions between states are represented as a single headed arrows.

↻ - Circular transitions indicate operations that don't result in a transition to another state.

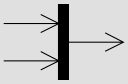
◇ - Decision boxes split state flow based on a logical test. Decision conditions are called Guards and are identified by bracketed text.

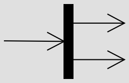
< [text] > Bracketed text indicates transitions that are gated. Text within the brackets describes the pre-condition that must be met before state transition may occur.

< /name > Transitions may list the events that trigger state transition. The forward slash demarcates event names.

● - The starting point for reading state diagrams.

◉ - The ending point for state diagrams. Perpetual state systems may not have an ending indicator.


- The collection bar consolidates multiple identical transition events into a single transition arrow.


- The distribution bar splits transitions to flow into multiple states.

Ⓗ - The history indicator means state values are remembered across context switches or power-cycles.

**End of informative comment**

## 9.1 TPM Initialization & Operation State Flow

**Start of informative comment**

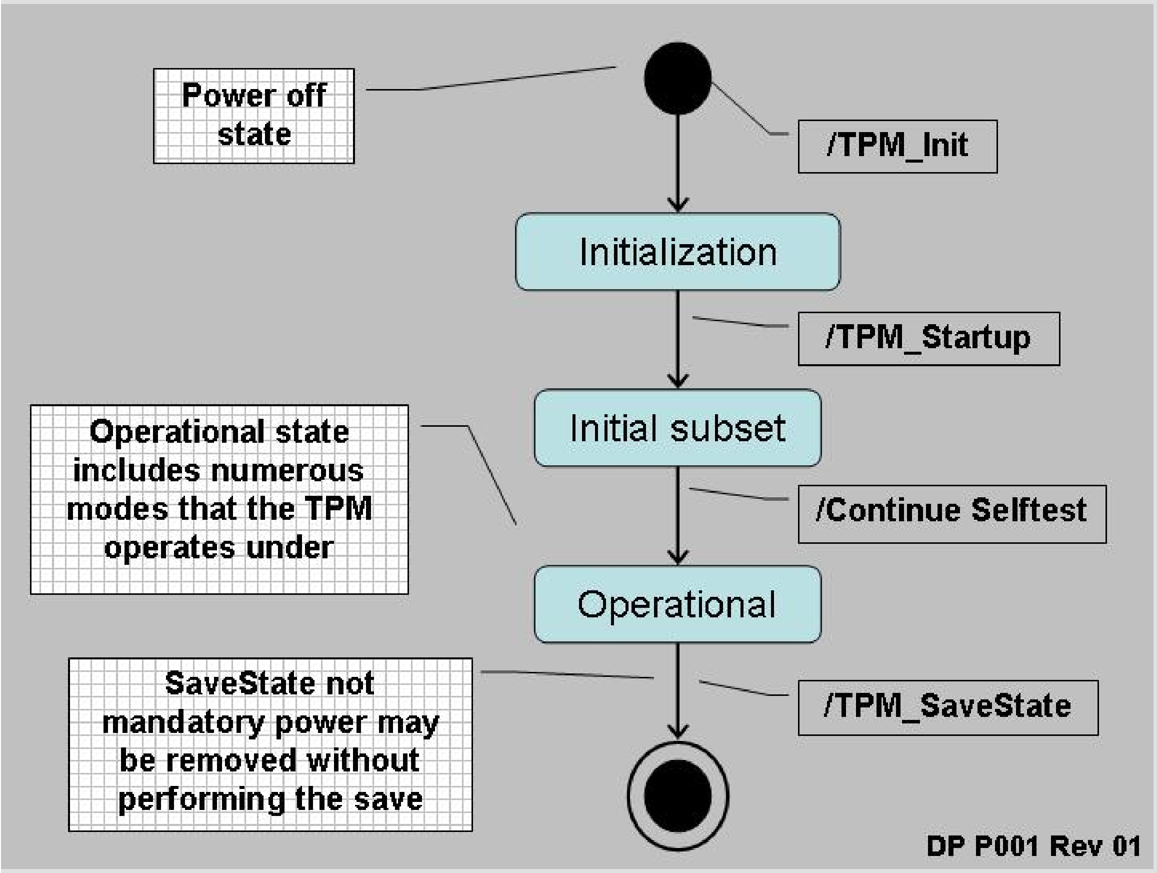

Power off state
/TPM_Init
Initialization
/TPM_Startup
Operational state includes numerous modes that the TPM operates under
Initial subset
/Continue Selftest
Operational
SaveState not mandatory power may be removed without performing the save
/TPM_SaveState
DP P001 Rev 01

**Figure 9:b - TPM Operational States**

**End of informative comment**

### 9.1.1 Initialization

**Start of informative comment**

TPM_Init transitions the TPM from a power-off state to one where the TPM begins an initialization process. TPM_Init could be the result of power being applied to the platform or a hard reset.

TPM_Init sets an internal flag to indicate that the TPM is undergoing initialization. The TPM must complete initialization before it is operational. The completion of initialization requires the receipt of the TPM_Startup command.

The TPM is not fully operational until all of the self-tests are complete. Successful completion of the self-tests allows the TPM to enter fully operational mode.

Fully operational does not imply that all functions of the TPM are available. The TPM needs to have a TPM Owner and be enabled for all functions to be available.

The TPM transitions out of the operational mode by having power removed from the system. Prior to the exiting operational mode, the TPM prepares for the transition by executing the TPM_SaveState command. There is no requirement that TPM_SaveState execute before the transition to power-off mode occurs.

**End of informative comment**

1. After TPM_Init and until receipt of TPM_Startup the TPM MUST return TPM_INVALID_POSTINIT for all commands. Prior to receipt of TPM_Startup the TPM MAY enter shutdown or failure mode.

## 1525 9.2 Self-Test Modes

1526**Start of informative comment**

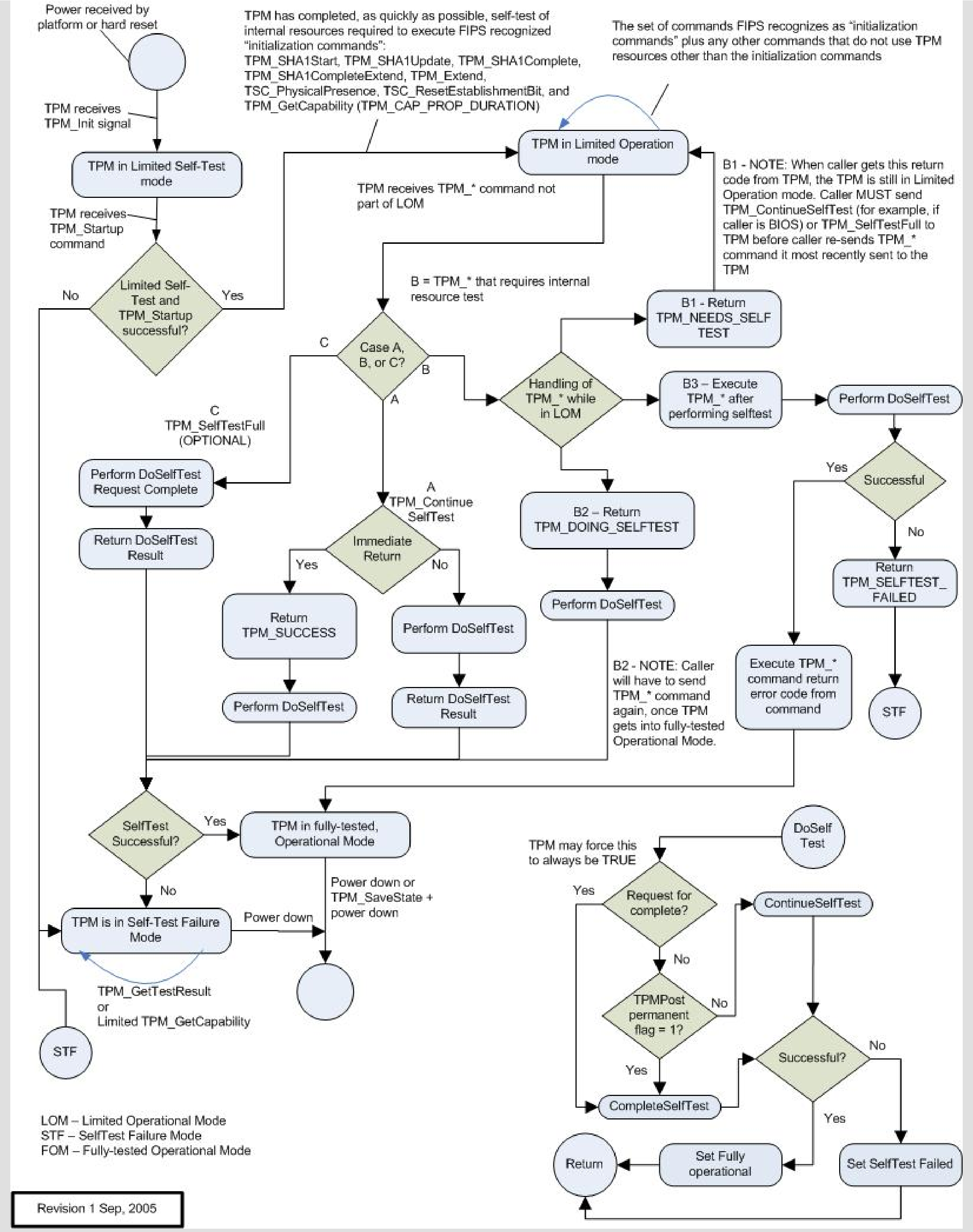

Power received by platform or hard reset
TPM receives TPM_Init signal
TPM in Limited Self-Test mode
TPM receives TPM_Startup command
Limited Self-Test and TPM_Startup successful?
No
Yes
TPM has completed, as quickly as possible, self-test of internal resources required to execute FIPS recognized "initialization commands": TPM_SHA1Start, TPM_SHA1Update, TPM_SHA1Complete, TPM_SHA1CompleteExtend, TPM_Extend, TSC_PhysicalPresence, TSC_ResetEstablishmentBit, and TPM_GetCapability (TPM_CAP_PROP_DURATION)
The set of commands FIPS recognizes as "initialization commands" plus any other commands that do not use TPM resources other than the initialization commands
TPM in Limited Operation mode
TPM receives TPM_* command not part of LOM
B1 - NOTE: When caller gets this return code from TPM, the TPM is still in Limited Operation mode. Caller MUST send TPM_ContinueSelfTest (for example, if caller is BIOS) or TPM_SelfTestFull to TPM before caller re-sends TPM_* command it most recently sent to the TPM
B = TPM_* that requires internal resource test
B1 - Return TPM_NEEDS_SELF TEST
Case A, B, or C?
C
B
A
Handling of TPM_* while in LOM
B3 – Execute TPM_* after performing selftest
Perform DoSelfTest
C TPM_SelfTestFull (OPTIONAL)
Perform DoSelfTest Request Complete
A TPM_Continue SelfTest
B2 – Return TPM_DOING_SELFTEST
Yes
Successful
No
Return DoSelfTest Result
Immediate Return
Yes
No
Return TPM_SUCCESS
Perform DoSelfTest
Perform DoSelfTest
Return TPM_SELFTEST_ FAILED
B2 - NOTE: Caller will have to send TPM_* command again, once TPM gets into fully-tested Operational Mode.
Execute TPM_* command return error code from command
STF
Perform DoSelfTest
Return DoSelfTest Result
SelfTest Successful?
Yes
TPM in fully-tested, Operational Mode
No
TPM is in Self-Test Failure Mode
Power down
Power down or TPM_SaveState + power down
TPM_GetTestResult or Limited TPM_GetCapability
STF
TPM may force this to always be TRUE
DoSelf Test
Yes
Request for complete?
ContinueSelfTest
No
TPMPost permanent flag = 1?
No
Yes
Successful?
No
CompleteSelfTest
Yes
Return
Set Fully operational
Set SelfTest Failed
LOM – Limited Operational Mode
STF – SelfTest Failure Mode
FOM – Fully-tested Operational Mode
Revision 1 Sep, 2005

Figure 9:c - Self-Test States

After initialization the TPM performs a limited self-test. This test provides the assurance that a selected subset of TPM commands will perform properly. The limited nature of the self-test allows the TPM to be functional in as short of time as possible. The commands enabled by this self-test are:

TPM_SHA1xxx – Enabling the SHA-1 commands allows the TPM to assist the platform startup code. The startup code may execute in an extremely constrained memory environment and having the TPM resources available to perform hash functions can allow the measurement of code at an early time. While the hash is available, there are no speed requirements on the I/O bus to the TPM or on the TPM itself so use of this functionality may not meet platform startup requirements.

TPM_Extend – Enabling the extend, and by reference the PCR, allows the startup code to perform measurements. Extending could use the SHA-1 TPM commands or perform the hash using the main processor.

TPM_Startup – This command must be available as it is the transition command from the initial environment to the limited operational state.

TPM_ContinueSelfTest – This command causes the TPM to complete the self-tests on all other TPM functions. If TPM receives a command, and the self-test for that command has not been completed, the TPM may implicitly perform the actions of the TPM_ContinueSelfTest command.

TPM_SelfTestFull – A TPM MAY allow this command after initialization, but typically TPM_ContinueSelfTest would be used to avoid repeating the limited self tests.

TPM_GetCapability – A subset of capabilities can be read in the limited operation state.

TSC_PhysicalPresence and TSC_ResetEstablishmentBit.

The complete self-test ensures that all TPM functionality is available and functioning properly.

**End of informative comment**

1. At startup, a TPM MUST self-test all internal functions that are necessary to do TPM_SHA1Start, TPM_SHA1Update, TPM_SHA1Complete, TPM_SHA1CompleteExtend, TPM_Extend, TPM_Startup, TPM_ContinueSelfTest, a subset of TPM_GetCapability, TPM_GetTestResult, TSC_PhysicalPresence and TSC_ResetEstablishmentBit.
2. The TPM MAY allow TPM_SelfTestFull to be used before completion of the actions of TPM_ContinueSelfTest.
3. The TPM MAY implicitly run the actions of TPM_ContinueSelfTest upon receipt of a command that requires untested resources.
4. The platform specific specification MUST define the maximum startup self-test time.

### 9.2.1 Operational Self-Test

**Start of informative comment**

The completion of self-test is initiated by TPM_ContinueSelfTest. The TPM MAY allow TPM_SelfTestFull to be issued instead of TPM_ContinueSelfTest.

TPM_ContinueSelfTest is the command issued during platform initialization after the platform has made use of the early commands (perhaps for an early measurement), the platform is now performing other initializations, and the TPM can be left alone to complete the self-tests. Before any command other than the limited subset is executed, all self-tests must be complete.

TPM_SelfTestFull is a request to have the TPM perform another complete self-test. This test will take some time but provides an accurate assessment of the TPM's ability to perform all operations.

The original design of TPM_ContinueSelfTest was for the TPM to test those functions that the original startup did not test. The FIPS-140 evaluation of the specification requested a change such that TPM_ContinueSelfTest would perform a complete self-test. The rationale is that the original tests are only part of the initialization of the TPM; if they fail, the TPM does not complete initialization. Performing a complete test after initialization meets the FIPS-140 requirements. The TPM may work differently in FIPS mode or the TPM may simply write the TPM_ContinueSelfTest command such that it always performs the complete check.

TPM_ContinueSelfTest causes a test of the TPM internal functions. When TPM_ContinueSelfTest is asynchronous, the TPM immediately returns a successful result code before starting the tests. When testing is complete, the TPM does not return any result. When TPM_ContinueSelfTest is synchronous, the TPM completes the self-tests and then returns a success or failure result code.

The TPM may reject any command other than the limited subset if self test has not been completed. Alternatively, the actions of TPM_ContinueSelfTest may start automatically if the TPM receives a command and there has been no testing of the underlying functionality. If the TPM implements this implicit self-test, it may immediately return a result code indicating that it is doing self-test. Alternatively, it may do the self-test, then do the command, and return only the result code of the command.

Programmers of TPM drivers should take into account the time estimates for self-test and minimize the polling for self-test completion. While self-test is executing, the TPM may return an out-of-band "busy" signal to prevent command from being issued. Alternatively, the TPM may accept the command but delay execution until after the self-test completes. Either of those alternatives may appear as if the TPM is blocking to upper software layers. Alternatively, the TPM may return an indication that is doing a self-test.

Upon the completion of the self-tests, the result of the self-tests are held in the TPM such that a subsequent call to TPM_GetTestResult returns the self-test result.

In version 1.1, there was a separate command to create a signed self-test, TPM_CertifySelfTest. Version 1.2 deprecates the command. The new use model is to perform TPM_GetTestResult inside of a transport session and then use TPM_ReleaseTransportSigned to obtain the signature.

If self-tests fail, the TPM goes into failure state and does not allow most other operations to continue. The TPM_GetTestResult will operate in failure mode so an outside observer can obtain information as to the reason for the self-test failure.

A TPM may take three courses of action when presented with a command that requires an untested resource.

1. The TPM may return TPM_NEEDS_SELFTEST, indicating that the execution of the command requires TPM_ContinueSelfTest.

2. The TPM may implicitly execute the self-test and return a TPM_DOING_SELFTEST return code, causing the external software to retry the command.
3. The TPM may implicitly execute the self-test, execute the ordinal, and return the results of the ordinal.

The following example shows how software can detect either mechanism with a single piece of code

1. SW sends TPM_xxx command
2. SW checks return code from TPM
3. If return code is TPM_DOING_SELFTEST, SW attempts to resend
    a. If the TIS times out waiting for TPM ready, pause for self-test time then resend
    b. if TIS timeout, then error
4. else if return code is TPM_NEEDS_SELFTEST
    a. Send TPM_ContinueSelfTest
5. else
    a. Process the ordinal return code

**End of informative comment**

1. The TPM MUST provide startup self-tests. The TPM MUST provide mechanisms to allow the self-tests to be run on demand. The response from the self-tests is pass or fail.
2. The TPM MUST complete the startup self-tests in a manner and timeliness that allows the TPM to be of use to the BIOS during the collection of integrity metrics.
3. The TPM MUST complete the required checks before a given feature is in use. If a function self-test is not complete the TPM MUST return TPM_NEEDS_SELFTEST or TPM_DOING_SELFTEST, or do the self-test before using the feature.
4. There are two sections of startup self-tests: required and recommended. The recommended tests are not a requirement due to time constraints. The TPM manufacturer should perform as many tests as possible within the time constraints.
5. The TPM MUST report the result of the tests that it performs.
6. The TPM MUST provide a mechanism to allow self-test to execute on request by any challenger.
7. The TPM MUST provide for testing of some operations during each execution of the operation.
8. The TPM MUST check the following:
    a. RNG functionality
    b. Reading and extending the integrity registers. The self-test for the integrity registers will leave the integrity registers in a known state.
    c. Testing the EK integrity, if it exists

    i. This requirement specifies that the TPM will verify that the endorsement key pair can encrypt and decrypt a known value. This tests the RSA engine. If the EK has not yet been generated the TPM action is manufacturer specific.

  d. The integrity of the protected capabilities of the TPM

    i. This means that the TPM must ensure that its “microcode” has not changed, and not that a test must be run on each function.

  e. Any tamper-resistance markers

    i. The tests on the tamper-resistance or tamper-evident markers are under programmable control. There is no requirement to check tamper-evident tape or the status of epoxy surrounding the case.

9. The TPM MUST check the following:

  a. The hash functionality

    i. This check MAY hash a known value and compare it to an expected result.

  b. Any symmetric algorithms

    i. This check MAY use known data with a random key to encrypt and decrypt the data

  c. Any asymmetric algorithms

    i. This check MAY use known data to encrypt and decrypt.

  d. Any hardware crypto accelerators

10. Self-Test Failure

  a. When the TPM detects a failure during any self-test, the TPM MUST enter shutdown mode. This shutdown mode will allow only the following operations to occur:

    i. Update. The update function MAY replace invalid microcode, providing that the parts of the TPM that provide update functionality have passed self-test.

    ii. TPM_GetTestResult. This command can assist the TPM manufacturer in determining the cause of the self-test failure.

    iii. TPM_GetCapability may return limited information as specified in the ordinal.

    iv. All other operations will return the error code TPM_FAILEDSELFTEST.

  b. The TPM MUST leave failure mode only after receipt of TPM_Init.

  c. When the TPM detects a failure during any self-test, it SHOULD delete values preserved by TPM_SaveState.

11. Prior to the completion of the actions of TPM_ContinueSelfTest the TPM MAY respond in two ways

  a. The TPM MAY automatically invoke the actions of TPM_ContinueSelfTest.

    i. The TPM MAY return TPM_DOING_SELFTEST.

    ii. The TPM may complete the self-test, execute the command, and return the command result.

  b. The TPM MAY return the error code TPM_NEEDS_SELFTEST

## 9.3 Startup

**Start of informative comment**

Startup transitions the TPM from the initialization state to an operational state. The transition includes information from the platform to inform the TPM of the platform operating state. TPM_Startup has three options: Clear, State and Deactivated.

The Clear option informs the TPM that the platform is starting in a "cleared" state or most likely a complete reboot. The TPM is to set itself to the default values and operational state specified by the TPM Owner.

The State option informs the TPM that the platform is requesting the TPM to recover a saved state and continue operation from the saved state. The platform previously made the TPM_SaveState request to the TPM such that the TPM prepares values to be recovered later. If the TPM enters failure mode after TPM_SaveState, the saved state should be deleted. It is then possible that the State option will fail.

The Deactivated state informs the TPM that it should not allow further operations and should fail all subsequent command requests. The Deactivated state can only be reset by performing another TPM_Init.

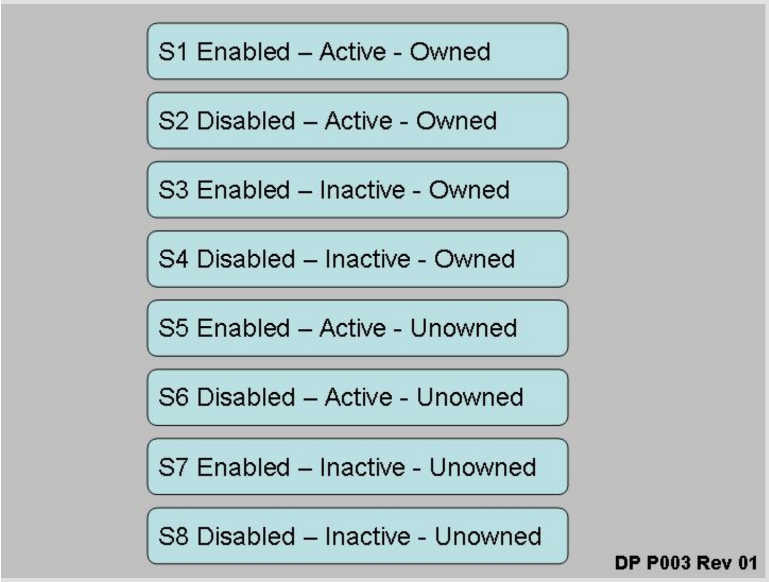
**End of informative comment**

## 9.4 Operational Mode

**Start of informative comment**

After the TPM completes both TPM_Startup and self-tests, the TPM is ready for operation.

There are three discrete states, enabled or disabled, active or inactive and owned or unowned. These three states when combined form eight operational modes.



S1 Enabled – Active - Owned
S2 Disabled – Active - Owned
S3 Enabled – Inactive - Owned
S4 Disabled – Inactive - Owned
S5 Enabled – Active - Unowned
S6 Disabled – Active - Unowned
S7 Enabled – Inactive - Unowned
S8 Disabled – Inactive - Unowned

Figure 9:d - Eight Modes of Operation

S1 is the fully operational state where all TPM functions are available. S8 represents a mode where all TPM features (except those to change the state) are off.

TCG Published

Given the eight modes of operation, the TPM can be flexible in accommodating a wide range of usage scenarios. The default delivery state for a TPM should be S8 (disabled, inactive and unowned). In S8, the only mechanism available to move the TPM to S1 is having physical access to the platform.

Two examples illustrate the possibilities of shipping combinations.

Example 1

The customer does not want the TPM to attest to any information relative to the platform. The customer does not want any remote entity to attempt to change the control options that the platform owner is setting. For this customer the platform manufacturer sets the TPM in S8 (disabled, deactivated and unowned).

To change the state of the platform the platform owner would assert physical presence and enable, activate and insert the TPM Owner shared secret. The details of how to change the various modes is in subsequent sections.

This particular sequence gives maximum control to the customer.

Example 2

A corporate customer wishes to have platforms shipped to their employees and the IT department wishes to take control of the TPM remotely. To satisfy these needs the TPM should be in S5 (enabled, active and unowned). When the platform connects to the corporate LAN the IT department would execute the TPM_TakeOwnership command remotely.

This sequence allows the IT department to accept platforms into their network without having to have physical access to each new machine.

**End of informative comment**

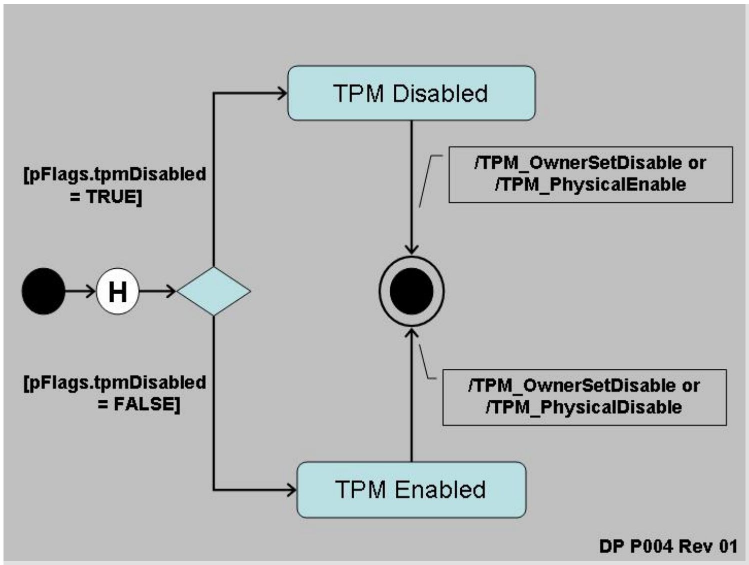
The TPM MUST have commands to perform the following:

1. Enable and disable the TPM. These commands MUST work as TPM Owner authorized or with the assertion of physical presence
2. Activate and deactivate the TPM. These commands MUST work as TPM Owner authorized or with the assertion of physical presence
3. Activate and deactivate the ability to take ownership of the TPM
4. Assert ownership of the TPM.

### 9.4.1 Enabling a TPM

**Informative comment**

A disabled TPM is not able to execute commands that use the resources of a TPM. While some commands are available (SHA-1 for example) the TPM is not able to load keys and perform TPM_Seal and other such operations. These restrictions are the same as for an inactive TPM. The difference between inactive and disabled is that a disabled TPM is unable to execute the TPM_TakeOwnership command. A disabled TPM that has a TPM Owner is not able to execute normal TPM commands.



pFlags.tpmDisabled contains the current enablement status. When set to TRUE the TPM is disabled, when FALSE the TPM is enabled.

Changing the setting pFlags.tpmDisabled has no effect on any secrets or other values held by the TPM. No keys, monotonic counters or other resources are invalidated by changing TPM enablement. There is no guarantee that session resources (like transport sessions) survive the change in enablement, but there is no loss of secrets.

The TPM_OwnerSetDisable command can be used to transition in either Enabled or Disabled states. The desired state is a parameter to TPM_OwnerSetDisable. This command requires TPM Owner authentication to operate. It is suitable for post-boot and remote invocation.

An unowned TPM requires the execution of TPM_PhysicalEnable to enable the TPM and TPM_PhysicalDisable to disable the TPM. Operators of an owned TPM can also execute these two commands. The use of the physical commands allows a platform operator to disable the TPM without TPM Owner authorization.

TPM_PhysicalEnable transitions the TPM from Disabled to Enabled state. This command is guarded by a requirement of operator physical presence. Additionally, this command can be invoked by a physical event at the platform, whether or not the TPM has an Owner or there is a human physically present. This command is suitable for pre-boot invocation.

TPM_PhysicalDisable transitions the TPM from Enabled to Disabled state. It has the same guard and invocation properties as TPM_PhysicalEnable.

The subset of commands the TPM is able to execute is defined in the structures document in the persistent flag section.

Misuse of the disabled state can result in denial-of-service. Proper management of Owner AuthData and physical access to the platform is a critical element in ensuring availability of the system.

**End of informative comment**

1. The TPM MUST provide an enable and disable command that is executed with TPM Owner authorization.

2. The TPM MUST provide an enable and disable command this is executed locally using physical presence.

### 9.4.2 Activating a TPM

**Informative comment**

A deactivated TPM is not able to execute commands that use TPM resources. A major difference between deactivated and disabled is that a deactivated TPM CAN execute the TPM_TakeOwnership command.

Deactivated may be used to prevent the (obscure) attack where a TPM is readied for TPM_TakeOwnership but a remote rogue manages to take ownership of a platform just before the genuine owner, and immediately has use of the TPM's facilities. To defeat this attack, a genuine owner should set disable==FALSE, ownership==TRUE, deactivate==TRUE, execute TPM_takeOwnership, and then set deactivate==FALSE after verifying that the genuine owner is the actual TPM owner.

Activation control is with both persistent and volatile flags. The persistent flag is never directly checked by the TPM, rather it is the source of the original setting for the volatile flag. During TPM initialization the value of pFlags.tpmDeactivated is copied to vFlags.tpmDeactivated. When the TPM execution engine checks for TPM activation, it only references vFlags.tpmDeactivated.

Toggling the state of pFlags.tpmDeactivated uses TPM_PhysicalSetDeactivated. This command requires physical presence. There is no associated TPM Owner authenticated command as the TPM Owner can always execute TPM_OwnerSetDisabled which results in the same TPM operations. The toggling of this flag does not affect the current operation of the TPM but requires a reboot of the platform such that the persistent flag is again copied to the volatile flag.

The volatile flag, vFlags.tpmDeactivated, is set during initialization by the value of pFlags.tpmDeactivated. If vFlags.tpmDeactivated is TRUE the only way to reactivate the TPM is to reboot the platform and have pFlags reset the vFlags value.

If vFlags.tpmDeactivated is FALSE, running TPM_SetTempDeactivated will set vFlags.tpmDeactivated to TRUE and then require a reboot of the platform to reactivate the platform.

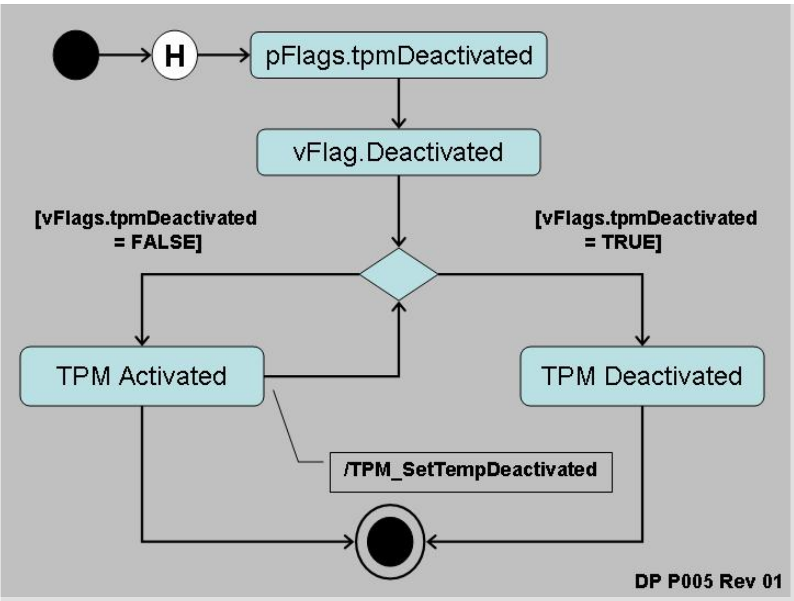


Figure 9:e - Activated and Deactivated States

TPM activation is for Operator convenience. It allows the operator to deactivate the platform (temporarily, using TPM_SetTempDeactivated) during a user session when the operator does not want to disclose platform or attestation identity. This provides operator privacy, since PCRs could provide cryptographic proof of an operation. PCRs are inaccessible when a TPM is deactivated. They cannot be used for authorization, nor can they be read. The reboot required to activate a TPM also resets the PCRs.

The subset of commands that are available when the TPM is deactivated is contained in the structures document. The TPM_TakeOwnership command is available when deactivated. The TPM_Extend command is available when deactivated so that software (e.g. a BIOS) can run the command without the need to handle an error. The PCR extend operation is irrelevant, since the resulting PCR value cannot be used.

**End of informative comment**

1. The TPM MUST maintain a non-volatile flag that indicates the activation state
2. The TPM MUST provide for the setting of the non-volatile flag using a command that requires physical presence
3. The TPM MUST sets a volatile flag using the current setting of the non-volatile flag.
4. The TPM MUST provide for a command that deactivates the TPM immediately
5. The only mechanism to reactivate a TPM once deactivated is to power-cycle the system.

### 9.4.3 Taking TPM Ownership

**Start of informative comment**

The owner of the TPM has ultimate control of the TPM. The owner of the TPM can enable or disable the TPM, create AIK and set policies for the TPM. The process of taking ownership must be a tightly controlled process with numerous checks and balances.

The protections around the taking of ownership include the enablement status, specific persistent flags and the assertion of physical presence.

Control of the TPM revolves around knowledge of the TPM Owner authentication value. Proving knowledge of authentication value proves the calling entity is the TPM Owner. It is possible for more than one entity to know the TPM Owner authentication value.

The TPM provides no mechanisms to recover a lost TPM Owner authentication value.

Recovery from a lost or forgotten TPM Owner authentication value involves removing the old value and installing a new one. The removal of the old value invalidates all information associated with the previous value. Insertion of a new value can occur after the removal of the old value.

A disabled and inactive TPM that has no TPM Owner cannot install an owner.

To invalidate the TPM Owner authentication value use either TPM_OwnerClear or TPM_ForceClear.

**End of informative comment**

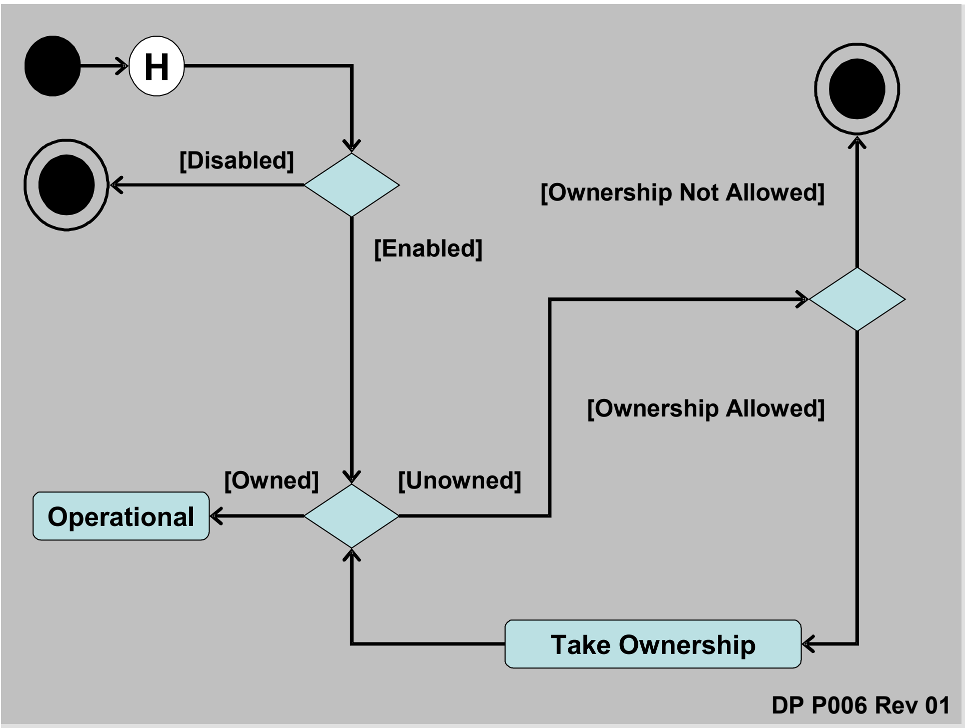
1. The TPM Owner authentication value MUST be a 160-bits
2. The TPM Owner authentication value MUST be held in persistent storage
3. The TPM MUST have no mechanisms to recover a lost TPM Owner authentication value

#### 9.4.3.1 Enabling Ownership

**Informative comment**

For the TPM_TakeOwnership command to succeed, pFlags.disable must be FALSE and pFlags.ownership must be TRUE.

The following diagram shows the states and the operational checks the TPM makes before allowing the insertion of the TPM Ownership value.



The TPM checks pFlags.disable. If the TPM is enabled, the TPM checks for the existence of a TPM Owner. If an Owner is not present, the TPM checks pFlags.ownership. If TRUE, the TPM_TakeOwnership command will execute.

While the TPM has no Owner but is enabled and active, a limited subset of commands will successfully execute.

The TPM_SetOwnerInstall command toggles the state of the pFlags.ownership flag. TPM_SetOwnerInstall requires the assertion of physical presence to execute.

**End of informative comment**

### 9.4.4 Transitioning Between Operational States

**Start of informative comment**

The following table is a recap of the commands necessary to transition a TPM from one state to another.

| State | TPM Owner Auth | Physical Presence | Persistence |
|---|---|---|---|
| Disabled to Enabled | TPM_OwnerSetDisable | TPM_PhysicalEnable | permanent |
| Enabled to Disabled | TPM_OwnerSetDisable | TPM_PhysicalDisable | permanent |
| Inactive to Active | | TPM_PhysicalSetDeactivated | permanent |
| Active to Inactive | | TPM_PhysicalSetDeactivated | permanent |
| Active to Inactive | | TPM_SetTempDeactivated | boot cycle |

**End of informative comment**

## 9.5 Clearing the TPM

**Start of informative comment**

Clearing the TPM is the process of returning the TPM to factory defaults. It is possible the platform owner will change when in this state.

The commands to clear a TPM require either TPM Owner authentication or the assertion of physical presence.

The clear process performs the following tasks:

Invalidate the SRK. Once invalidated all information stored using the SRK is now unavailable. The invalidation does not change the blobs using the SRK rather there is no way to decrypt the blobs after invalidation of the SRK.

Invalidate tpmProof. tpmProof is a value that provides the uniqueness to values stored off of the TPM. By invalidating tpmProof all off TPM blobs will no longer load on the TPM.

Invalidate the TPM Owner authentication value. With the authentication value invalidated there are no TPM Owner authenticated commands that will execute.

Reset volatile and non-volatile data to manufacturer defaults.

The clear must not affect the EK.

Once cleared the TPM will return TPM_NOSRK to commands that require authentication.

The PCR values are undefined after a clear operation. The TPM must go through TPM_Init to properly set the PCR values.

Clear authentication comes from either the TPM owner or the assertion of physical presence. As the clear commands present a real opportunity for a denial of service attack there are mechanisms in place disabling the clear commands.

Disabling TPM_OwnerClear uses the TPM_DisableOwnerClear command. The state of ability to execute TPM_OwnerClear is then held as one of the non-volatile flags.

Enablement of TPM_ForceClear is held in the volatile disableForceClear flag. disableForceClear is set to FALSE during TPM_Init. To disable the command software should issue the TPM_DisableForceClear command.

During the TPM startup processing anyone with physical access to the machine can issue the TPM_ForceClear command. This command performs the clear operations if it has not been disabled by vFlags.DisabledForceClear being TRUE.

The TPM can be configured to block all forms of clear operations. It is advisable to block clear operations to prevent an otherwise trivial denial-of-service attack. The assumption is the system startup code will issue the TPM_DisableForceClear on each power-cycle after it is determined the TPM_ForceClear command will not be necessary. The purpose of the TPM_ForceClear command is to recover from the state where the Owner has lost or forgotten the TPM Owner-authentication-data.

The TPM_ForceClear must only be possible when the issuer has physical access to the platform. The manufacturer of a platform determines the exact definition of physical access.

The commands to clear a TPM require either TPM Owner authentication, TPM_OwnerClear, or the assertion of physical presence, TPM_ForceClear.

**End of informative comment**

1. The TPM MUST support the clear operations.
   a. Clear operations MUST be authenticated by either the TPM Owner or physical presence
   b. The TPM MUST support mechanisms to disable the clear operations
2. The clear operation MUST perform at least the following actions
   a. SRK invalidation
   b. tpmProof invalidation
   c. TPM Owner authentication value invalidation
   d. Resetting non-volatile values to defaults
   e. Invalidation of volatile values
   f. Invalidation of internal resources
3. The clear operation must not affect the EK.

# 10. Physical Presence

**Start of informative comment**

This specification describes commands that require physical presence at the platform before the command will operate. Physical presence implies direct interaction by a person – i.e. Operator with the platform / TPM.

The type of controls that imply special privilege include:

- Clearing an existing Owner from the TPM,
- Temporarily deactivating a TPM,
- Temporarily disabling a TPM.

Physical presence implies a level of control and authorization to perform basic administrative tasks and to bootstrap management and access control mechanisms.

Protection of low-level administrative interfaces can be provided by physical and electrical methods; or by software; or a combination of both. The guiding principle for designers is the protection mechanism should be difficult or impossible to spoof by rogue software. Designers should take advantage of restricted states inherent in platform operation. For example, in a PC, software executed during the power-on self-test (POST) cannot be disturbed without physical access to the platform. Alternatively, a hardware switch indicating physical presence is very difficult to circumvent by rogue software or remote attackers.

TPM and platform manufacturers will determine the actual implementation approach. The strength of the protection mechanisms is determined by an evaluation of the platform.

Physical presence indication is implemented as a flag in volatile memory known as the physicalPresenceV flag. When physical presence is established (TRUE) several TPM commands are able to function. They include:

TPM_PhysicalEnable,

TPM_PhysicalDisable,

TPM_PhysicalSetDeactivated,

TPM_ForceClear,

TPM_SetOwnerInstall,

In order to execute these commands, the TPM must obtain unambiguous assurance that the operation is authorized by physical-presence at the platform. The command processor in the I/O component checks the physicalPresenceV flag before continuing processing of TPM command blocks. The volatile physicalPresenceV flag is set only while the Operator is indeed physically present.

TPM designers should take precautions to ensure testing of the physicalPresenceV flag value is not mask-able. For example, a special bus cycle could be used or a dedicated line implemented.

There is an exception to physical presence semantics that allows a remote entity the ability to assert physical presence when that entity is not physically present. The TSC_PhysicalPresence command is used to change polarity of the physicalPresenceV flag.

Its use is heavily guarded. See sections describing the TPM Opt-In component; and Volatile and Non-volatile memory components.

The following diagram illustrates the flow of logic controlling updates to the physicalPresenceV flag:

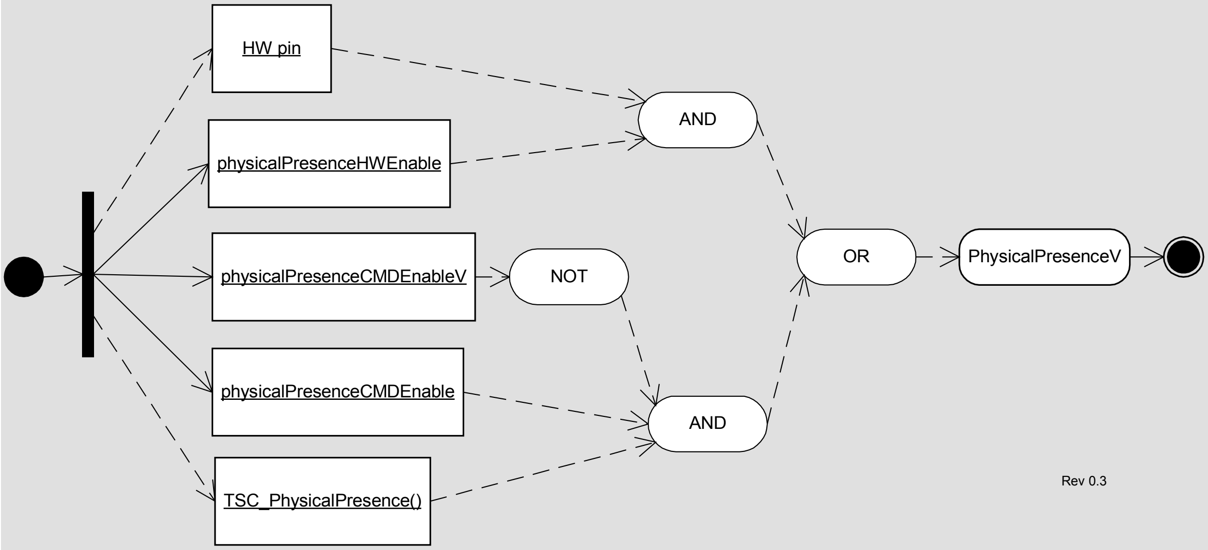


Figure 10:f - Physical Presence Control Logic

This diagram shows that the physicalPresenceV flag may be updated either by a HW pin or through the TSC_PhysicalPresence command, but gated by persistent control flags and a temporal lock. Observe, the reverse logic surrounding the use of TSC_PhysicalPresence command. When the physicalPresenceCMDEnable flag is set and the physicalPresenceLock flag is not set, the TSC_PhysicalPresence command may set physicalPresenceV.

The physicalPresenceV flag may be overridden by unambiguous physical presence. Conceptually, the use of dedicated electrical hardware providing a trusted path to the Operator has higher precedence than the physicalPresenceV flag value. Implementers should consider this when implementing physical presence indicators.

**End of informative comment**

1. The requirement for physical presence MUST be met by the platform manufacturer using some physical mechanism.

2. It SHALL be impossible to intercept or subvert indication of physical presence to the TPM by the execution of software on the platform.

# 11. Root of Trust for Reporting (RTR)

**Start of informative comment**

The RTR is responsible for establishing platform identities, reporting platform configurations, protecting reported values, and providing a function for attesting to reported values. The RTR shares responsibility of protecting measurement digests with the RTS.

The interaction between the RTR and RTS is a critical component. The design and implementation of the interaction between the RTR and RTS should mitigate observation and tampering with the messages. It is strongly encouraged that the RTR and RTS implementation occur in the same package such there are no external observation points. For a silicon based TPM this would imply that the RTR and RTS are in the same silicon package with no external busses.

**End of informative comment**

1. An instantiation of the RTS and RTR SHALL do the following:
   a. Be resistant to all forms of software attack and to the forms of physical attack implied by the platform's Protection Profile
   b. Supply an accurate digest of all sequences of presented integrity metrics

## 11.1 Platform Identity

**Start of informative comment**

The RTR is a cryptographic identity used to distinguish and authenticate an individual TPM. The TPM uses the RTR to answer an integrity challenge.

In the TPM, the Endorsement Key (EK) is the RTR. The EK is cryptographically unique and bound to the TPM.

Prior to any use of the TPM, the RTR must be instantiated. Instantiation may occur during TPM manufacturing or platform manufacturing. The business issues and manufacturing flow determines how a specific TPM and platform is initialized with the EK.

As the RTR is cryptographically unique, the use of the RTR must only occur in controlled circumstances due to privacy concerns. The EK is only available for two operations: establishing the TPM Owner and establishing Attestation Identity Key (AIK) values and credentials. There is a prohibition on the use of the EK for any other operation.

**End of informative comment**

1. The RTR MUST have a cryptographic identity.
   a. The cryptographic identity of the RTR is the Endorsement Key (EK).
2. The EK MUST be
   a. Statistically unique
      i. When the TPM is in FIPS mode, the EK MUST be generated using a random number generator that meets FIPS requirements.
      ii. Difficult to forge or counterfeit
   b. Verifiable during the AIK creation process

3. The EK SHALL only participate in
   a. TPM Ownership insertion
   b. AIK creation and verification

## 11.2 RTR to Platform Binding

**Start of informative comment**

When performing validation of the EK and the platform the challenger wishes to have knowledge of the binding of RTR to platform. The RTR is bound to a TPM hence if the platform can show the binding of TPM to platform the challenger can reasonably believe the RTR and platform binding.

The TPM cannot provide all of the information necessary for the challenger to trust in the binding. That information comes from the manufacturing process and occurs outside the control of the TPM.

**End of informative comment**

1. The EK is transitively bound to the Platform via the TPM as follows:
   a. An EK is bound to one and only one TPM (i.e., there is a one to one correspondence between an Endorsement Key and a TPM.)
   b. A TPM is bound to one and only one Platform. (i.e., there is a one to one correspondence between a TPM and a Platform.)
   c. Therefore, an EK is bound to a Platform. (i.e., there is a one to one correspondence between an Endorsement Key and a Platform.)

## 11.3 Platform Identity and Privacy Considerations

**Start of informative comment**

The uniqueness property of cryptographic identities raises concerns that use of that identity could result in aggregation of activity logs. Analysis of the aggregated activity could reveal personal information that a user of a platform would not otherwise approve for distribution to the aggregators. Both EK and AIK identities have this property.

To counter undesired aggregation, TCG encourages the use of domain specific AIK keys and restricts the use of the EK key. The platform owner controls generation and distribution of AIK public keys.

If a digital signature was performed by the EK, then any entity could track the use of the EK. So use of the EK as a signature is cryptographically sound, but this does not ensure privacy. Therefore, a mechanism to allow verifiers (human or machine) to determine that the TPM really signed the message without using the EK is required.

**End of informative comment**

## 11.4 Attestation Identity Keys

**Start of informative comment**

An Attestation Identity Key (AIK) is an alias for the Endorsement Key (EK). The EK cannot perform signatures for security reasons and due to privacy concerns.

Generation of an AIK can occur anytime after establishment of the TPM Owner. The TPM can create a virtually unlimited number of AIK.

The TPM Owner controls all aspects of the generation and activation of an AIK. The TPM Owner controls any data associated with the AIK. The AIK credential may contain application specific information. The AIK must contain identification such that the TPM can properly enforce the restrictions placed on an AIK.

The AIK is an asymmetric key pair. For interoperability, the AIK is an RSA 2048-bit key. The TPM must protect the private portion of the asymmetric key and ensure that the value is never exposed. The user of an AIK must prove knowledge of the 160-bit AIK authorization value to use the AIK.

An AIK is a signature key, and is never used for encryption. It only signs information generated internally by the TPM. The data could include PCR, other keys and TPM status information. The AIK must never sign arbitrary external data, since it would be possible for an attacker to create a block of data that appears to be a PCR value.

AIK creation involves two TPM commands.

The TPM_MakeIdentity command causes the TPM to generate the AIK key pair. The command also discloses the EK-AIK binding to the service that will issue the AIK credential.

The TPM_ActivateIdentity command unwraps a session key that allows for the decryption of the AIK credential. The session key was encrypted using the PUBEK and requires the PRIVEK to perform the decryption.

Use of the AIK credential is outside of the control of the TPM.

**End of informative comment**

1. The TPM MUST permanently mark an AIK such that, for all subsequent uses of the AIK, the AIK restrictions are enforced.
2. An AIK MUST be:
   a. Statistically unique
   b. Difficult to forge or counterfeit
   c. Verifiable to challengers
3. For interoperability the AIK MUST be
   a. An RSA 2048-bit key
4. The AIK MUST only sign data generated by the TPM

### 11.4.1 AIK Creation

**Start of informative comment**

As the AIK is an alias for the EK. The AIK creation process requires TPM Owner authorization. The process actually requires two TPM Owner authorizations; creation and credential activation.

The AIK credential creation process is outside the control of the TPM. However, the certification authority (CA) will attest (with the AIK credential) that the AIK is tied to valid Endorsement, Platform and Conformance credentials.

Without these credentials, the AIK cannot prove that PCR values belong to a TPM. An owner may decide to trust any key generated by TPM_MakeIdentity without activating the identity (e.g., because he is an administrator in a controlled company environment). In this case, the owner needs no credential. Another challenger can only trust that the AIK belongs to a TPM by seeing the credential of a trustworthy CA.

**End of informative comment**

1. The TPM Owner MUST authorize the AIK creation process.
2. The TPM MUST use a protected function to perform the AIK creation.
3. The TPM Owner MUST indicate the entity that will provide the AIK credential as part of the AIK creation process.
4. The TPM Owner MAY indicate that NO credential will ever be created. If the TPM Owner does indicate that no credential will be provided the TPM MUST ensure that no credential can be created.
5. The TTP MAY apply policies to determine if the presented AIK should be granted a credential.
6. The credential request package MUST be useable by only the Privacy CA selected by the TPM Owner.
7. The AIK credential MUST be only obtainable by the TPM that created the AIK credential request.

### 11.4.2 AIK Storage

**Start of informative comment**

The AIK may be stored on some general-purpose storage device.

When held outside of the TPM the AIK sensitive data must be encrypted and integrity protected.

**End of informative comment**

1. When held outside of the TPM AIK encryption and integrity protection MUST protect the AIK sensitive information
2. The migration of AIK from one TPM to another MUST be prohibited

# 12. Root of Trust for Storage (RTS)

**Start of informative comment**

The RTS provides protection on data in use by the TPM but held in external storage devices. The RTS provides confidentiality and integrity for the external blobs.

The RTS also provides the mechanism to ensure that the release of information only occurs in a named environment. The naming of an environment uses the PCR selection to enumerate the values.

Data protected by the RTS can migrate to other TPM.

**End of informative comment**

1. The number and size of values held by the RTS SHOULD be limited only by the volume of storage available on the platform
2. The TPM MUST ensure that TPM_PERMANENT_DATA -> tpmProof is only inserted into TPM internally generated and non-migratable information.

## 12.1 Loading and Unloading Blobs

**Start of informative comment**

The TPM provides several commands to store and load RTS controlled data.

| | Class | Command | Analog | Comment |
|---|---|---|---|---|
| 1 | Data / Internal / TPM | TPM_MakeIdentity | TPM_ActivateIdentity | Special purpose data |
| 2 | Data / External / TPM | TSS_Bind | TPM_Unbind | |
| 3 | Data / Internal / PCR | TPM_Seal | TPM_Unseal | |
| 4 | Data / External / PCR | | | |
| 5 | Key / Internal / TPM | TPM_CreateWrapKey | TPM_LoadKey | |
| 6 | Key / External / TPM | TSS_WrapKey | TPM_LoadKey | |
| 7 | Key / Internal / PCR | | | |
| 8 | Key / External / PCR | TSS_WrapKeyToPcr | TPM_LoadKey | |

# 13. Transport Sessions and Authorization Protocols

**Start of informative comment**

The purpose of the authorization protocols and mechanisms is to prove to the TPM that the requestor has permission to perform a function and use some object. The proof comes from the knowledge of a shared secret.

AuthData is available for the TPM Owner and each entity (keys, for example) that the TPM controls. The AuthData for the TPM Owner and the SRK are held within the TPM itself and the AuthData for other entities are held with the entity.

The TPM Owner AuthData allows the Owner to prove ownership of the TPM. Proving ownership of the TPM does not immediately allow all operations – the TPM Owner is not a "super user" and additional AuthData must be provided for each entity or operation that has protection.

The TPM treats knowledge of the AuthData as complete proof of ownership of the entity. No other checks are necessary. The requestor (any entity that wishes to execute a command on the TPM or use a specific entity) may have additional protections and requirements where he or she (or it) saves the AuthData; however, the TPM places no additional requirements.

There are three protocols to securely pass a proof of knowledge of AuthData from requestor to TPM; the "Object-Independent Authorization Protocol" (OIAP), the "Object-Specific Authorization Protocol" (OSAP) and the "Delegate-Specific Authorization Protocol" (DSAP). The OIAP supports multiple authorization sessions for arbitrary entities. The OSAP supports an authentication session for a single entity and enables the confidential transmission of new authorization information. The DSAP supports the delegation of owner or entity authorization.

New authorization information is inserted by the "AuthData Insertion Protocol" (ADIP) during the creation of an entity. The "AuthData Change Protocol" (ADCP) and the "Asymmetric Authorization Change Protocol" (AACP) allow the changing of the AuthData for an entity. The protocol definitions allow expansion of protocol types to additional TCG required protocols and vendor specific protocols.

The protocols use a "rolling nonce" paradigm. This requires that a nonce from one side be in use only for a message and its reply. For instance, the TPM would create a nonce and send that on a reply. The requestor would receive that nonce and then include it in the next request. The TPM would validate that the correct nonce was in the request and then create a new nonce for the reply. This mechanism is in place to prevent replay attacks and man-in-the-middle attacks.

The basic protocols do not provide long-term protection of AuthData that is the hash of a password or other low-entropy entities. The TPM designer and application writer must supply additional protocols if protection of these types of data is necessary.

The design criterion of the protocols is to allow for ownership authentication, command and parameter authentication and prevent replay and man-in-the-middle attacks.

The passing of the AuthData, nonces and other parameters must follow specific guidelines so that commands coming from different computer architectures will interoperate properly.

**End of informative comment**

1. AuthData MUST use one of the following protocols
   a. OIAP
   b. OSAP
   c. DSAP
2. Entity creation MUST use one of the following protocols
   a. ADIP
3. Changing AuthData MUST use one of the following protocols
   a. ADCP
   b. AACP
4. The TPM MAY support additional protocols to authenticate, insert and change AuthData.
5. When a command has more than one AuthData value
   a. Each AuthData MUST use the same SHA-1 of the parameters
6. Keys MAY specify authDataUsage -> TPM_AUTH_NEVER
   a. If the caller changes the tag from TPM_TAG_RQU_AUTH1_xxx to TPM_TAG_RQU_XXX the TPM SHALL ignore the AuthData values
   b. If the caller leaves the tag as TPM_TAG_RQU_AUTH1
      i. The TPM will compute the AuthData based on the value store in the AuthData location within the key, IGNORING the state of the AuthDataUsage flag.
   c. Users may choose to use a well-known value for the AuthData when setting AuthDataUsage to TPM_AUTH_NEVER.
   d. If a key has AuthDataUsage set to TPM_AUTH_ALWAYS but is received in a command with the tag TPM_TAG_RQU_COMMAND, the command MUST return an error code.
7. For commands that normally have 2 authorization sessions, if the tag specifies only one in the parameter array, then the first session listed is ignored (authDataUsage must be TPM_AUTH_NEVER for this key) and the incoming session data is used for the second auth session in the list.
8. Keys MAY specify AuthDataUsage -> TPM_NO_READ_PUBKEY_AUTH
   a. If the key used in a command to read the public portion of the key (e.g. TPM_CertifyKey, TPM_GetPubKey)
      i. If the caller changes the tag from TPM_TAG_RQU_AUTH1_xxx to TPM_TAG_RQU_XXX, the TPM SHALL ignore the AuthData values
      ii. If the caller leaves the tag as TPM_TAG_RQU_AUTH1, the TPM will compute the AuthData based on the value store in the AuthData location within the key, IGNORING the state of the AuthDataUsage flag
   b. else if the key used in command to read/access the private portion of the key(e.g. TPM_Sign)

i. If the tag is TPM_TAG_RQU_COMMAND, the command MUST return an error code.

## 13.1 Authorization Session Setup

**Start of informative comment**

The TPM provides two protocols for authorizing the use of entities without revealing the AuthData on the network or the connection to the TPM. In both cases, the protocol exchanges nonce-data so that both sides of the transaction can compute a hash using shared secrets and nonce-data. Each side generates the hash value and can compare to the value transmitted. Network listeners cannot directly infer the AuthData from the hashed objects sent over the network.

The first protocol is the Object-Independent Authorization Protocol (OIAP), which allows the exchange of nonces with a specific TPM. Once an OIAP session is established, its nonces can be used to authorize the use of any entity managed by the TPM. The session can live indefinitely until either party requests the session termination. The TPM_OIAP function starts the OIAP session.

The second protocol is the Object Specific Authorization Protocol (OSAP). The OSAP allows establishment of an authentication session for a single entity. The session creates nonces that can authorize multiple commands without additional session-establishment overhead, but is bound to a specific entity. The TPM_OSAP command starts the OSAP session. The TPM_OSAP specifies the entity to which the authorization is bound.

Most commands allow either form of authorization protocol. In general, however, the OIAP is preferred – it is more generally useful because it allows usage of the same session to provide authorization for different entities. The OSAP is, however, necessary for operations that set or reset AuthData.

OIAP sessions were designed for reasons of efficiency; only one setup process is required for potentially many authorizations.

An OSAP session is doubly efficient because only one setup process is required for potentially many authorization calculations and the entity AuthData secret is required only once. This minimizes exposure of the AuthData secret and can minimize human interaction in the case where a person supplies the AuthData information. The disadvantage of the OSAP is that a distinct session needs to be setup for each entity that requires authorization. The OSAP creates an ephemeral secret that is used throughout the session instead of the entity AuthData secret. The ephemeral secret can be used to provide confidentiality for the introduction of new AuthData during the creation of new entities. Termination of the OSAP occurs in two ways. Either side can request session termination (as usual) but the TPM forces the termination of an OSAP session after use of the ephemeral secret for the introduction of new AuthData.

For both the OSAP and the OIAP, session setup is independent of the commands that are authorized. In the case of OIAP, the requestor sends the TPM_OIAP command, and with the response generated by the TPM, can immediately begin authorizing object actions. The OSAP is very similar, and starts with the requestor sending a TPM_OSAP operation, naming the entity to which the authorization session should be bound.

The DSAP session is to provide delegated authorization information.

All session types use a “rolling nonce” paradigm. This means that the TPM creates a new nonce value each time the TPM receives a command using the session.

Example OIAP and OSAP sessions are used to illustrate session setup and use. The fictitious command named TPM_Example occupies the place where an ordinary TPM command might be used, but does not have command specific parameters. The session connects to a key object within the TPM. The key contains AuthData that will be used to secure the session.

There could be as many as 2 authorization sessions applied to the execution of a single TPM command or as few as 0. The number of sessions used is determined by TCG 1.2 Command Specification and is indicated by the command ordinal parameter.

It is also possible to secure authorization sessions using ephemeral shared-secrets. Rather than using AuthData contained in the stored object (e.g. key), the AuthData is supplied as a parameter to OSAP session creation. In the examples below the key.usageAuth parameter is replaced by the ephemeral secret.

**End of informative comment**

## 13.2 Parameter Declarations for OIAP and OSAP Examples

**Start of informative comment**

To follow OIAP and OSAP protocol examples (Table 13:d and ), the reader should become familiar with the parameters declared in Table 13:b and Table 13:c.

Several conventions are used in the parameter tables that may facilitate readability.

The Param column (Table 13:b) identifies the sequence in which parameters are packaged into a command or response message as well as the size in bytes of the parameter value. If this entry in the row is blank, that parameter is not included in the message. <> in the size column means that the size of the element is variable. It is defined either explicitly by the preceding parameter, or implicitly by the parameter type.

The HMAC column similarly identifies the parameters that are included in HMAC calculations. This column also indicates the default parameters that are included in the audit log. Exceptions are noted under the specific ordinal, e.g. TPM_ExecuteTransport.

The HMAC # column details the parameters used in the HMAC calculation. Parameters 1S, 2S, etc. are concatenated and hashed to inParamDigest or outParamDigest, implicitly called 1H1 and possibly 1H2 if there are two authorization sessions. For the first session, 1H1, 2H1, 3H1, and 4H1 are concatenated and HMAC'ed. For the second session, 1H2, 2H2, 3H2, and 4H2 are concatenated and HMAC'ed.

In general, key handles are not included in HMAC calculations. This allows a lower software layer to map the physical handle value generated by the TPM to a logical value used by an upper software layer. The upper layer generally holds the HMAC key and generates the HMAC. Excluding the key handle allows the mapping to occur without breaking the HMAC. It is important to use a different authorization secret for each key to prevent a man-in-the-middle from altering the key handle.

The Type column identifies the TCG data type corresponding to the passed value. An encapsulation of the parameter type is not part of the command message.

The Name column is a fictitious variable name that aids in following the examples and descriptions.

The double-lined row separator distinguishes authorization session parameters from command parameters. In Table 13:b the TPM_Example command has three parameters; keyHandle, inArgOne and inArgTwo. The tag, paramSize and ordinal parameters are message header values describing contents of a command message. The parameters below the double-lined row are OIAP / OSAP /DSAP or transport authorization session related. If a second authorization session were used, the table would show a second authorization section delineated by a second double-lined row. The authorization session parameters identify shared-secret values, session nonces, session digest and flags.

In this example, a single authorization session is used signaled by the TPM_TAG_RQU_AUTH1_COMMAND tag.

For an OIAP or transport session, the TPM_AUTHDATA description column specifies the HMAC key.

For an OSAP or DSAP session, the HMAC key is the shared secret that was calculated during the session setup, not the key specified in the description.  The key specified in the description was previously used in the shared secret calculation.

| Param | | HMAC | | Type | Name | Description |
|---|---|---|---|---|---|---|
| # | Sz | # | Sz | | | |
| 1 | 2 | | | TPM_TAG | tag | TPM_TAG_RQU_AUTH1_COMMAND |
| 2 | 4 | | | UINT32 | paramSize | Total number of input bytes including paramSize and tag |
| 3 | 4 | 1S | 4 | TPM_COMMAND_CODE | ordinal | Command ordinal, fixed value of TPM_Example |
| 4 | 4 | | | TPM_KEY_HANDLE | keyHandle | Handle of a loaded key. |
| 5 | 1 | 2S | 1 | BOOL | inArgOne | The first input argument |
| 6 | 20 | 3S | 20 | UNIT32 | inArgTwo | The second input argument. |
| 7 | 4 | | | TPM_AUTHHANDLE | authHandle | The authorization handle used for keyHandle authorization. |
| | | 2H1 | 20 | TPM_NONCE | authLastNonceEven | Even nonce previously generated by TPM to cover inputs |
| 8 | 20 | 3 H1 | 20 | TPM_NONCE | nonceOdd | Nonce generated by system associated with authHandle |
| 9 | 1 | 4 H1 | 1 | BOOL | continueAuthSession | The continue use flag for the authorization handle |
| 10 | 20 | | | TPM_AUTHDATA | inAuth | The AuthData digest for inputs and keyHandle. HMAC key: key.usageAuth. |

**Table 13:b - Authorization Protocol Input Parameters**

| Param | | HMAC | | Type | Name | Description |
|---|---|---|---|---|---|---|
| # | Sz | # | Sz | | | |
| 1 | 2 | | | TPM_TAG | Tag | TPM_TAG_RSP_AUTH1_COMMAND |
| 2 | 4 | | | UINT32 | paramSize | Total number of output bytes including paramSize and tag |
| 3 | 4 | 1S | 4 | TPM_RESULT | returnCode | The return code of the operation. See section 4.3. |
| | | 2S | 4 | TPM_COMMAND_CODE | ordinal | Command ordinal, fixed value of TPM_Example |
| 4 | 4 | 3S | 4 | UINT32 | outArgOne | Output argument |
| 5 | 20 | 2 H1 | 20 | TPM_NONCE | nonceEven | Even nonce newly generated by TPM to cover outputs |
| | | 3 H1 | 20 | TPM_NONCE | nonceOdd | Nonce generated by system associated with authHandle |
| 6 | 1 | 4 H1 | 1 | BOOL | continueAuthSession | Continue use flag, TRUE if handle is still active |
| 7 | 20 | | | TPM_AUTHDATA | resAuth | The AuthData digest for the returned parameters. HMAC key: key.usageAuth. |

**Table 13:c - Authorization Protocol Output Parameters**

**End of informative comment**

### 13.2.1 Object-Independent Authorization Protocol (OIAP)

**Start of informative comment**

The purpose of this section is to describe the authorization-related actions of a TPM when it receives a command that has been authorized with the OIAP protocol. OIAP uses the TPM_OIAP command to create the authorization session.

Many commands use OIAP authorization. The following description is therefore necessarily abstract. A fictitious TPM command, TPM_Example is used to represent ordinary TPM commands.

Assume that a TPM user wishes to send command TPM_Example. This is an authorized command that uses the key denoted by keyHandle. The user must know the AuthData for keyHandle (key.usageAuth) as this is the entity that requires authorization and this secret is used in the authorization calculation. Let us assume for this example that the caller of TPM_Example does not need to authorize the use of keyHandle for more than one command. This use model points to the selection of the OIAP as the authorization protocol.

For the TPM_Example command, the inAuth parameter provides the authorization to execute the command. The following table shows the commands executed, the parameters created and the wire formats of all of the information.

<inParamDigest> is the result of the following calculation: SHA1(ordinal, inArgOne, inArgTwo). <outParamDigest> is the result of the following calculation: SHA1(returnCode, ordinal, outArgOne). inAuthSetupParams refers to the following parameters, in this order: authLastNonceEven, nonceOdd, continueAuthSession. OutAuthSetupParams refers to the following parameters, in this order: nonceEven, nonceOdd, continueAuthSession

There are two even nonces used to execute TPM_Example, the one generated as part of the TPM_OAIP command (labeled authLastNonceEven below) and the one generated with the output arguments of TPM_Example (labeled as nonceEven below).

| Caller | On the wire | Dir | TPM |
|---|---|---|---|
| Send TPM_OIAP | TPM_OIAP | → | Create session<br>Create authHandle<br>Associate session and authHandle<br>Generate authLastNonceEven<br>Save authLastNonceEven with authHandle |
| Save authHandle, authLastNonceEven | authHandle, authLastNonceEven | ← | Returns |
| Generate nonceOdd<br>Compute inAuth = HMAC (key.usageAuth, inParamDigest, inAuthSetupParams)<br>Save nonceOdd with authHandle | | | |
| Send TPM_Example | tag<br>paramSize<br>ordinal<br>keyHandle<br>inArgOne<br>inArgTwo<br>authHandle<br>nonceOdd<br>continueAuthSession<br>inAuth | → | TPM retrieves key.usageAuth (key must have been previously loaded)<br>Verify authHandle points to a valid session, mismatch returns TPM_E_INVALIDAUTH<br>Retrieve authLastNonceEven from internal session storage<br>HM = HMAC (key.usageAuth, inParamDigest, inAuthSetupParams)<br>Compare HM to inAuth. If they do not compare return with TPM_E_INVALIDAUTH<br>Execute TPM_Example and create returnCode<br>Generate nonceEven to replace authLastNonceEven in session<br>Set resAuth = HMAC( key.usageAuth, outParamDigest, outAuthSetupParams) |
| Save nonceEven<br>HM = HMAC( key.usageAuth, outParamDigest, outAuthSetupParams)<br>Compare HM to resAuth. This verifies returnCode and output parameters. | tag<br>paramSize<br>returnCode<br>outArgOne<br>nonceEven<br>continueAuthSession<br>resAuth | ← | Return output parameters<br>If continueAuthSession is FALSE then destroy session |

Suppose now that the TPM user wishes to send another command using the same session. For the purposes of this example, we will assume that the same example command is used (ordinal = TPM_Example). However, a different key (newKey) with its own secret (newKey.usageAuth) is to be operated on. To re-use the previous session, the continueAuthSession output boolean must be TRUE.

The previous example shows the command execution reusing an existing authorization session. The parameters created and the wire formats of all of the information.

In this case, authLastNonceEven is the nonceEven value returned by the TPM with the output parameters from the first protocol example.

| Caller | On the wire | Dir | TPM |
|---|---|---|---|
| Generate nonceOdd<br>Compute inAuth = HMAC (newKey.usageAuth, inParamDigest, inAuthSetupParams)<br>Save nonceOdd with authHandle | | | |
| Send TPM_Example | tag<br>paramSize<br>ordinal<br>keyHandle<br>inArgOne<br>inArgTwo<br>nonceOdd<br>continueAuthSession<br>inAuth | → | TPM retrieves newKey.usageAuth (newKey must have been previously loaded)<br>Retrieve authLastNonceEven from internal session storage<br>HM = HMAC (newKey.usageAuth, inParamDigest, inAuthSetupParams)<br>Compare HM to inAuth. If they do not compare return with TPM_E_INVALIDAUTH<br>Execute TPM_Example and create returnCode<br>Generate nonceEven to replace authLastNonceEven in session<br>Set resAuth = HMAC(newKey.usageAuth, outParamDigest, outAuthSetupParams) |
| Save nonceEven<br>HM = HMAC( newKey.usageAuth, outParamDigest, outAuthSetupParams)<br>Compare HM to resAuth. This verifies returnCode and output parameters. | tag<br>paramSize<br>returnCode<br>outArgOne<br>nonceEven<br>continueAuthSession<br>resAuth | ← | Return output parameters<br>If continueAuthSession is FALSE then destroy session |

The TPM user could then use the session for further authorization sessions. Suppose, however, that the TPM user no longer requires the authorization session. There are three possibilities in this case:

The user issues a TPM_Terminate_Handle command to the TPM (section 5.3).

The input argument continueAuthSession can be set to FALSE for the last command. In this case, the output continueAuthSession value will be FALSE.

In some cases, the TPM automatically terminates the authorization session regardless of the input value of continueAuthSession. In this case as well, the output continueAuthSession value will be FALSE.

When an authorization session is terminated for any reason, the TPM invalidates the session’s handle and terminates the session’s thread (releases all resources allocated to the session).

**End of informative comment**

## OIAP Actions

1. The TPM MUST verify that the authorization handle (H, say) referenced in the command points to a valid session. If it does not, the TPM returns the error code TPM_INVALID_AUTHHANDLE

2. The TPM SHALL retrieve the latest version of the caller’s nonce (nonceOdd) and continueAuthSession flag from the input parameter list, and store it in internal TPM memory with the authSession ‘H’.

3. The TPM SHALL retrieve the latest version of the TPM’s nonce stored with the authorization session H (authLastNonceEven) computed during the previously executed command.

4. The TPM MUST retrieve the secret AuthData (SecretE, say) of the target entity. The entity and its secret must have been previously loaded into the TPM.

5. The TPM SHALL perform a HMAC calculation using the entity secret data, ordinal, input command parameters and authorization parameters according to previously specified normative regarding HMAC calculation.

6. The TPM SHALL compare HM to the AuthData value received in the input parameters. If they are different, the TPM returns the error code TPM_AUTHFAIL if the authorization session is the first session of a command, or TPM_AUTH2FAIL if the authorization session is the second session of a command. Otherwise, the TPM executes the command which (for this example) produces an output that requires authentication.

7. The TPM SHALL generate a nonce (nonceEven).

8. The TPM creates an HMAC digest to authenticate the return code, return values and authorization parameters to the same entity secret according to previously specified normative regarding HMAC calculation.

9. The TPM returns the return code, output parameters, authorization parameters and AuthData digest.

10. If the output continueUse flag is FALSE, then the TPM SHALL terminate the session. Future references to H will return an error.

TCG Published

### 13.2.2 Object-Specific Authorization Protocol (OSAP)

**Start of informative comment**

This section describes the actions of a TPM when it receives a TPM command via OSAP session. Many TPM commands may be sent to the TPM via an OSAP session. Therefore, the following description is necessarily abstract.

The OSAP session is initialized through the creation of an ephemeral secret which is used to protect session traffic. Sessions are created using the TPM_OSAP command. This section illustrates OSAP using a fictitious command called TPM_Example.

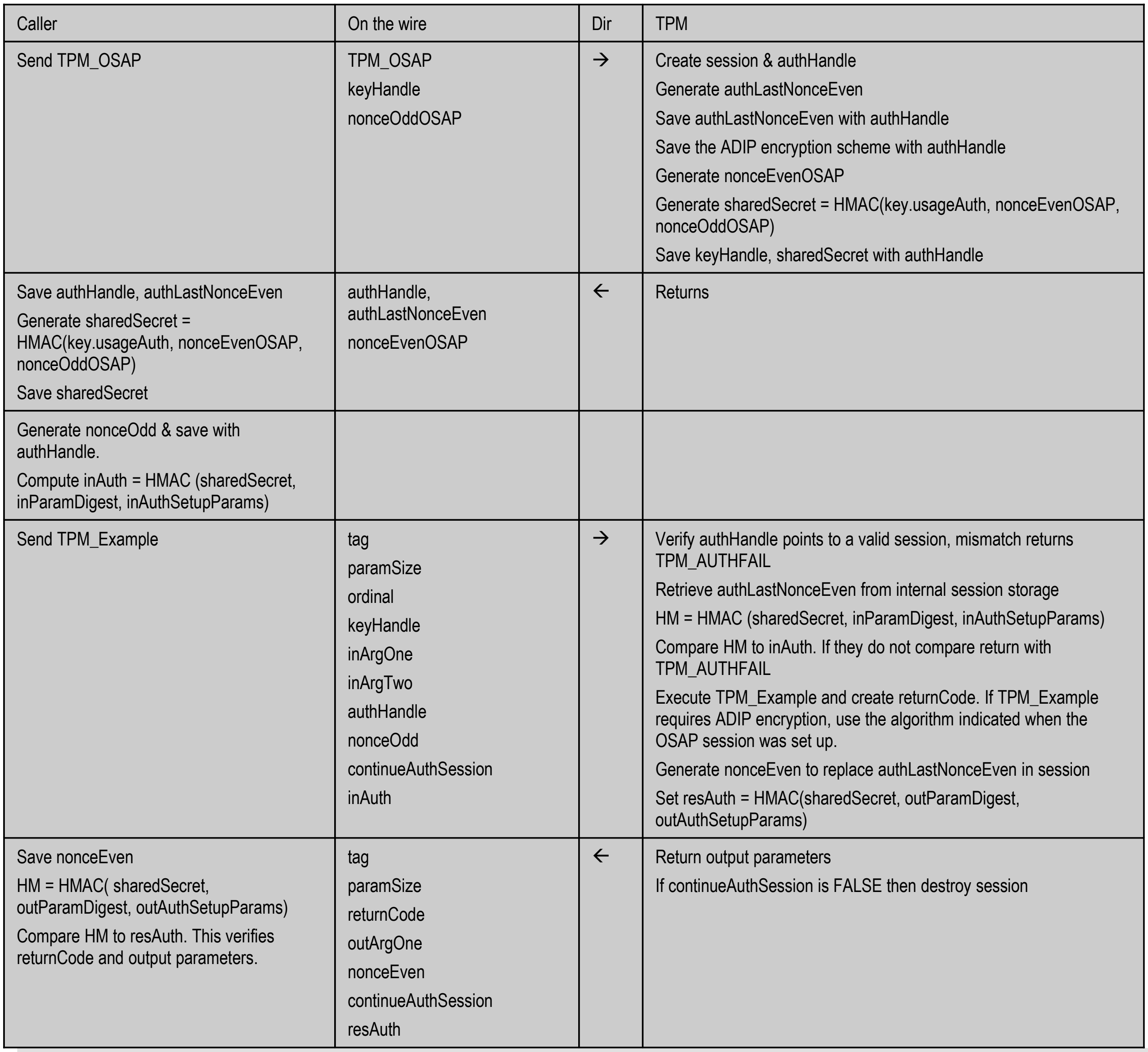
Assume that a TPM user wishes to send the TPM_Example command to the TPM. The keyHandle signifies that an OSAP session is being used and has the value “Auth1”. The user must know the AuthData for keyHandle (key.usageAuth) as this is the entity that requires authorization and this secret is used in the authorization calculation.

Let us assume that the sender needs to use this key multiple times but does not wish to obtain the key secret more than once. This might be the case if the usage AuthData were derived from a typed password. This use model points to the selection of the OSAP as the authorization protocol.

For the TPM_Example command, the inAuth parameter provides the authorization to execute the command. The following table shows the commands executed, the parameters created and the wire formats of all of the information.

<inParamDigest> is the result of the following calculation: SHA1(ordinal, inArgOne, inArgTwo). <outParamDigest> is the result of the following calculation: SHA1(returnCode, ordinal, outArgOne). inAuthSetupParams refers to the following parameters, in this order: authLastNonceEven, nonceOdd, continueAuthSession. OutAuthSetupParams refers to the following parameters, in this order: nonceEven, nonceOdd, continueAuthSession

In addition to the two even nonces generated by the TPM (authLastNonceEven and nonceEven) that are used for TPM_OIAP, there is a third, labeled nonceEvenOSAP that is used to generate the shared secret. For every even nonce, there is also an odd nonce generated by the system.



| Caller | On the wire | Dir | TPM |
|---|---|---|---|
| Send TPM_OSAP | TPM_OSAP<br>keyHandle<br>nonceOddOSAP | → | Create session & authHandle<br>Generate authLastNonceEven<br>Save authLastNonceEven with authHandle<br>Save the ADIP encryption scheme with authHandle<br>Generate nonceEvenOSAP<br>Generate sharedSecret = HMAC(key.usageAuth, nonceEvenOSAP, nonceOddOSAP)<br>Save keyHandle, sharedSecret with authHandle |
| Save authHandle, authLastNonceEven<br>Generate sharedSecret = HMAC(key.usageAuth, nonceEvenOSAP, nonceOddOSAP)<br>Save sharedSecret | authHandle, authLastNonceEven<br>nonceEvenOSAP | ← | Returns |
| Generate nonceOdd & save with authHandle.<br>Compute inAuth = HMAC (sharedSecret, inParamDigest, inAuthSetupParams) | | | |
| Send TPM_Example | tag<br>paramSize<br>ordinal<br>keyHandle<br>inArgOne<br>inArgTwo<br>authHandle<br>nonceOdd<br>continueAuthSession<br>inAuth | → | Verify authHandle points to a valid session, mismatch returns TPM_AUTHFAIL<br>Retrieve authLastNonceEven from internal session storage<br>HM = HMAC (sharedSecret, inParamDigest, inAuthSetupParams)<br>Compare HM to inAuth. If they do not compare return with TPM_AUTHFAIL<br>Execute TPM_Example and create returnCode. If TPM_Example requires ADIP encryption, use the algorithm indicated when the OSAP session was set up.<br>Generate nonceEven to replace authLastNonceEven in session<br>Set resAuth = HMAC(sharedSecret, outParamDigest, outAuthSetupParams) |
| Save nonceEven<br>HM = HMAC( sharedSecret, outParamDigest, outAuthSetupParams)<br>Compare HM to resAuth. This verifies returnCode and output parameters. | tag<br>paramSize<br>returnCode<br>outArgOne<br>nonceEven<br>continueAuthSession<br>resAuth | ← | Return output parameters<br>If continueAuthSession is FALSE then destroy session |

Table 13:d - Example OSAP Session

Suppose now that the TPM user wishes to send another command using the same session to operate on the same key. For the purposes of this example, we will assume that the same ordinal is to be used (TPM_Example). To re-use the previous session, the continueAuthSession output boolean must be TRUE.

The following table shows the command execution, the parameters created and the wire formats of all of the information.

In this case, authLastNonceEven is the nonceEven value returned by the TPM with the output parameters from the first execution of TPM_Example.

| Caller | On the wire | Dir | TPM |
|---|---|---|---|
| Generate nonceOdd<br>Compute inAuth = HMAC (sharedSecret, inParamDigest, inAuthSetupParams)<br>Save nonceOdd with authHandle | | | |
| Send TPM_Example | tag<br>paramSize<br>ordinal<br>keyHandle<br>inArgOne<br>inArgTwo<br>nonceOdd<br>continueAuthSession<br>inAuth | → | Retrieve authLastNonceEven from internal session storage<br>HM = HMAC (sharedSecret, inParamDigest, inAuthSetupParams)<br>Compare HM to inAuth. If they do not compare return with TPM_AUTHFAIL<br>Execute TPM_Example and create returnCode<br>Generate nonceEven to replace authLastNonceEven in session<br>Set resAuth = HMAC(sharedSecret, outParamDigest, outAuthSetupParams) |
| Save nonceEven<br>HM = HMAC( sharedSecret, outParamDigest, outAuthSetupParams)<br>Compare HM to resAuth. This verifies returnCode and output parameters. | tag<br>paramSize<br>returnCode<br>outArgOne<br>nonceEven<br>continueAuthSession<br>resAuth | ← | Return output parameters<br>If continueAuthSession is FALSE then destroy session |

Table 13:e - Example Re-used OSAP Session

The TPM user could then use the session for further authorization sessions or terminate it in the ways that have been described above in TPM_OIAP. Note that termination of the OSAP session causes the TPM to destroy the shared secret.

**End of informative comment**

## OSAP Actions

1. The TPM MUST have been able to retrieve the shared secret (Shared, say) of the target entity when the authorization session was established with TPM_OSAP. The entity and its secret must have been previously loaded into the TPM.

2. The TPM MUST verify that the authorization handle (H, say) referenced in the command points to a valid session. If it does not, the TPM returns the error code TPM_INVALID_AUTHHANDLE.

3. The TPM MUST calculate the HMAC (HM1, say) of the command parameters according to previously specified normative regarding HMAC calculation.

4. The TPM SHALL compare HM1 to the AuthData value received in the command. If they are different, the TPM returns the error code TPM_AUTHFAIL if the authorization session is the first session of a command, or TPM_AUTH2FAIL if the authorization session is the second session of a command., the TPM executes command C1 which produces an output (O, say) that requires authentication and uses a particular return code (RC, say).

5. The TPM SHALL generate the latest version of the even nonce (nonceEven).

6. The TPM MUST calculate the HMAC (HM2) of the return parameters according to previously specified normative regarding HMAC calculation.

7. The TPM returns HM2 in the parameter list.

8. The TPM SHALL retrieve the continue flag from the received command. If the flag is FALSE, the TPM SHALL terminate the session and destroy the thread associated with handle H.

9. If the shared secret was used to provide confidentiality for data in the received command, the TPM SHALL terminate the session and destroy the thread associated with handle H.

10. Each time that access to an entity (key) is authorized using OSAP, the TPM MUST ensure that the OSAP shared secret is that derived from the entity using TPM_OSAP.

## 13.3 Authorization Session Handles

**Start of informative comment**

The TPM generates authorization handles to allow for the tracking of information regarding a specific authorization invocation.

The TPM saves information specific to the authorization, such as the nonce values, ephemeral secrets and type of authentication in use.

The TPM may create any internal representation of the handle that is appropriate for the TPM's design. The requestor always uses the handle in the authorization structure to indicate authorization structure in use.

The TPM must support a minimum of two concurrent authorization handles. The use of these handles is to allow the Owner to have an authorization active in addition to an active authorization for an entity.

To ensure garbage collection and the proper removal of security information, the requestor should terminate all handles. Termination of the handle uses the continue-use flag to indicate to the TPM that the handle should be terminated.

Termination of a handle instructs the TPM to perform garbage collection on all AuthData. Garbage collection includes the deletion of the ephemeral secret.

**End of informative comment**

1. The TPM MUST support authorization handles. See Section 23 Session pool.
2. The TPM MUST support authorization handle termination. The termination includes secure deletion of all authorization session information.

## 13.4 Authorization-Data Insertion Protocol (ADIP)

**Start of informative comment**

The ADIP allows for the creation of new entities and the secure insertion of the new entity AuthData. The transmission of the new AuthData uses encryption with the key based on the shared secret of an OSAP session.

The creation of AuthData is the responsibility of the entity owner. He or she may use whatever process he or she wishes. The transmission of the AuthData from the entity owner to the TPM requires confidentiality and integrity. Since these requirements are not always met (e.g., because the insertion of the AuthData occurs over a network) additional measures have to be taken. The ADIP protocol ensures confidentiality of the AuthData, while the OSAP session HMAC provides integrity.

When ADIP uses the parent key shared secret, care must be taken when that secret is a well known value. In that case, it may be appropriate to wrap the ADIP command in a transport session.

When the requestor is sending the AuthData to the TPM, the command requires the authorization of the entity parent. For example, to create a new TPM identity key and set its AuthData requires the AuthData of the TPM Owner. To create a new wrapped key requires the AuthData of the parent key.

The creation of a new entity requires the authorization of the entity owner. When the requestor starts the creation process, the creator must establish an OSAP session using the parent of the new entity.

For the mandatory XOR encryption algorithm, the creator builds an encryption key using a SHA-1 hash of the OSAP shared secret and a session nonce. The creator XOR encrypts the new AuthData using the encryption key as a one-time pad and sends this encrypted data along with the creation request to the TPM. The TPM decrypts the AuthData using the same OSAP shared secret and session nonce.

The XOR encryption algorithm is sufficient for almost all use models. There may be additional use models where a different encryption algorithm would be beneficial. The TPM may support AES as an additional encryption algorithm. The key and IV or counter use the OSAP shared secret and session nonces.

The creator believes that the OSAP creates a shared secret known only to the creator and the TPM. The TPM believes that the creator is the entity owner by their knowledge of the parent entity AuthData. The creator believes that the process completed correctly and that the AuthData is correct because the HMAC will only verify with the OSAP shared secret.

In the following example, we want to send the previously described command TPM_EXAMPLE to create a new entity. In the example, we assume there is a third input parameter encAuth, and that one of the input parameters is named parentHandle to reference the parent for the new entity (e.g., the SRK and its children).

| Caller | On the wire | Dir | TPM |
|---|---|---|---|
| Send TPM_OSAP | TPM_OSAP<br>parentHandle<br>nonceOddOSAP | → | Create session & authHandle<br>Generate authLastNonceEven<br>Save authLastNonceEven with authHandle<br>Save the ADIP encryption scheme with authHandle<br>Generate nonceEvenOSAP<br>Generate sharedSecret = HMAC(parent.usageAuth, nonceEvenOSAP, nonceOddOSAP)<br>Save parentHandle, sharedSecret with authHandle |
| Save authHandle, authLastNonceEven<br>Generate sharedSecret = HMAC(parent.usageAuth, nonceEvenOSAP, nonceOddOSAP)<br>Save sharedSecret | authHandle, authLastNonceEven<br>nonceEvenOSAP | ← | Returns |
| Generate nonceOdd & save with authHandle.<br>Compute input parameter newAuth = XOR( entityAuthData, SHA1(sharedSecret, authLastNonceEven))<br>Compute inAuth = HMAC (sharedSecret, inParamDigest, inAuthSetupParams) | | | |
| Send TPM_Example | tag<br>paramSize<br>ordinal<br>parentHandle<br>inArgOne<br>inArgTwo<br>encAuth<br>authHandle<br>nonceOdd<br>continueAuthSession<br>inAuth | → | Verify authHandle points to a valid session, mismatch returns TPM_AUTHFAIL<br>Retrieve authLastNonceEven from internal session storage<br>HM = HMAC (sharedSecret, inParamDigest, inAuthSetupParams)<br>Compare HM to inAuth. If they do not compare return with TPM_AUTHFAIL<br>Execute TPM_Example: decrypt encAuth to entityAuth, create entity and build returnCode. Use the ADIP encryption scheme indicated when the OSAP session was set up.<br>Generate nonceEven to replace authLastNonceEven in session<br>Set resAuth = HMAC(sharedSecret, outParamDigest, outAuthSetupParams) |
| Save nonceEven<br>HM = HMAC( sharedSecret, outParamDigest, outAuthSetupParams)<br>Compare HM to resAuth. This verifies returnCode and output parameters. | tag<br>paramSize<br>returnCode<br>outArgOne<br>nonceEven<br>continueAuthSession<br>resAuth | ← | Return output parameters<br>Terminate the authorization session associated with authHandle |

Table 13:f - Example ADIP Session

**End of informative comment**

1. The TPM MUST enable ADIP by using the OSAP or DSAP

    a. When an ordinal Action indicates that OSAP is required for the ADIP protocol (e.g., TPM_CreateWrapKey), DSAP shall satisfy that requirement.

    b. The upper byte of the entity type indicates the encryption scheme.

    c. The TPM internally stores the encryption scheme as part of the session and enforces the encryption choice on the subsequent use of the session.

    d. When TPM_ENTITY_TYPE is used for ordinals other than TPM_OSAP or TPM_DSAP (i.e., for cases where there is no ADIP encryption action), the TPM_ENTITY_TYPE upper byte MUST be 0x00.

2. The TPM MUST destroy the session whenever a new entity AuthData is created.

3. The TPM MUST encrypt the AuthData for the new entity.

    a. The TPM MUST support the XOR encryption scheme.

    b. The TPM MAY support AES symmetric key encryption schemes.

        i. If TPM_PERMANENT_FLAGS -> FIPS is TRUE

            (1) All encrypted authorizations MUST use a symmetric key encryption scheme.

    a. Encrypted AuthData values occur in the following commands

        i. TPM_CreateWrapKey

        ii. TPM_ChangeAuth

        iii. TPM_ChangeAuthOwner

        iv. TPM_Seal

        v. TPM_Sealx

        vi. TPM_MakeIdentity

        vii. TPM_CreateCounter

        viii. TPM_CMK_CreateKey

        ix. TPM_NV_DefineSpace

            (1) This ordinal contains a special case where no encryption is used.

        x. TPM_Delegate_CreateKeyDelegation

        xi. TPM_Delegate_CreateOwnerDelegation

4. If the entity type indicates XOR encryption for the AuthData secret

    a. Create X1 the SHA-1 of the concatenation of (authHandle -> sharedSecret || authLastNonceEven).

    b. Create the decrypted AuthData the XOR of X1 and the encrypted AuthData.

    c. If the command ordinal contains a second AuthData2 secret (e.g. TPM_CreateWrapKey)

        i. Create X2 the SHA-1 of the concatenation of (authHandle -> sharedSecret || nonceOdd).

        ii. Create the decrypted AuthData2 the XOR of X2 and the encrypted AuthData2.

5. If the entity type indicates symmetric key encryption
   a. The key for the encryption algorithm is the first bytes of the OSAP shared secret.
      i. E.g., For AES128, the key is the first 16 bytes of the OSAP shared secret.
      ii. There is no support for AES keys greater than 128 bits.
   b. If the entity type indicates CTR mode
      i. The initial counter value for AuthData is the first bytes of authLastNonceEven.
         (1) E.g., For AES128, the initial counter value is the first 16 bytes of authLastNonceEven.
      ii. If the command ordinal contains a second AuthData2 secret (e.g. TPM_CreateWrapKey)
         (1) The initial counter value for AuthData2 is the first bytes of nonceOdd.
      iii. Additional counter values as required are generated by incrementing the counter value as described in 31.1.3 TPM_ES_SYM_CTR.

**Start of informative comment**

The method of incrementing the counter value is different from that used by some standard crypto libraries (e.g. openSSL, Java JCE) that increment the entire counter value. TPM users should be aware of this to avoid errors when the counter wraps.

**End of informative comment**

## 13.5 AuthData Change Protocol (ADCP)

**Start of informative comment**

All entities from the Owner to the SRK to individual keys and data blobs have AuthData. This data may need to change at some point in time after the entity creation. The ADCP allows the entity owner to change the AuthData. The entity owner of a wrapped key is the owner of the parent key.

A requirement is that the owner must remember the old AuthData. The only mechanism to change the AuthData when the entity owner forgets the current value is to delete the entity and then recreate it.

To protect the data from exposure to eavesdroppers or other attackers, the AuthData uses the same encryption mechanism in use during the ADIP.

Changing AuthData requires opening two authentication handles. The first handle authenticates the entity owner (or parent) and the right to load the entity. This first handle is an OSAP and supplies the data to encrypt the new AuthData according to the ADIP protocol. The second handle can be either an OIAP or an OSAP, it authorizes access to the entity for which the AuthData is to be changed.

The AuthData in use to generate the OSAP shared secret must be the AuthData of the parent of the entity to which the change will be made.

When changing the AuthData for the SRK, the first handle OSAP must be setup using the TPM Owner AuthData. This is because the SRK does not have a parent, per se.

If the SRKAuth data is known to userA and userB, userA can snoop on userB while userB is changing the AuthData for a child of the SRK, and deduce the child's newAuth. Therefore, if SRKAuth is a well known value, TPM_ChangeAuthAsymStart and TPM_ChangeAuthAsymFinish are preferred over TPM_ChangeAuth when changing AuthData for children of the SRK.

This applies to all children of the SRK, including TPM identities.

**End of informative comment**

1. Changing AuthData for the TPM SHALL require authorization of the current TPM Owner.
2. Changing AuthData for the SRK SHALL require authorization of the TPM Owner.
3. If SRKAuth is a well known value, TPM_ChangeAuth SHOULD NOT be used to change the AuthData value of a child of the SRK, including the TPM identities.
4. All other entities SHALL require authorization of the parent entity.

## 13.6 Asymmetric Authorization Change Protocol (AACP)

**Start of informative comment**

This is now deprecated. Use the normal change session inside of a transport session with confidentiality.

This asymmetric change protocol allows the entity owner to change entity authorization, under the parent’s execution authorization, to a value of which the parent has no knowledge.

In contrast, the TPM_ChangeAuth command uses the parent entity AuthData to create the shared secret that encrypts the new AuthData for an entity. This creates a situation where the parent entity ALWAYS knows the AuthData for entities in the tree below the parent. There may be instances where this knowledge is not a good policy.

This asymmetric change process requires two commands and the use of an authorization session.

**End of informative comment**

1. Changing AuthData for the SRK SHALL involve authorization by the TPM Owner.
2. If SRKAuth is a well known value,
   a. TPM_ChangeAuthAsymStart and TPM_ChangeAuthAsymFinish SHOULD be used to change the AuthData value of a child of the SRK, including the TPM identities.
3. All other entities SHALL involve authorization of the parent entity.

# 14. FIPS 140 Physical Protection

**Start of informative comment**

The FIPS 140-2 program provides assurance that a cryptographic device performs properly. It is appropriate for TPM vendors to attempt to obtain FIPS 140-2 certification.

The TPM design should be such that the TPM vendor has the opportunity of obtaining FIPS 140-2 certification.

**End of informative comment**

## 14.1 TPM Profile for FIPS Certification

**Start of informative comment**

The FIPS mode of the TPM does require some changes over the normal TPM. These changes are listed here such that there is a central point of determining the necessary FIPS changes.

**Key creation and use**

TPM_LoadKey, TPM_CMK_CreateKey and TPM_CreateWrapKey changed to disallow the creation or loading of TPM_AUTH_NEVER, legacy and keys less than 1024 bits. TPM_MakeIdentity changed to disallow TPM_AUTH_NEVER.

**End of informative comment**

1. Each TPM Protected Capability MUST be designed such that some profile of the Capability is capable of obtaining FIPS 140-2 certification

# 15. Maintenance

**Start of informative comment**

The maintenance feature is a vendor-specific feature, and its implementation is vendor-specific. The implementation must, however, meet the minimum security requirements so that implementations of the maintenance feature do not result in security weaknesses.

There is no requirement that the maintenance feature is available, but if it is implemented, then the requirements must be met.

The maintenance feature described in the specification is an example only, and not the only mechanism that a manufacturer could implement that meets these requirements.

Maintenance is different from backup/migration, because maintenance provides for the migration of both migratory and non-migratory data. Maintenance is an optional TPM function, but if a TPM enables maintenance, the maintenance capabilities in this specification are mandatory – no other migration capabilities shall be used. Maintenance necessarily involves the manufacturer of a Subsystem.

When maintaining computer systems, it is sometimes the case that a manufacturer or its representative needs to replace a Subsystem containing a TPM. Some manufacturers consider it a requirement that there be a means of doing this replacement without the loss of the non-migrational keys held by the original TPM.

The owner and users of TCG platforms need assurance that the data within protected storage is adequately protected against interception by third parties or the manufacturer.

This process MUST only be performed between two platforms of the same manufacturer and model. If the maintenance feature is supported, this section defines the required functions defined at a high level. The final function definitions and entire maintenance process is left to the manufacturer to define within the constraints of these high level functions.

Any maintenance process must have certain properties. Specifically, any migration to a replacement Subsystem must require collaboration between the Owner of the existing Subsystem and the manufacturer of the existing Subsystem. Further, the procedure must have adequate safeguards to prevent a non-migrational key being transferred to multiple Subsystems.

The maintenance capabilities TPM_CreateMaintenanceArchive and TPM_LoadMaintenanceArchive enable the transfer of all Protected Storage data from a Subsystem containing a first TPM (TPM$_1$) to a Subsystem containing a second TPM (TPM$_2$):

A manufacturer places a public key in non-volatile storage into its TPMs at manufacture time.

The Owner of TPM$_1$ uses TPM_CreateMaintenanceArchive to create a maintenance archive that enables the migration of all data held in Protected Storage by TPM$_1$. The Owner of TPM$_1$ must provide his or her authorization to the Subsystem. The TPM then creates the TPM_MIGRATE_ASYMKEY structure and follows the process defined.

The XOR process prevents the manufacturer from ever obtaining plaintext TPM$_1$ data.

The additional random data provides a means to assure that a maintenance process cannot subvert archive data and hide such subversion.

The random mask can be generated by two methods, either using the TPM RNG or MGF1 on the TPM Owners AuthData.

The manufacturer takes the maintenance blob, decrypts it with its private key, and satisfies itself that the data bundle represents data from that Subsystem manufactured by that manufacturer. Then the manufacturer checks the endorsement certificate of TPM$_2$ and verifies that it represents a platform to which data from TPM$_1$ may be moved.

The manufacturer dispatches two messages.

The first message is made available to CAs, and is a revocation of the TPM$_1$ endorsement certificate.

The second message is sent to the Owner of TPM$_2$, which will communicate the SRK, tpmProof and the manufacturer's permission to install the maintenance blob only on TPM$_2$

The Owner uses TPM_LoadMaintenanceArchive to install the archive copy into TPM$_2$, and overwrite the existing TPM$_2$-SRK and TPM$_2$-tpmProof in TPM$_2$. TPM$_2$ overwrites TPM$_2$-SRK with TPM$_1$-SRK, and overwrites TPM$_2$-tpmProof with TPM$_1$-tpmProof.

Note that the command TPM_KillMaintenanceFeature prevents the operation of TPM_CreateMaintenanceArchive and TPM_LoadMaintenanceArchive. This enables an Owner to block maintenance (and hence the migration of non-migratory data) either to or from a TPM.

It is required that a manufacturer takes steps that prevent further access of migrated data by TPM$_1$. This may be achieved by deleting the existing Owner from TPM$_1$, for example.

For the manufacturer to validate that the maintenance blob is coming from a valid TPM, the manufacturer can require that a TPM identity sign the maintenance blob. The identity would be from a CA under the control of the manufacturer and hence the manufacturer would be satisfied that the blob is from a valid TPM.

**End of informative comment**

1. The maintenance feature MUST ensure that the information can be on only one TPM at a time. Maintenance MUST ensure that at no time the process will expose a shielded location. Maintenance MUST require the active participation of the Owner.
2. Any migration of non-migratory data protected by a Subsystem SHALL require the cooperation of both the Owner of that non-migratory data and the manufacturer of that Subsystem. That manufacturer SHALL NOT cooperate in a maintenance process unless the manufacturer is satisfied that non-migratory data will exist in exactly one Subsystem. A TPM SHALL NOT provide capabilities that support migration of non-migratory data unless those capabilities are described in the TCG specification.
3. The maintenance feature MUST move the following
4. TPM_KEY for SRK. The maintenance process will reset the SRK AuthData to match the TPM Owners AuthData
5. TPM_PERMANENT_DATA -> tpmProof
6. TPM Owner's authorization

## 15.1 Field Upgrade

**Start of informative comment**

A TPM, once in the field, may need to update the protected capabilities. This command, which is optional, provides the mechanism to perform the update.

The goal is that field upgrade should only affect protected capabilities and not shielded location, so that a patch can be applied without loss of user data.  It is understood that this goal may not be achievable in all cases.

**End of informative comment**

The TPM SHOULD have provisions for upgrading the subsystem after shipment from the manufacturer. If provided the mechanism MUST implement the following guidelines:

1. The upgrade mechanisms in the TPM MUST not require the TPM to hold a global secret. The definition of global secret is a secret value shared by more than one TPM.

2. The TPM is not allowed to pre-store or use unique identifiers in the TPM for the purpose of field upgrade. The TPM MUST NOT use the endorsement key for identification or encryption in the upgrade process. The upgrade process MAY use a TPM Identity (AIK) to deliver upgrade information to specific TPM devices.

3. The upgrade process SHOULD only change protected capabilities.  The upgrade process SHOULD NOT change shielded locations.

4. The upgrade process SHOULD only access data in shielded locations where this data is necessary to validate the TPM Owner, validate the TPME and manipulate the blob

5. The TPM MUST conform to the TCG specification, protection profiles and security targets after the upgrade. The upgrade MAY NOT decrease the security values from the original security target.

6. The security target used to evaluate this TPM MUST include this command in the TOE.

# 16. Proof of Locality

**Start of informative comment**

When a platform is designed with a trusted process, the trusted process may wish to communicate with the TPM and indicate that the command is coming from the trusted process. The definition of a trusted process is a platform specific issue.

The commands that the trusted process sends to the TPM are the normal TPM commands with a modifier that indicates that the trusted process initiated the command. The TPM accepts the command as coming from the trusted process merely because the modifier is set. The TPM itself is not responsible for how the signal is asserted; only that it honors the assertions. The TPM cannot verify the validity of the modifier.

The definition of the modifier is a platform specific issue. Depending on the platform, the modifier could be a special bus cycle or additional input pins on the TPM. The assumption is that spoofing the modifier to the TPM requires more than just a simple hardware attack, but would require expertise and possibly special hardware. One example would be special cycles on the LPC bus that inform the TPM it is under the control of a process on the PC platform.

To allow for multiple mechanisms and for finer grained reporting, the TPM will include 4 locality modifiers. These four modifiers allow the platform specific specification to properly indicate exactly what is occurring and for TPM's to properly respond to locality.

**End of informative comment**

1. The TPM modifies the receipt of a command and indicates that the trusted process sent the command when the TPM determines that the modifier is on. The modifier MUST only affect the individual command just received and MUST NOT affect any other commands. However, TPM_ExecuteTransport MUST propagate the modifier to the wrapped command.
2. A TPM platform specific specification MAY indicate the presence of a maximum of 4 local modifiers. The modifier indication uses the TPM_MODIFIER_INDICATOR data type.
3. The received modifier MUST indicate a single level.
4. The definition of the trusted source is in the platform specific specification.
5. For ease in reading this specification the indication that the TPM has received any modifier will be LOCAL_MOD = TRUE.

# 17. Monotonic Counter

**Start of informative comment**

The monotonic counter provides an ever-increasing incremental value. The TPM must support at least 4 concurrent counters. Implementations inside the TPM may create 4 unique counters or there may be one counter with pointers to keep track of the pointers current value. A naming convention to allow for unambiguous reference to the various components the following terms are in use:

Internal Base – This is the main counter. It is in use internally by the TPM and is not directly accessible by any outside process.

External Counter – A counter in use by external processes. This could be related to the main counter via pointers and difference values or it could be a totally unique value. The value of an external counter is not affected by any use, increment or deletion of any other external counter.

Max Value – The max count value of all counters (internal and external). So if there were 3 external counters having values of 10, 15 and 201 and the internal base having a value of 201 then Max Value is 201. In the same example if the internal base was 502 then Max Value would be 502.

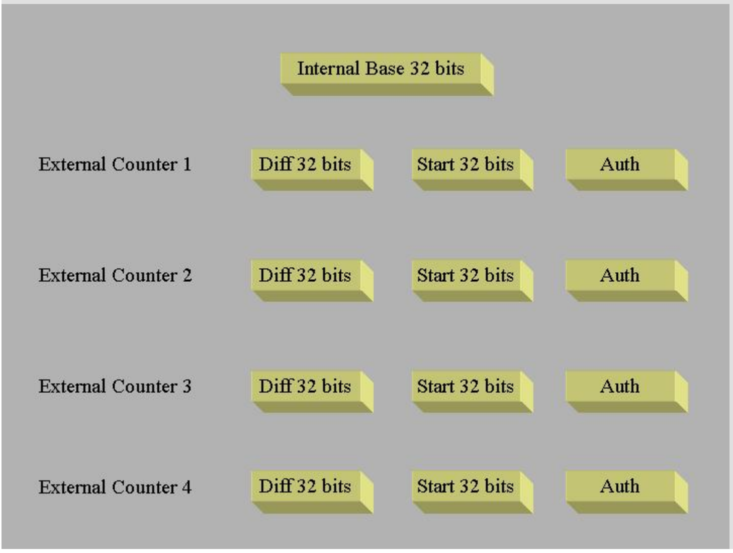
The external counter must allow for 7 years of increments every 5 seconds without causing a hardware failure. The TPM may create a throttling mechanism that limits the ability to increment an external counter within a certain time range. The output of the counter is a 32-bit value.

To create an external counter requires TPM Owner authorization. To increment an external counter the command must pass authorization to use the counter.

External counters can be tagged with a short text string to facilitate counter administration.

Manufacturers are free to implement the monotonic counter using any mechanism.

To illustrate the counters and base the following example is in use. This mechanism uses two saving values (diff and start), however this is only an example and not meant to indicate any specific implementation.



The internal base (IB) always moves forward and can never be reset. IB drives all external counters on the machine.

The purpose of the following example is to show the two external counters always moving forward independent of the other and how the IB moves forward also.

Starting condition is that IB is at 22 and no other external counters are active.

```
Start external counter A
    Increment IB (set new Max Value) IB = 23
    Assign start value of A to 23 (or Max Value)
    Assign difference of A to 23 (we always start at current value of IB)
    Assign a handle for A
Increment A 5 times
    IB is now 28
Request current A value
    Return 28 = 28 (IB) + 23 (difference) – 23 (start value)
    Counter A has gone from the start of 23 to 28 incremented 5 times.
TPM_Startup(ST_CLEAR)
Start Counter B
    Save A difference 28 = 23 (old difference) + 28 (IB) – 23 (start value)
    Increment IB (set new Max Value) IB = 29
    Set start value of B to 29 (or Max Value)
    Assign difference of B to 29
    Assign handle for B
Increment B 8 times
    IB is now 37
Request B value
    Return 37 = 37 (IB) + 29 (difference) – 29 (start value)
TPM_Startup(ST_CLEAR)
Increment A
    Store B difference (37)
    Load A start value of 37
    Increment IB to 38
Return A value
    Return 29 = 38 (IB) + 28 (difference) – 37 (start value)
```

Notice that A has gone from 28 to 29 which is correct, while B is at 37. Depending on the order of increments A may pass B or it may always be less than B.

**End of informative comment**

1. The counter MUST be designed to not wear out in the first 7 years of operation. The counter MUST be able to increment at least once every 5 seconds. The TPM, in response to operations that would violate these counter requirements, MAY throttle the counter usage (cause a delay in the use of the counter) or return an error.
2. The TPM MUST support at least 4 concurrent counters.
3. The establishment of a new counter MUST prevent the reuse of any previous counter value. I.E. if the TPM has 3 counters and the max value of a current counter is at 36 then the establishment of a new counter would start at 37.
4. After a successful TPM_Startup(ST_CLEAR) the first successful TPM_IncrementCounter sets the counter handle. Any attempt to issue TPM_IncrementCounter with a different handle MUST fail.
5. TPM_CreateCounter does NOT set the counter handle.

# 18. Transport Protection

**Start of informative comment**

The creation of sessions allows for the grouping of a set of commands into a session. The session provides a log of all commands and can provide confidentiality of the commands using the session.

Session establishment creates a shared secret and then uses the shared secret to authorize and protect commands sent to the TPM using the session. The shared secret is passed to the TPM using an asymmetric encryption key. For best security, the caller should certify that the key is never available outside the TPM.

After establishing the session, the caller uses the session to wrap a command to execute. The user of the transport session can wrap any command except for commands that would create nested transport sessions.

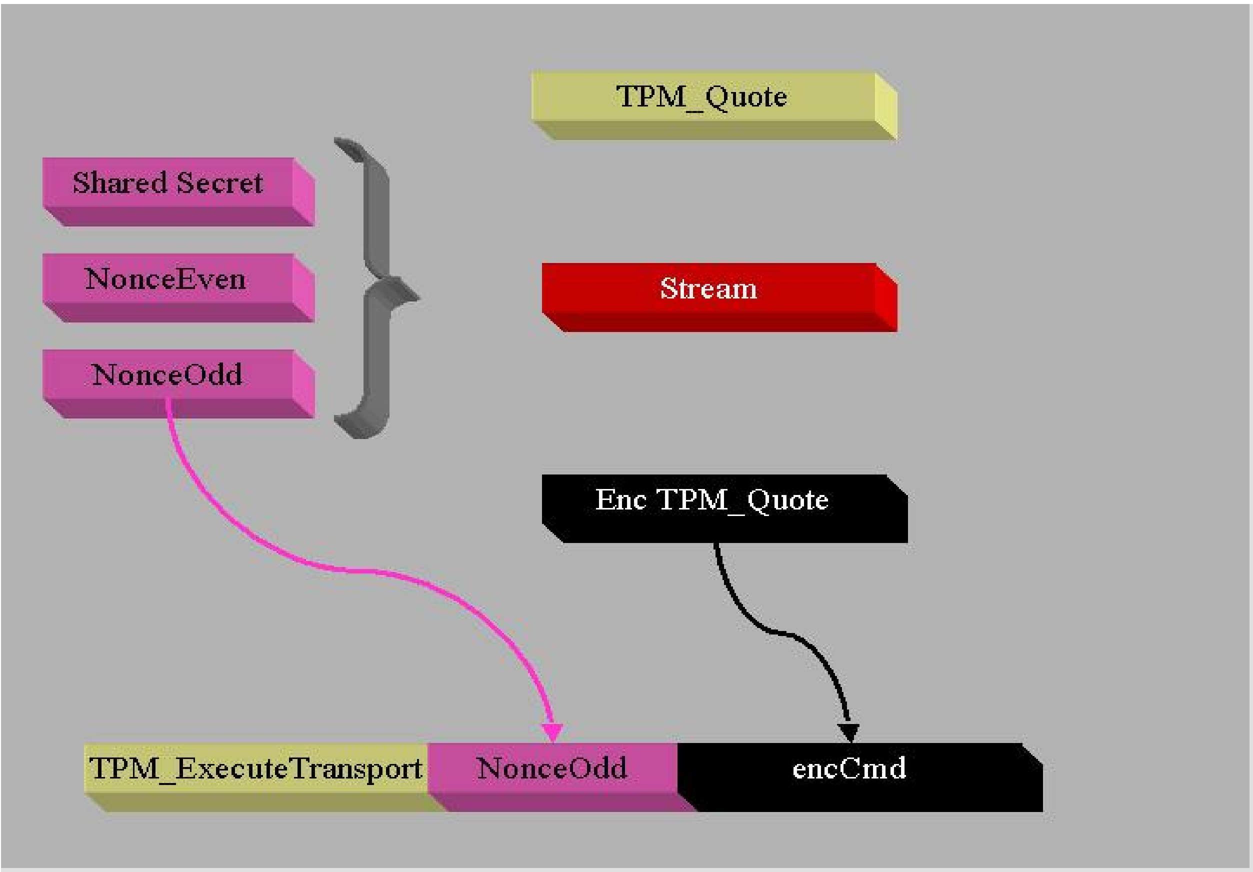
The log of executed commands uses a structure that includes the parameters and current tick count. The session log provides a record of each command using the session.

The transport session uses the same rolling nonce protocol that authorization sessions use. This protocol defines two nonces for each command sent to the TPM; nonceOdd provided by the caller and nonceEven generated by the TPM.

For confidentiality, the caller can use the MGF1 function to create an XOR string the same size as the command to execute. The inputs to the MGF1 function are the shared secret, nonceOdd and nonceEven. A symmetric key encryption algorithm can also be specified.

There is no explicit close session as the caller can use the continueSession flag set to false to end a session. The caller can also call the sign session log, which also ends the session. If the caller losses track of which sessions are active the caller should use the flush commands to regain control of the TPM resources.

For an attacker to successfully break the encryption the attacker must be able to determine from a few bits what an entire SHA-1 output was. This is equivalent to breaking SHA-1. The reason that the attacker will know some bits is that the commands are in a known format. This then allows the attacker to determine what the XOR bits were. Knowledge of 159 bits of the XOR stream does not provide any greater that 50% probability of knowing the 160th bit.



This picture shows the protection of a TPM_Quote command. Previously executed was session establishment. The nonces in use for the TPM_Quote have no relationship with the nonces that are in use for the TPM_ExecuteTransport command.

**End of informative comment**

1. The TPM MUST support a minimum of one transport session.
2. The TPM MUST NOT support the nesting of transport sessions. The definition of nesting is attempting to execute a wrapped command that is a transport session command. So for example when executing TPM_ExecuteTransport the wrapped command MUST not be TPM_ExecuteTransport.
3. The TPM MUST ensure that if transport logging is active that the inclusion of the tick count in the session log does not provide information that would make a timing attack on the operations using the session more successful.
4. The transport session MAY be exclusive. Any command executed outside of the exclusive transport session MUST cause the invalidation of the exclusive transport session.
   a. The TPM_ExecuteTransport command specifying the exclusive transport session is the only command that does not terminate the exclusive session.
5. It MAY be ineffective to wrap TPM_SaveState in a transport session. Since the TPM MAY include transport sessions in the saved state, the saved state MAY be invalidated by the wrapping TPM_ExecuteTransport.

## 18.1 Transport encryption and authorization

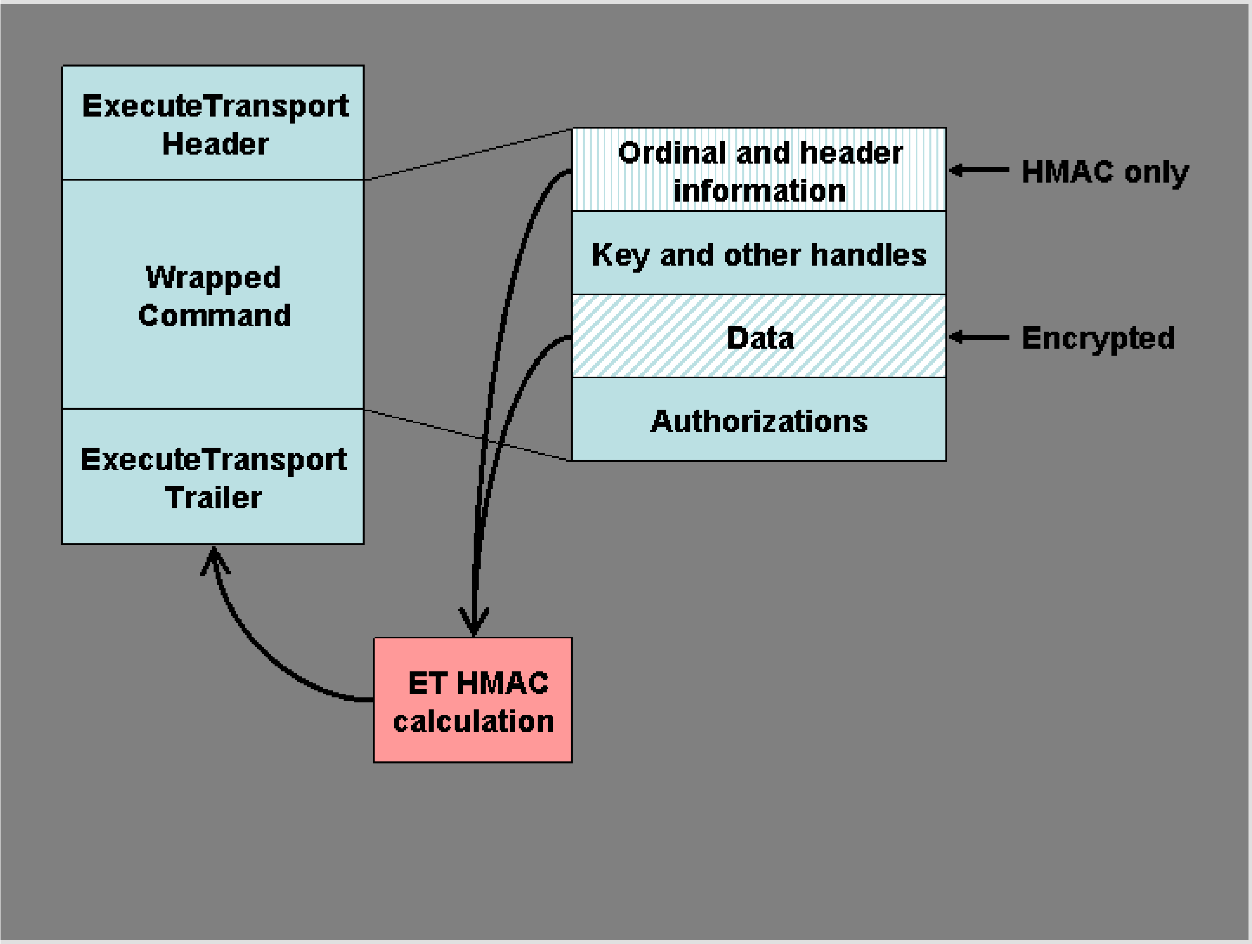
**Start of informative comment**

The confidentially of the transport protection is provided by a encrypting the wrapped command. Encryption of various items in the wrapped command makes resource management of a TPM impossible. For this reason, encryption of the entire command is not possible. In addition to the encryption issue, there are difficulties with creating the HMAC for the TPM_ExecuteTransport authorization.

The solution to these problems is to provide limited encryption and HMAC information.

The HMAC will only include two areas from the wrapped command, the command header information up to the handles, and the data after the handles. The format of all TPM commands is such that all handles are in the data stream prior to the payload or data. After the data comes the authorization information. To enable resource management, the HMAC for TPM_ExecuteTransport only includes the ordinal, header information and the data. The HMAC does not include handles and the authorization handles and nonces.

The exception is TPM_OwnerReadInternalPub, which uses fixed value key handles that are included in the encryption and HMAC calculation.



A more exact representation of the execute transport command would be the following

```
*****************************************************************
* TAGet | LENet | ORDet | wrappedCmdSize | wrappedCmd | AUTHet *
*****************************************************************
```

wrappedCmd looks like

```
***********************************************************************
* TAGw | LENw | ORDw | HANDLESw(o) | DATAw | AUTH1w (o) | AUTH2w (o) *
***********************************************************************
```

A more exact representation of the execute transport response would be the following

```
********************************************************************
* TAGet | LENet | RCet | … | wrappedRspSize | wrappedRsp | AUTHet *
********************************************************************
```

wrappedRsp looks like

```
**********************************************************************
* TAGw | LENw | RCw | HANDLESw(o) | DATAw | AUTH1w (o) | AUTH2w (o) *
**********************************************************************
```

The calculation for AUTHet takes as the data component of the HMAC calculation the concatenation of ORDw and DATAw. A normal HMAC calculation would have taken the entire wrappedCmd value but for the executeTransport calculation only the above two values are active. This does require the executeTransport command to parse the wrappedCmd to find the appropriate values.

The data for the command HMAC calculation is the following:

H1 = SHA-1 (ORDw || DATAw)

inParamDigest = SHA-1 (ORDet || wrappedCmdSize || H1)

AUTHet = HMAC (inParamDigest || lastNonceEven(et) || nonceOdd(et) || continue(et))

The data for the response HMAC calculation is the following:

H2 = SHA-1 (RCw || ORDw || DATAw)

outParamDigest = SHA-1 (RCet || ORDet || currentTicks || locality || wrappedRspSize || H1)

AUTHet = HMAC (outParamDigest || nonceEven(et) || nonceOdd(et) || continue(et))

DATAw is the unencrypted data. wrappedCmdSize and wrappedRspSize ares the actual size of the DATAw area and not the size of H1 or H2.

**End of informative comment**

The TPM MUST release a transport session and all information related to the session when:

1. TPM_ReleaseTransportSigned is executed
2. TPM_ExecuteTransport is executed with continueTransSession set to FALSE
3. Any failure of the integrity check during execution of TPM_ExecuteTransport
4. If the session has TPM_TRANSPORT_LOG set and the TPM tick session is interrupted for any reason. This is due to the return of tick values without the nonces associated with the session.

5. The TPM executes some command that deactivates the TPM or removes the TPM Owner or EK.

### 18.1.1 MGF1 parameters

**Start of informative comment**

MGF1 provides the confidentiality for the transport session. MGF1 is a function from PKCS 1 version 2.0. This function provides a mechanism to distribute entropy over a large sequence. The sequence provides a value to XOR over the message. This in effect creates a stream cipher but not one that is available for bulk encryption.

Transport confidentiality uses MGF1 as a stream cipher and obtains the entropy for each message from the following three parameters; nonceOdd, nonceEven and session authData.

It is imperative that the stream cipher not use the same XOR sequence at any time. The following illustrates how the sequence changes for each message (both input and output).

M1Input – N2, N1, sessionSecret)

M1Output – N4, N1, sessionSecret)

M2Input – N4, N3, sessionSecret)

M2Output – N6, N3, sessionSecret)

There is an issue with this sequence. If the caller does not change N1 to N3 between M1Output and M2Input then the same sequence will be generated. The TPM does not enforce the requirement to change this value so it is possible to leak information.

The fix for this is to add one more parameter, the direction. So the sequence is now this:

M1Input – N2, N1, “in”, sessionSecret)

M1Output – N4, N1, “out”, sessionSecret)

M2Input – N4, N3, “in”, sessionSecret)

M2Output – N6, N3, “out”, sessionSecret)

Where “in” indicates the in direction and “out” indicates the out direction.

Notice the calculation for M1Output uses “out” and M2Input uses “in”, so if the caller makes a mistake and does not change nonceOdd, the sequence will still be different.

nonceEven is under control of the TPM and is always changing, so there is no need to worry about nonceEven not changing.

**End of informative comment**

### 18.1.2 HMAC calculation

**Start of informative comment**

The HMAC calculation for transports presents some issues with what should and should not be in the calculation. The idea is to create a calculation for the wrapped command and add that to the wrapper.

So the data area for a wrapped command is not entirely HMAC’d like a normal command would be.

The process is to calculate the inParamDigest of the unencrypted wrapped command according to the normal rules of command HMAC calculations. Then use that value as the 3S parameter in the calculation. 2S is the actual wrapped command size, and not the size of inParamDigest.

Example using a wrapped TPM_LoadKey command

Calculate the SHA-1 value for the TPM_LoadKey command (ordinal and data) as per the normal HMAC rules. Take the digest and use that value as 3S for the TPM_ExecuteTransport HMAC calculation.

**End of informative comment**

### 18.1.3 Transport log creation

**Start of informative comment**

The log of information that a transport session creates needs a mechanism to tie any keys in use during the session to the session. As the HMAC and encryption for the command specifically exclude handles, there is no direct way to create the binding.

When creating the transport input log, if the handle(s) points to a key or keys, the public keys are digested into the log. The session owner knows the value of any keys in use and hence can still create a log that shows the values used by the log and can validate the session.

**End of informative comment**

### 18.1.4 Additional Encryption Mechanisms

**Start of informative comment**

The TPM can optionally implement alternate algorithms for the encryption of commands sent to the TPM_ExecuteTransport command. The designation of the algorithm uses the TPM_ALGORITHM_ID and TPM_ENC_SCHEME elements of the TPM_TRANSPORT_PUBLIC parameter of the TPM_EstablishTransport command.

The anticipation is that AES will be supported by various TPM’s. Symmetric algorithms have options available to them like key size, block size and operating mode. When using an algorithm other than MGF1 the algorithm and scheme must specify these options.

**End of informative comment**

1. The TPM MAY support other symmetric algorithms for the confidentiality requirement in TPM_EstablishTransport

## 18.2 Transport Error Handling

**Start of informative comment**

With the transport hiding the actual execution of commands and the transport capable of generating errors, rules must be established to allow for the errors and the results of commands to be properly passed to TPM callers.

**End of informative comment**

1. There are 3 error cases:

2. C1 is the case where an error occurs during the processing of the transport package at the TPM. In this case, the wrapped command has not been sent to the command decoder. Errors occurring during C1 are sent back to the caller as a response to the TPM_ExecuteTransport command. The error response does not have confidentiality.

3. C2 is the case where an error occurs during the processing of the wrapped command. This results in an error response from the command. The session returns the error response according to the attributes of the session.

4. C3 is the case where an error occurs after the wrapped command has completed processing and the TPM is preparing the response to the TPM_ExecuteTransport command. In this case, where the TPM does have an internal error, the TPM has no choice but to return the error as in C1. This however hides the results of the wrapped command. If the wrapped command completed successfully then there are session nonces that are being returned to the caller that are lost. The loss of these nonces causes the caller to be unsure of the state of the TPM and requires the reestablishment of sessions and keys.

## 18.3 Exclusive Transport Sessions

**Start of informative comment**

The caller may establish an exclusive session with the TPM. When an exclusive session is running, execution of any command other then TPM_ExecuteTransport or TPM_ReleaseTransportSigned targeting the exclusive session causes the abnormal invalidation of the exclusive transport session. Invalidation means that the handle is no longer valid and all subsequent attempts to use the handle return an error.

The design for the exclusive session provides an assurance that no other command executed on the TPM. It is not a lock to prevent other operations from occurring. Therefore, the caller is responsible for ensuring no interruption of the sequence of commands using the TPM.

**One exclusive session**

The TPM only supports one exclusive session at a time. There is no nesting or other commands possible. The TPM maintains an internal flag that indicates the existence of an exclusive session.

**TSS responsibilities**

It is the responsibility of the TSS (or other controlling software) to ensure that only commands using the session reach the TPM. As the purpose of the session is to show that nothing else occurred on the TPM during the session, the TSS should control access to the TPM and prevent any other uses of the TPM. The TSS design must take into account the possibility of exclusive session handle invalidation.

**Sleep states**

Exclusive sessions as defined here do not work across TPM_SaveState and TPM_Startup(ST_STATE) invocations. To have this sequence work properly there would need to be exceptions to allowing only TPM_ExecuteTranport and TPM_ReleaseTransportSigned in an exclusive session. The requirement for these exceptions would come from the attempt of the TSS to understand the current state of the TPM. Commands like TPM_GetCapability and others would have to execute to inform the TSS as to the internal state of the TPM. For this reason, there are no exceptions to the rule and the exclusive session does not remain active across a TPM_SaveState command.

**End of informative comment**

1. The TPM MUST support only one exclusive transport session
2. The TPM MUST invalidate the exclusive transport session upon the receipt of any command other than TPM_ExecuteTransport or TPM_ReleaseTransportSigned targeting the exclusive session.
    a. Invalidation includes the release of any resources assigned to the session

## 18.4 Transport Audit Handling

**Start of informative comment**

Auditing of TPM_ExecuteTransport occurs as any other command that may require auditing. There are two entries in the log, one for input one for output. The execution of the wrapped command can create an anomaly in the log.

Assume that both TPM_ExecuteTransport and the wrapped commands require auditing, the audit flow would look like the following:

TPM_ExecuteTransport input parameters

wrapped command input parameters

wrapped command output parameters

TPM_ExecuteTransport output parameters

**End of informative comment**

1. Audit failures are reported using the AUTHFAIL error commands and reflect the success or failure of the wrapped command.

### 18.4.1 Auditing of wrapped commands

**Start of informative comment**

Auditing provides information to allow an auditor to recreate the operations performed. Confidentiality on the transport channel is to hide what operations occur. These two features are in conflict. According to the TPM design philosophy, the TPM Owner takes precedence.

For a command sent on a transport session, with the session using confidentiality and the command requiring auditing, the TPM will execute the command however the input and output parameters for the command are ignored.

**End of informative comment**

1. When the wrapped command requires auditing and the transport session specifies encryption, the TPM MUST perform the audit. However, when computing the audit digest:
    a. For input, only the ordinal is audited.
    b. For output, only the ordinal and return code are audited.

# 19. Audit Commands

**Start of informative comment**

To allow the TPM Owner the ability to determine that certain operations on the TPM have been executed, auditing of commands is possible. The audit value is a digest held internally to the TPM and externally as a log of all audited commands. With the log held externally to the TPM, the internal digest must allow the log auditor to determine the presence of attacks against the log. The evidence of tampering may not provide evidence of the type of attack mounted against the log.

The TPM cannot enforce any protections on the external log. It is the responsibility of the external log owner to properly maintain and protect the log.

The TPM provides mechanisms for the external log maintainer to resynchronize the internal digest and external logs.

The Owner has the ability to set which functions generate an audit event and to change which functions generate the event at any time.

The status of the audit generation is not sensitive information and so the command to determine the status of the audit generation is not an owner authorized command.

It is important to note the difference between auditing and the logging of transport sessions. The audit log provides information on the execution of specific commands. There will be a very limited number of audited commands, most likely those commands that provide identities and control of the TPM. Commands such as TPM_Unseal would not be audited. They would use the logging functions of a transport session.

The auditing of an ordinal happens in a two-step process. The first step involves auditing the receipt of the command and the input parameters; the second step involves auditing the response to the command and the output parameters.

There is a requirement to enable verification of the external audit log both during a power session and across power sessions and to enable detection of partial or inconsistent audit logs throughout the lifetime of a TPM.

A TPM will hold an internal record consisting of a non-volatile counter (that increments once per session, when the first audit event of that session occurs) and a digest (that holds the digest of the current session). Most probably, the audit digest will be volatile. Note, however, that nothing in this specification prevents the use of a non-volatile audit digest. This arrangement of counter and digest is advantageous because it is easier to build a high endurance non-volatile counter than a high endurance non-volatile digest. This arrangement is insufficient, however, because the truncation of an audit log of any session is possible without trace. It is therefore necessary to perform an explicit close on the audit session. If there is no record of a close-audit event in an audit session, anything could have happened after the last audit event in the audit log. The essence of a typical TPM audit recording mechanism is therefore:

The TPM contains a volatile digest used like a PCR, where the “integrity metrics” are digests of command parameters in the current audit session.

An audit session opens when the volatile “PCR” digest is “extended” from its NULL state. This occurs whenever an audited command is executed AND no audit session currently exists, and in no other circumstances. When an audit session opens, a non-volatile counter is automatically incremented.

An audit session closes when a TPM receives TPM_GetAuditDigestSigned with a closeAudit parameter asserted. An audit session must be considered closed if the value in the volatile digest is invalid (for whatever reason).

TPM_GetCapability should report the effect of TPM_Startup on the volatile digest. (TPMs may initialize the volatile digest on the first audit command after TPM_Startup(ST_CLEAR), or on the first audit command after any version of TPM_Startup, or may be independent of TPM_Startup.)

When the TPM signs its audit digest, it signs the concatenation of the non-volatile counter and the volatile digest, and exports the value of the non-volatile counter, plus the value of the volatile digest, plus the value of the signature.

If the audit digest is initialized by TPM_Startup(ST_STATE), then it may be useless to audit the TPM_SaveState ordinal. Any command after TPM_SaveState MAY invalidate the saved state. If authorization sessions are part of the saved state, TPM_GetAuditDigestSigned will most likely invalidate the state as it changes the preserved authorization session nonce. It may therefore be impossible to get the audit results.

The system designer needs to ensure that the selected TPM can handle the specific environment and avoid burnout of the audit monotonic counter.

**End of informative comment**

1. Audit functionality is optional
    a. If the platform specific specification requires auditing, the specification SHALL indicate how the TPM implements audit
2. The TPM MUST maintain an audit monotonic count that is only available for audit purposes.
    a. The increment of this audit counter is under the sole control of the TPM and is not usable for other count purposes.
    b. This monotonic count MUST BE incremented by one whenever the audit digest is “extended” from a NULL state.
3. The TPM MUST maintain an audit digest.
    a. This digest MUST be set to all zeros upon the execution of TPM_GetAuditDigestSigned with a TRUE value of closeAudit provided that the signing key is an identity key.
    b. This digest MAY be set to all zeros on TPM_Startup[ST_CLEAR] or TPM_Startup[ST_STATE].
    c. When an audited command is executed, this register MUST be extended with the digest of that command.
4. Each command ordinal has an indicator in non-volatile TPM memory that indicates if execution of the command will generate an audit event. The setting of the ordinal indicator MUST be under control of the TPM Owner.

## 19.1 Audit Monotonic Counter

**Start of informative comment**

The audit monotonic counter (AMC) performs the task of sequencing audit logs across audit sessions. The AMC must have no other uses other than the audit log.

The TPM and platform should be matched such that the expected AMC endurance matches the expected platform audit sessions and sleep cycles.

Given the size of the AMC it is not anticipated that the AMC would roll over. If the AMC were to roll over, and the storage of the AMC still allowed updates, the AMC could cycle and start at 0 again.

**End of informative comment**

1. The AMC is a TPM_COUNTER_VALUE.
2. The AMC MUST last for 7 years or at least 1,000,000 audit sessions, whichever occurs first. After this amount of usage, there is no guarantee that the TPM will continue to properly increment the monotonic counter.

# 20. Design Section on Time Stamping

**Start of informative comment**

The TPM provides a service to apply a time stamp to various blobs. The time stamp provided by the TPM is not an actual universal time clock (UTC) value but is the number of timer ticks the TPM has counted. It is the responsibility of the caller to associate the ticks to an actual UTC time.

The TPM counts ticks from the start of a timing session. Timing sessions are platform dependent events that may or may not coincide with TPM_Init and TPM_Startup sessions. The reason for this difference is the availability of power to the TPM. In a PC desktop, for instance power could be continually available to the TPM by using power from the wall socket. For a PC mobile platform, power may not be available when only using the internal battery. It is a platform designer's decision as to when and how they supply power to the TPM to maintain the timing ticks.

The TPM can provide a time stamping service. The TPM does not maintain an internal secure source of time rather the TPM maintains a count of the number of ticks that have occurred since the start of a timing session.

On a PC, the TPM may use the timing source of the LPC bus or it may have a separate clock circuit. The anticipation is that availability of the TPM timing ticks and the tick resolution is an area of differentiation available to TPM manufactures and platform providers.

**End of informative comment**

1. This specification makes no requirement on the mechanism required to implement the tick counter in the TPM.
2. This specification makes no requirement on the ability for the TPM to maintain the ability to increment the tick counter across power cycles or in different power modes on a platform.

## 20.1 Tick Components

**Start of informative comment**

The TPM maintains for each tick session the following values:

Tick Count Value (TCV) – The count of ticks for the session.

Tick Increment Rate (TIR) – The rate at which the TCV is incremented. There is a set relationship between TIR and seconds, the relationship is set during manufacturing of the TPM and platform. This is the TPM_CURRENT_TICKS -> tickRate parameter.

Tick Session Nonce (TSN) – The session nonce is set at the start of each tick session.

**End of informative comment**

1. The TCV MUST be set to 0 at the start of each tick session. The TPM MUST start a new tick session if the TPM loses the ability to increment the TCV according to the TIR.
2. The TSN MUST be set to the next value from the TPM RNG at the start of each new tick session. When the TPM loses the ability to increment the TCV according to the TIR the TSN MUST be set to all zeros.

3. If the TPM discovers tampering with the tick count (through timing changes etc) the TPM MUST treat this as an attack and shut down further TPM processing as if a self-test had failed.

## 20.2 Basic Tick Stamp

**Start of informative comment**

The TPM does not provide a secure time source, nor does it provide a signature over some time value. The TPM does provide a signature over some current tick counter. The signature covers a hash of the blob to stamp, the current counter value, the tick session nonce and some fixed text.

The Tick Stamp Result (TSR) is the result of the tick stamp operation that associates the TCV, TSN and the blob. There is no association with the TCV or TSR with any UTC value at this point.

**End of informative comment**

## 20.3 Associating a TCV with UTC

**Start of informative comment**

An outside observer would like to associate a TCV with a relevant time value. The following shows how to accomplish this task. This protocol is not required but shows how to accomplish the job.

EntityA wants to have BlobA time stamped. EntityA performs TPM_TickStamp on BlobA. This creates TSRB (TickStampResult for Blob). TSRB records TSRBTCV, the current value of the TCV, and associates TSRBTCV with the TSN.

Now EntityA needs to associate a TCV with a real time value. EntityA creates blob TS which contains some known text like “Tick Stamp”. EntityA performs TPM_TickStamp on blob TS creating TSR1. This records TSR1TCV, the current value of the TCV, and associates TSR1TCV with the TSN.

EntityA sends TSR1 to a Time Authority (TA). TA creates TA1 which associates TSR1 with UTC1.

EntityA now performs TPM_TickStamp on TA1. This creates TSR2. TSR2 records TSR2TCV, the current values of the TCV, and associates TSR2TCV with the TSN.

**Analyzing the associations**

EntityA has three TSR’s; TSRB the TSR of the blob that we wanted to time stamp, TSR1 the TSR associated with the TS blob and TSR2 the TSR associated with the information from the TA. EntityA wants to show an association between the various TSR such that there is a connection between the UTC and BlobA.

From TSR1 EntityA knows that TSR1TCV is less than the UTC. This is true since the TA is signing TSR1 and the creation of TSR1 has to occur before the signature of TSR1. Stated mathematically:

$$TSR1TCV < UTC1$$

From TSR2 EntityA knows that TSR2TCV is greater than the UTC. This is true since the TPM is signing TA1 which must be created before it was signed. Stated mathematically:

$$TSR2TCV > UTC1$$

EntityA now knows TSR1TCV and TSR2TCV bound UTC1. Stated mathematically:

$$TSR1TCV < UTC1 < TSR2TCV$$

This association holds true if the TSN for TSR1 matches the TSN for TSR2. If some event occurs that causes the TPM to create a new TSN and restart the TCV then EntityA must start the process all over again.

EntityA does not know when UTC1 occurred in the interval between TSR1TCV and TSR2TCV. In fact, the value TSR2TCV minus TSR1TCV (TSRDELTA) is the amount of uncertainty to which a TCV value should be associated with UTC1. Stated mathematically:

$$TSRDELTA = TSR2TCV – TSR1TCV \text{ iff } TSR1TSN = TSR2TSN$$

EntityA can obtains k1 the relationship between ticks and seconds using the TPM_GetCapability command. EntityA also obtains k2 the possible errors per tick. EntityA now calculate DeltaTime which is the conversion of ticks to seconds and the TSRDELTA. State mathematically:

$$DeltaTime = (k1 * TSRDELTA) + (k2 * TSRDELTA)$$

To make the association between DeltaTime, UTC and TSRB note the following:

$$DeltaTime = (k1*TSRDelta) + Drift = TimeChange + Drift$$

$$\text{Where } ABSOLUTEVALUE(Drift)<k2*TSRDelta$$

(1) TSR1TCV < UTC1 < TSR2TCV

True since you cannot sign something before it exists

(2) TSR1TCV < UTC1 < TSR1TCV + TSR2TCV-TSR1TCV <= TSR1TCV + DeltaTime (= TSR1TCV +TimeChange +Drift)

True because TSR1 and TSR2 are in the same tick session proved by the same TSN. (Note TimeChange is positive!)

(3) 0 < UTC1-TSR1TCV < DeltaTime

(Subtract TSR1TCV from all sides)

(4) 0 > TSR1TCV - UTC1 > -DeltaTime = -TimeChange - Drift

(Multiply through by -1)

(5) TimeChange/2 > [ TSR1TCV - (UTC1-TimeChange/2)] > -TimeChange/2 - Drift

(add TimeChange/2 to all sides)

(6) TimeChange/2 + ABSOLUTEVALUE(Drift) > [ TSR1TCV - (UTC1-TimeChange/2)]
> -TimeChange/2 - ABSOLUTEVALUE(Drift)

Making the large side of an equality bigger, and potentially making the small side smaller.

(7) ABSOLUTEVALUE[ TSR1TCV - (UTC1-TimeChange/2)] < TimeChange/2 +
ABSOLUTEVALUE(Drift)

(Definition of Absolute Value, and TimeChange is positive)

From which we see that TSR1TCV is approximately UTC1-TimeChange/2 with a symmetric possible error of TimeChange/2 + AbsoluteValue(Drift)

We can calculate this error as being less than k1*TSRDelta/2 + k2*TSRDelta.

EntityA now has the ability to associate UTC1 with TSBTSV and by allow others to know that BlobA was signed at a certain time. First TSBTSN must equal TSR1TSN. This relationship allows EntityA to assert that TSRB occurs during the same session as TSR1 and TSR2.

EntityA calculates HashTimeDelta which is the difference between TSR1TCV and TSRBTCV and the conversion of ticks to seconds. HashTimeDelta includes the same k1 and k2 as calculated above. Stated mathematically:

$$E = k2(\mathrm{TSR1TCV} - \mathrm{TSRBTCV})$$

$$\mathrm{HashTimeDelta} = k1(\mathrm{TSR1TCV} - \mathrm{TSRBTCV}) + E$$

Now the following relationships hold:

(1) $\mathrm{UTC1} - \mathrm{DeltaTime} < \mathrm{TSRBTCV} - (\mathrm{TSRBTCV} - \mathrm{TSR1TCV}) < \mathrm{UTC1}$

(2) $\mathrm{UTC1} - \mathrm{DeltaTime} < \mathrm{TSRBTCV} + \mathrm{HashTimeDelta} + E < \mathrm{UTC1}$

(3) $\mathrm{UTC1} - \mathrm{HashTimeDelta} - \mathrm{DeltaTime} - E < \mathrm{TSRBTCV} < \mathrm{UTC1} - \mathrm{HashTimeDelta} + E$

(4) $\mathrm{TSRBTCV} = (\mathrm{UTC1} - \mathrm{HashTimeDelta} - \mathrm{DeltaTime}/2) + (E + \mathrm{DeltaTime}/2)$

This has the correct properties

As DeltaTime grows so does the error bar (or the uncertainty of the time association)

As the difference between the time of the measurement and the time of the time stamp grows, so does the E as a function of E is HashTimeDelta

**End of informative comment**

## 20.4 Additional Comments and Questions

**Start of informative comment**

**Time Difference**

If two things are time stamped, say at TCVs and TCVe (for TCV at start, TCV at end) then any entity can calculate the time difference between the two events and will get:

$$\mathrm{TimeDiff} = k1*|\mathrm{TCVe} - \mathrm{TCVs}| + k2*|\mathrm{TCVe} - \mathrm{TCVs}|$$

This TimeDiff does not indicate what time the two events occurred at it merely gives the time between the events. This time difference doesn’t require a Time Authority.

**Why is TSN (tick session nonce) required?**

Without it, there is no way to associate a Time Authority stamp with any TSV, as the TSV resets at the start of every tick session. The TSN proves that the concatenation of TSV and TSN is unique.

**How does the protocol prevent replay attacks?**

The TPM signs the TSR sent to the TA. This TSR contains the unique combination of TSV and TSN. Since the TSN is unique to a tick session and the TSV continues to increment any attempt to recreate the same TSR will fail. If the TPM is reset such that the TSV is at the same value, the TSN will be a new value. If the TPM is not reset then the TSV continues to increment and will not repeat.

**How does EntityA know that the TSR1 that the TA signs is recent?**

It doesn't. EntityA checks however to ensure that the TSN is the same in all TSR. This ensures that the values are all related. If TSR1 is an old value then the HashTimeDelta will be a large value and the uncertainty of the relation of the signing to the UTC will be large.

**Why does associating a UTC time with a TSV take two steps?**

This is because it takes some time between when a request goes to a time authority and when the response comes. The protocol measures this time and uses it to create the time deltas. The relationship of TSV to UTC is somewhere between the request and response.

**Affect of power on the tick counter**

As the TPM is not required to maintain an internal clock and battery, how the platform provides power to the TPM affects the ability to maintain the tick counter. The original mechanism had the TPM maintaining an indication of how the platform provided the power. Previous performance does not predict what might occur in the future, as the platform may be unable to continue to provide the power (dead battery, pulled plug from wall etc). With the knowledge that the TPM cannot accurately report the future, the specification deleted tick type from the TPM.

The information relative to what the platform is doing to provide power to the TPM is now a responsibility of the TSS. The TSS should first determine how the platform was built, using the platform credential. The TSS should also attempt to determine the actual performance of the TPM in regards to maintaining the tick count. The TSS can help in this determination by keeping track of the tick nonce. The tick nonce changes each time the tick count is lost. By comparing the tick nonce across system events the TSS can obtain a heuristic that represents how the platform provides power to the TPM.

The TSS must define a standard set of values as to when the tick nonce continues to increment across system events.

The following are some PC implementations that give the flavor of what is possible regarding the clock on a specific platform.

TICK_INC - No TPM power battery. Clock comes from PCI clock, may stop from time to time due to clock stopping protocols such as CLKRUN.

TICK_POWER - No TPM power battery. Clock source comes from PCI clock, always runs except in S3+.

TICK_STSTATE - External power (might be battery) consumed by TPM during S3 only. Clock source comes either from a system clock that runs during S3 or from crystal/internal TPM source.

TICK_STCLEAR - Standby power used to drive counter. In desktop, may be related to when system is plugged into wall. Clock source comes either from a system clock that runs when standby power is available or from crystal/internal TPM source.

TICK_ALWAYS - TPM power battery. Clock source comes either from a battery powered system clock that crystal/internal TPM source.

**End of informative comment**

# 21. Context Management

**Start of informative comment**

The TPM is a device that contains limited resources. Caching of the resources may occur without knowledge or assistance from the application that loaded the resource. In version 1.1 there were two types of resources that had need of this support keys and authorization sessions. Each type had a separate load and restore operation. In version 1.2 there is the addition of transport sessions. To handle these situations generically 1.2 is defining a single context manager that all types of resources may use.

The concept is simple, a resource manager requests that wrapping of a resource in a manner that securely protects the resource and only allows the restoring of the resource on the same TPM and during the same operational cycle.

Consider a key successfully loaded on the TPM. The parent keys that loaded the key may have required a different set of PCR registers than are currently set on the TPM. For example, the end result is to have key5 loaded. Key3 is protected by key2, which is protected by key1, which is protected by the SRK. Key1 requires PCR1 to be in a certain state, key2 requires PCR2 to load and key3 requires PCR3. Now at some point in time after key1 loaded key2, PCR1 was extended with additional information. If key3 is evicted then there is no way to reload key3 until the platform is rebooted. To avoid this type of problem the TPM can execute context management routines. The context management routines save key3 in its current state and allow the TPM to restore the state without having to use the parent keys (key1 and key2).

There are numerous issues with performing context management on sessions. These issues revolve around the use of the nonces in the session. If an attacker can successfully store, attack, fail and then reload the session the attacker can repeat the attack many times.

The key that the TPM uses to encrypt blobs may be a volatile or non-volatile key. One mechanism would be for the TPM to generate a new key on each TPM_Startup command. Another would be for the TPM to generate the key and store it persistently in the TPM_PERMANENT_DATA area.

The symmetric key should be relatively the same strength as a 2048-bit RSA key. 128-bit AES would be appropriate.

**End of informative comment**

1. Context management is a required function.
2. Execution of the context commands MUST NOT cause the exposure of any TPM shielded location.
3. The TPM MUST NOT allow the context saving of the EK or the SRK.
4. The TPM MAY use either symmetric or asymmetric encryption. For asymmetric encryption the TPM MUST use a 2048 RSA key.
5. A wrapped session blob MUST only be loadable once. A wrapped key blob MAY be reloadable.
6. The TPM MUST support a minimum of 16 concurrent saved contexts other than keys. There is no minimum or maximum number of concurrent saved key contexts.

7. All external session blobs (of type TPM_RT_TRANS or TPM_RT_AUTH) can be invalidated upon specific request (via TPM_FlushXXX using TPM_RT_CONTEXT as resource type). This does not include saved context blobs of type TPM_RT_KEY.

8. External session blobs are invalidated on TPM_Startup(ST_CLEAR) or on TPM_Startup(any) based on the startup effects settings

    a. Saved context blobs of type TPM_RT_KEY with the attributes of parentPCRStatus = FALSE and isVolatile = FALSE SHOULD not invalidated on TPM_Startup(any)

9. All external session invalidate automatically upon installation of a new owner due to the setting of a new tpmProof.

10. If the TPM enters failure mode ALL session blobs (including keys) MUST be invalidated

    a. Invalidation includes ensuring that contextNonceKey and contextNonceSession will change when the TPM recovers from the failure.

11. Attempts to restore a wrapped blob after the successful completion of TPM_Startup(ST_CLEAR) MUST fail. The exception is a wrapped key blob which may be long-term and which MAY restore after a TPM_Startup(ST_CLEAR).

12. The save and load context commands are the generic equivalent to the context commands in 1.1. Version 1.2 deprecates the following commands:

    a. TPM_AuthSaveContext

    b. TPM_AuthLoadContext

    c. TPM_KeySaveContext

    d. TPM_KeyLoadContext

# 22. Eviction

**Start of informative comment**

The TPM has numerous resources held inside of the TPM that may need eviction. The need for eviction occurs when the number or resources in use by the TPM exceed the available space. For resources that are hard to reload (i.e. keys tied to PCR values) the outside entity should first perform a context save before evicting items.

In version 1.1 there were separate commands to evict separate resource types. This new command set uses the resource types defined for context saving and creates a generic command that will evict all resource types.

**End of informative comment**

1. The TPM MUST NOT flush the EK or SRK using this command.
2. Version 1.2 deprecates the following commands:
   a. TPM_Terminate_Handle
   b. TPM_EvictKey
   c. TPM_Reset

# 23. Session pool

**Start of informative comment**

The TPM supports two types of sessions that use the rolling nonce protocol, authorization and transport. These sessions require much of the same handling and internal storage by the TPM. To allow more flexibility the internal storage for these sessions will be defined as coming from the same pool (or area).

The pool requires that three (3) sessions be available. The entities using the TPM can determine the usage models of what sessions are active. This allows a TPM to have 3 authorization sessions or 3 transport sessions at one time.

Using all available pool resources for transport sessions is not a very usable model. If all resources are in use by transport, there are no resources available for authorization sessions and hence no ability to execute any commands requiring authorization. A more realistic model would be to have two transport sessions and one authorization session. While this is an unrealistic model for actual execution there will be no requirement that the TPM prevent this from happening. A model of how it could occur would be when there are two applications running, both using 2 transport sessions and one authorization session. When switching between the applications, if the requirement was that only 2 transport sessions could be active the TSS that would provide the context switch would have to ensure that the transport sessions were context saved first.

Sessions can be virtualized, so while the TPM may only have 3 loaded sessions, there may be an unlimited number of context saved sessions stored outside the TPM.

**End of informative comment**

1. The TPM MUST support a minimum of three (3) concurrent sessions. The sessions MAY be any mix of authentication and transport sessions.

# 24. Initialization Operations

**Start of informative comment**

Initialization is the process where the TPM establishes an operating environment from a no power state. Initialization occurs in many different flavors with PCR, keys, handles, sessions and context blobs all initialized, reloaded or unloaded according to the rules and platform environment.

Initialization does not affect the operational characteristics of the TPM (like TPM Ownership).

Clear is the process of returning the TPM to factory defaults. The clear commands need protection from unauthorized use and must allow for the possibility of changing Owners. The clear process requires authorization to execute and locks to prevent unauthorized operation.

The clear functionality performs the following tasks:

Invalidate SRK. Invalidating the SRK invalidates all protected storage areas below the SRK in the hierarchy. The areas below are not destroyed they just have no mechanism to be loaded anymore.

All TPM volatile and non-volatile data is set to default value except the endorsement key pair. The clear includes the Owner-AuthData, so after performing the clear, the TPM has no Owner. The PCR values are undefined after a clear operation.

The TPM shall return TPM_NOSRK until an Owner is set. After the execution of the clear command, the TPM must go through a power cycle to properly set the PCR values.

The Owner has ultimate control of when a clear occurs.

The Owner can perform the TPM_OwnerClear command using the TPM Owner authorization. If the Owner wishes to disable this clear command and require physical access to perform the clear, the Owner can issue the TPM_DisableOwnerClear command.

During the TPM startup processing anyone with physical access to the machine can issue the TPM_ForceClear command. This command performs the clear. The TPM_DisableForceClear disables the TPM_ForceClear command for the duration of the power cycle. TSS startup code that does not issue the TPM_DisableForceClear leaves the TPM vulnerable to a denial of service attack. The assumption is that the TSS startup code will issue the TPM_DisableForceClear on each power cycle after the TSS determines that it will not be necessary to issue the TPM_ForceClear command. The purpose of the TPM_ForceClear command is to recover from the state where the Owner has lost or forgotten the TPM Ownership token.

The TPM_ForceClear must only be possible when the issuer has physical access to the platform. The manufacturer of a platform determines the exact definition of physical access.

**End of informative comment**

1. The TPM MUST support proper initialization. Initialization MUST properly configure the TPM to execute in the platform environment.
2. Initialization MUST ensure that handles, keys, sessions, context blobs and PCR are properly initialized, reloaded or invalidated according to the platform environment.

3. The description of the platform environment arrives at the TPM in a combination of TPM_Init and TPM_Startup.

# 25. HMAC digest rules

**Start of informative comment**

The order of calculation of the HMAC is critical to being able to validate the authorization and parameters of a command. All commands use the same order and format for the calculation.

A more exact representation of a command would be the following

```
*****************************************************************
* TAG  | LEN  | ORD  | HANDLES | DATA  | AUTH1 (o) | AUTH2  (o) *
*****************************************************************
```

The text area for the HMAC calculation would be the concatenation of the following:

ORD || DATA

**End of informative comment**

The HMAC digest of parameters uses the following order

1. Skip tag and length
2. Include ordinal. This is the 1S parameter in the HMAC column for each command
3. Skip handle(s). This includes key and other session handles
4. Include data and other parameters for the command. This starts with the 2S parameter in the HMAC column for each command.
5. Skip all AuthData values.

# 26. Generic authorization session termination rules

**Start of informative comment**

These rules are the generic rules that govern all authorization sessions, a specific session type may have additional rules or modifications of the generic rules

**End of informative comment**

1. A TPM SHALL unilaterally perform the actions of TPM_FlushSpecific for a session upon any of the following events
   a. "continueUse" flag in the authorization session is FALSE
   b. Shared secret of the session in use to create the exclusive-or for confidentiality of data. Example is TPM_ChangeAuth terminates the authorization session. TPM_ExecuteTransport does not terminate the session due to protections inherent in transport sessions.
   c. When the associated entity is invalidated
   d. When the command returns a fatal error. This is due to error returns not setting a nonceEven. Without a new nonceEven the rolling nonces sequence is broken hence the TPM MUST terminate the session.
   e. Failure of an authorization check at the start of the command
   f. Execution of TPM_Startup(ST_CLEAR)
2. The TPM MAY perform the actions of TPM_FlushSpecific for a session upon the following events
   a. Execution of TPM_Startup(ST_STATE)

# 27. PCR Grand Unification Theory

**Start of informative comment**

This section discusses the unification of PCR definition and use with locality.

The PCR allow the definition of a platform configuration. With the addition of locality, the meaning of a configuration is somewhat larger. This section defines how the two combine to provide the TPM user information relative to the platform configuration.

These are the issues regarding PCR and locality at this time

**Definition of configuration**

A configuration is the combination of PCR, PCR attributes and the locality.

**Passing the creators configuration to the user of data**

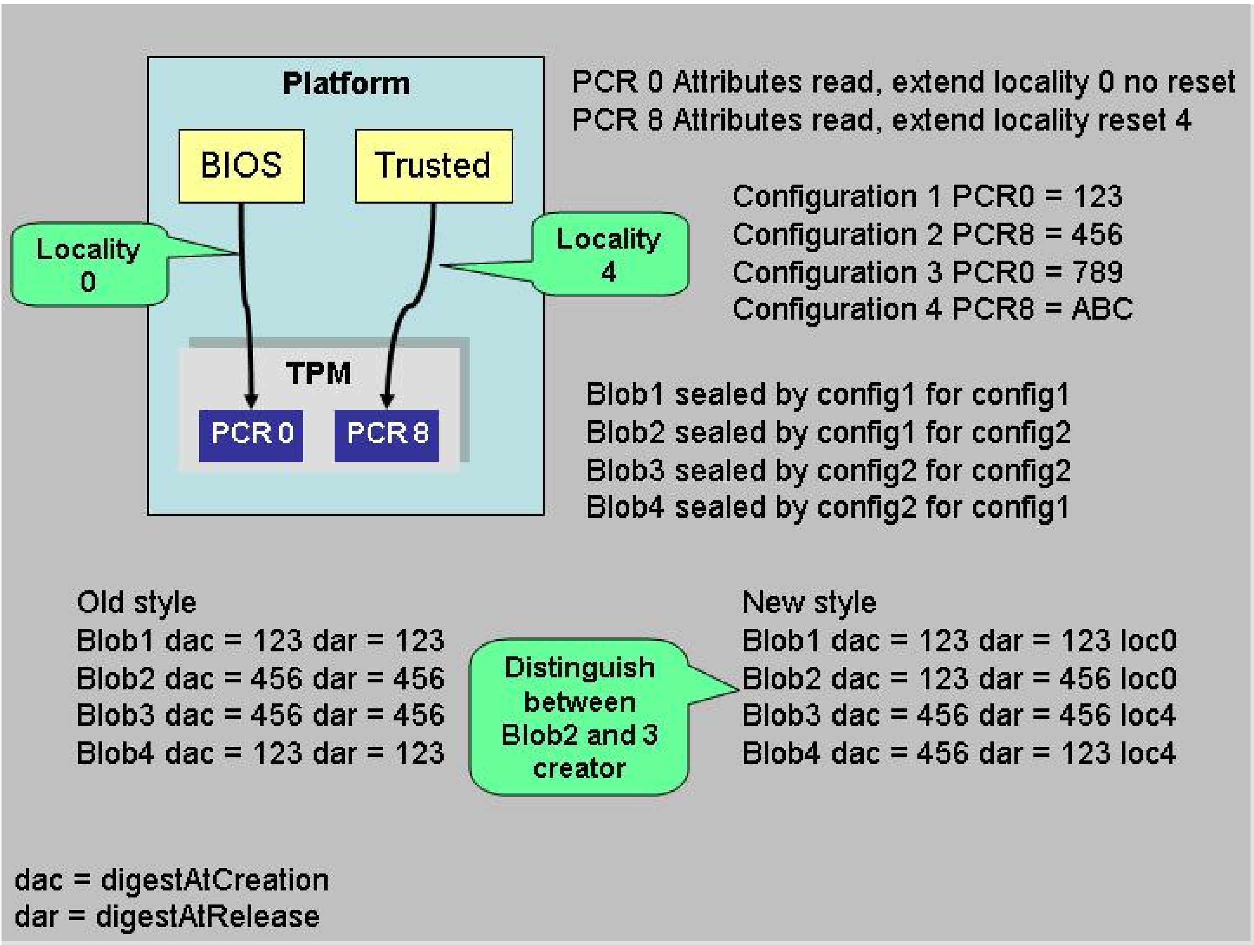
For many reasons, from the creator's viewpoint and the user's viewpoint, the configuration in use by the creator is important information. This information needs transmitting to the user with the data and with integrity.

The configuration must include the locality and may not be the same configuration that will use the data. This allows one configuration to seal a value for future use and the end user to know the genealogy of where the data comes from.

**Definition of "Use"**

See the definition of TPM_PCR_ATTRIBUTES for the attributes and the normative statements regarding the use of the attributes. The use of a configuration is when the TPM needs to ensure that the proper platform configuration is present. The first example is for Unseal, the TPM must only release the information sealed if the platform configuration matches the configuration specified by the seal creator. Here the use of locality is implicit in the PCR attributes, if PCR8 requires locality 2 to be present then the seal creator ensures that locality 2 is asserted by defining a configuration that uses PCR8.

The creation of a blob that specifies a configuration for use is not a "use" itself. So the SEAL command does is not a use for specifying the use of a PCR configuration.



PCR 0 Attributes read, extend locality 0 no reset
PCR 8 Attributes read, extend locality reset 4

Configuration 1 PCR0 = 123
Configuration 2 PCR8 = 456
Configuration 3 PCR0 = 789
Configuration 4 PCR8 = ABC

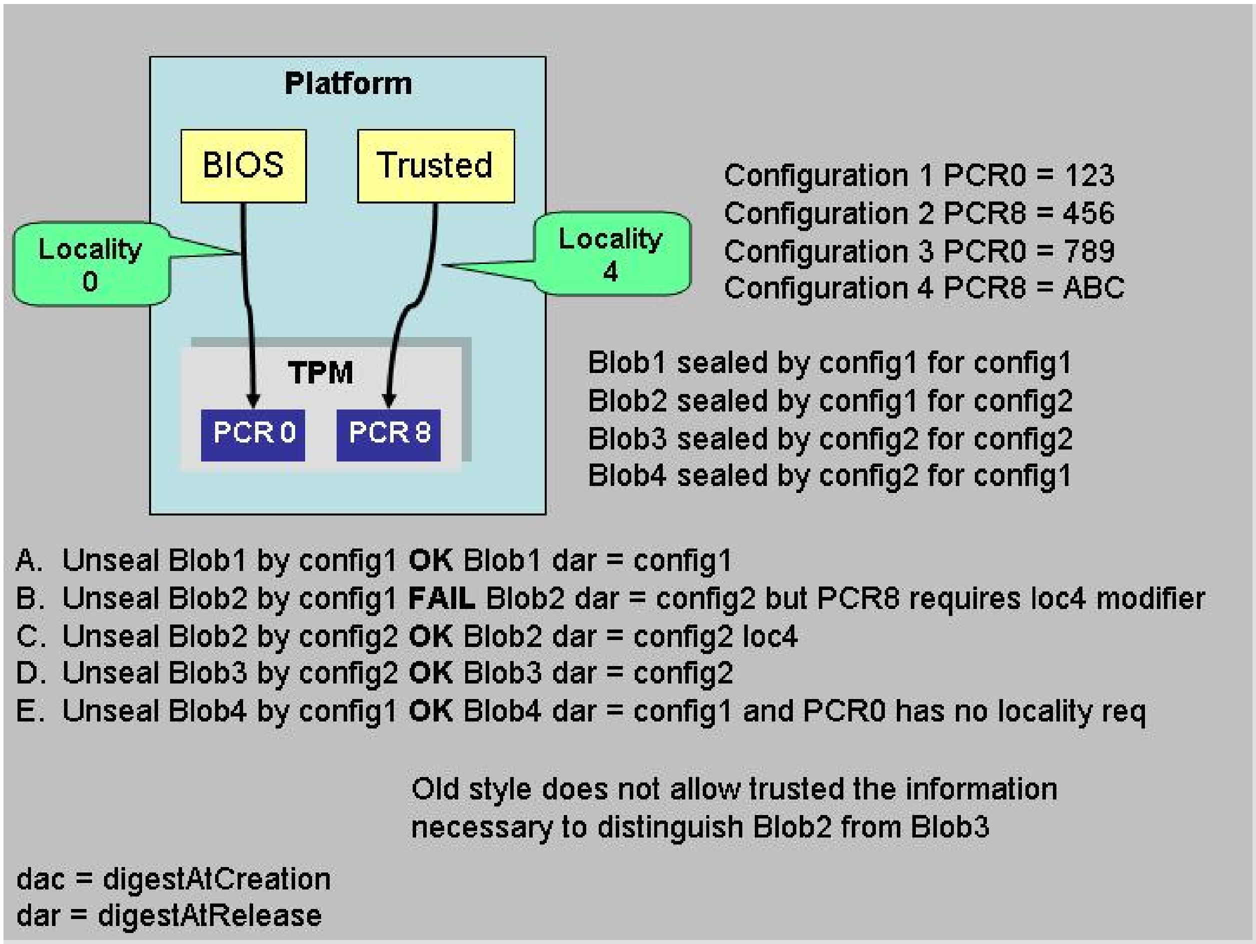
Blob1 sealed by config1 for config1
Blob2 sealed by config1 for config2
Blob3 sealed by config2 for config2
Blob4 sealed by config2 for config1

Old style
Blob1 dac = 123 dar = 123
Blob2 dac = 456 dar = 456
Blob3 dac = 456 dar = 456
Blob4 dac = 123 dar = 123

New style
Blob1 dac = 123 dar = 123 loc0
Blob2 dac = 123 dar = 456 loc0
Blob3 dac = 456 dar = 456 loc4
Blob4 dac = 456 dar = 123 loc4

dac = digestAtCreation
dar = digestAtRelease

By using the "new style" or TPM_PCR_INFO_LONG structure the user can determine that Blob2 is different that Blob3.



Configuration 1 PCR0 = 123
Configuration 2 PCR8 = 456
Configuration 3 PCR0 = 789
Configuration 4 PCR8 = ABC

Blob1 sealed by config1 for config1
Blob2 sealed by config1 for config2
Blob3 sealed by config2 for config2
Blob4 sealed by config2 for config1

A. Unseal Blob1 by config1 **OK** Blob1 dar = config1
B. Unseal Blob2 by config1 **FAIL** Blob2 dar = config2 but PCR8 requires loc4 modifier
C. Unseal Blob2 by config2 **OK** Blob2 dar = config2 loc4
D. Unseal Blob3 by config2 **OK** Blob3 dar = config2
E. Unseal Blob4 by config1 **OK** Blob4 dar = config1 and PCR0 has no locality req

Old style does not allow trusted the information necessary to distinguish Blob2 from Blob3

dac = digestAtCreation
dar = digestAtRelease

Case B is the only failure and this shows the use of the locality modifier and PCR locality attribute.

Additional attempts are obvious failures, config3 and config4 are unable to unseal any of the 4 blobs.

One example is illustrative of the problems of just specifying locality without an accompanying PCR. Assume Blob5 which specifies a dar of config1 and a locality 4 modifier. Now either config2 or config4 can unseal Blob5. In fact there is no way to restrict ANY process that gains access to locality 4 from performing the unseal. As many platforms will have no restrictions as to which process can load in locality 4 there is no additional benefit of specifying a locality modifier. If the sealer wants protections, they need to specify a PCR that requires a locality modifier.

### Defining locality modifiers dynamically

This feature would enable the platform to specify how and when a locality modifier applies to a PCR. The current definition of PCR attributes has the values set in TPM manufacturing and static for all TPM in a specific platform type (like a PC).

Defining dynamic attributes would make the use of a PCR very difficult. The sealer would have to have some way of ensuring that their wishes were enforced and challengers would have to pay close attention to the current PCR attributes. For these reasons the setting of the PCR attributes is defined as a static operation made during the platform specific specification.

**End of informative comment**

## 27.1 Validate Key for use

**Start of informative comment**

The following shows the order and checks done before the use of a key that has PCR or locality restrictions.

Note that there is no check for the PCR registers on the DSAP session. This is due to the fact that DSAP checks for the continued validity of the PCR that are attached to the DSAP and any change causes the invalidation of the DSAP session.

The checks must validate the locality of the DSAP session as the PCR registers in use could have locality restrictions.

**End of informative comment**

1. If the authorization session is DSAP
   a. If the DSAP -> localityAtRelease is not 0x1F (or in other words some localities are not allowed)
      i. Validate that TPM_STANY_FLAGS -> localityModifier is matched by DSAP -> pcrInfo -> localityAtRelease, on mismatch return TPM_BAD_LOCALITY
   b. If DSAP -> digestAtRelease is not 0
      i. Calculate the current digest and compare to digestAtRelease, return TPM_BAD_PCR on mismatch
   c. If the DSAP points to an ordinal delegation
      i. Check that the DSAP authorizes the use of the intended ordinal
   d. If the DSAP points to a key delegation
      i. Check that the DSAP authorizes the use of the key
   e. If the key delegated is a CMK key
      i. The TPM MUST check the CMK_DELEGATE restrictions
2. Set LK to the loaded key that is being used
3. If LK -> pcrInfoSize is not 0
   a. If LK -> pcrInfo -> releasePCRSelection identifies the use of one or more PCR
      i. Calculate H1 a TPM_COMPOSITE_HASH of the PCR selected by LK -> pcrInfo -> releasePCRSelection
      ii. Compare H1 to LK -> pcrInfo -> digestAtRelease on mismatch return TPM_WRONGPCRVAL
   b. If localityAtRelease is NOT 0x1F
      i. Validate that TPM_STANY_FLAGS -> localityModifier is matched by LK -> pcrInfo -> localityAtRelease on mismatch return TPM_BAD_LOCALITY
4. Allow use of the key

# 28. Non Volatile Storage

**Start of informative comment**

The TPM contains protected non-volatile storage. There are many uses of this type of area; however, a TPM needs to have a defined set of operations that touch any protected area. The idea behind these instructions is to provide an area that the manufacturers and owner can use for storing information in the TPM.

The TCG will define a limited set of information that it sees a need of storing in the TPM. The TPM and platform manufacturer may add additional areas.

The NV storage area has a limited use before it will no longer operate Hence the NV commands are under TPM Owner control.

Controls exist to allow a manufacturer to define and write NV indexes during manufacturing before an owner exists. This is strictly a manufacturing mode, as it allows a manufacturer to bypass security.

To locate if an index is available, use TPM_GetCapability to return the index and the size of the area in use by the index.

The area may not be larger than the TPM input buffer. The TPM will report the maximum size available to allocate.

The storage area is an opaque area to the TPM. The TPM, other than providing the storage, does not review the internals of the area.

To SEAL a blob, the creator of the area specifies the use of PCR registers to read the value. This is the exact property of SEAL.

To obtain a signed indication of what is in a NV store area the caller would setup a transport session with logging on and then get the signed log. The log shows the parameters so the caller can validate that the TPM holds the value.

There is an attribute, for each index, that defines the expected write scheme for the index. The TPM may handle data storage differently based on the write scheme attribute that defines the expected for the index. Whenever possible the NV memory should be allocated with the write scheme attribute set to update as one block and not as individual bytes.

The non-volatile storage described here is defined by TPM_NV_DefineSpace. Other structures that a manufacturer might decide to store in non-volatile memory (e.g., PCRs, keys, the audit digest) are logically separate and do not affect the space available for the NV indexed storage described here. An exception is a key that is moved from volatile to NV memory when set as “owner evict”. This NV memory may come from a pool shared with NV define space.

**End of informative comment**

1. The TPM MUST support the NV commands. The TPM MUST support the NV area as defined by the TPM_NV_INDEX values.

2. The TPM MAY manage the storage area using any allocation and garbage collection scheme.

3. To remove an area from the NV store the TPM owner would use the TPM_NV_DefineSpace command with a size of 0. Any authorized user can change the value written in the NV store.

4. The TPM MUST treat the NV area as a shielded location.

   a. The TPM does not provide any additional protections (like additional encryption) to the NV area.

5. If a write operation is interrupted, then the TPM makes no guarantees about the data stored at the specified index. It MAY be the previous value, MAY be the new value or MAY be undefined or unpredictable. After the interruption the TPM MAY indicate that the index contains unpredictable information.

   a. The TPM MUST ensure that in case of interruption of a write to an index that all other indexes are not affected

6. Minimum size of NV area is platform specific. The maximum area is TPM vendor specific.

7. A TPM MUST NOT use the NV area to store any data dependent on data structures defined in Part II of the TPM specifications, except for the NV Storage structures implied by required index values or reserved index values.

## 28.1 NV storage design principles

**Start of informative comment**

This section lists the design principles that motivate the NV area in the TPM. There was the realization that the current design made use of NV storage but not necessarily efficiently. The DIR, BIT and other commands placed demands on the TPM designer and required areas that while allowing for flexible use reserved space most likely never used (like DIR for locality 1).

The following are the design principles that drive the function definitions.

1. Provide efficient use of NV area on the TPM. NV storage is a very limited resource and data stored in the NV area should be as small as possible.

2. The TPM does not control, edit, validate or manipulate in any manner the information in the NV store. The TPM is merely a storage device. The TPM does enforce the access rules as set by the TPM Owner.

3. Allocation of the NV area for a specific use must be under control of the TPM Owner.

4. The TPM Owner, when defining the area to use, will set the access and use policy for the area. The TPM Owner can set AuthData values, delegations, PCR values and other controls on the access allowed to the area.

5. There must be a capability to allow TPM and platform manufacturers to use this area without a TPM Owner being present. This allows the manufacturer to place information into the TPM without an onerous manufacturing flow. Information in this category would include EK credential and platform credential.

6. The management and use of the NV area should not require a large number of ordinals.

7. The management and use of the NV area should not introduce new operating strategies into the TPM and should be easy to implement.

**End of informative comment**

### 28.1.1 NV Storage use models

**Start of informative comment**

This informative section describes some of the anticipated use models and the attributes a user of the storage area would need to set.

**Owner authorized for all access**

TPM_NV_DefineSpace: attributes = PER_OWERREAD || PER_OWNERWRITE

WriteValue(TPM Owner Auth, data)

ReadValue(TPM Owner Auth, data)

**Set AuthData value**

TPM_NV_DefineSpace: attributes = PER_AUTHREAD || PER_AUTHWRITE, auth = authValue

WriteValue( authValue, data)

ReadValue( authValue, data)

**Write once, only way to change is to delete and redefine**

TPM_NV_DefineSpace: attributes = PER_WRITEDEFINE

WriteValue( size = x, data) // successful

WriteValue(size = 0) // locks

WriteValue(size = x) // fails

…

TPM_Startup(ST_Clear) // Does not affect lock

WriteValue(size = x, data) // fails

**Write until specific index is locked, lock reset on Startup(ST_Clear)**

TPM_NV_DefineSpace: index = 3, attributes = PER_WRITE_STCLEAR

TPM_NV_DefineSpace: index = 5, attributes = PER_WRITE_STCLEAR

WriteValue( index = 3, size = x, data) // successful

WriteValue( index = 5, size = x, data) // successful

WriteValue(index = 3, size = 0) // locks

WriteValue( index = 3, size = x, data) // fails

WriteValue( index = 5, size = x, data) // successful

```
…
TPM_Startup(ST_Clear) // clears lock
WriteValue( index = 3, size = x, data) // successful
WriteValue( index = 5, size = x, data) // successful
```

**Write until index 0 is locked, lock reset by Startup(ST_Clear)**

```
TPM_NV_DefineSpace: attributes = PER_GLOBALLOCK, index = 5
TPM_NV_DefineSpace: attributes = PER_GLOBALLOCK, index = 3
WriteValue( index = 3, size = x, data) // successful
WriteValue( index = 5, size = x, data) // successful

WriteValue(index = 0) // sets SV -> bGlobalLock to TRUE
WriteValue( index = 3, size = x, data) // fails
WriteValue( index = 5, size = x, data) // fails
…
TPM_Startup(ST_Clear) // clears lock
WriteValue( index = 3, size = x, data) // successful
WriteValue( index = 5, size = x, data) // successful
```

**End of informative comment**

## 28.2 Use of NV storage during manufacturing

**Start of informative comment**

The TPM needs the ability to write values to the NV store during manufacturing. It is possible that the values written at this time would require authorization during normal TPM use. The actual enforcement of these authorizations during manufacturing would cause numerous problems for the manufacturer.

The TPM will not enforce the NV authorization restrictions until the execution of a TPM_NV_DefineSpace with the handle of TPM_NV_INDEX_LOCK.

The 'D' bit indicates an NV index defined (typically) during manufacturing and then locked. While nvLocked is FALSE, indices with the 'D' set can be defined, deleted, or redefined as desired. Once nvLocked is set TRUE, the 'D' bit indices are locked. They cannot be defined, deleted or redefined.

nvLocked has the lifetime of the endorsement key.

**End of informative comment**

1. The TPM MUST NOT enforce the NV authorizations (auth values, PCR etc.) prior to the execution of TPM_NV_DefineSpace with an index of TPM_NV_INDEX_LOCK

TCG Published

a. While the TPM is not enforcing NV authorizations, the TPM SHALL allow the use of TPM_NV_DefineSpace in any operational state (disabled, deactivated)

# 29. Delegation Model

**Start of informative comment**

The TPM Owner is an entity with a single “super user” privilege to control TPM operation. Thus if any aspect of a TPM requires management, the TPM Owner must perform that task himself or reveal his privilege information to another entity. This other entity thereby obtains the privilege to operate all TPM controls, not just those intended by the Owner. Therefore the Owner often must have greater trust in the other entity than is strictly necessary to perform an arbitrary task.

This delegation model addresses this issue by allowing delegation of individual TPM Owner privileges (the right to use individual Owner authorized TPM commands) to individual entities, which may be trusted processes.

Basic requirements:

**Consumer user does not need to enter or remember a TPM Owner password**. This is an ease of use and security issue. Not remembering the password may lead to bad security practices, increased tech support calls and lost data.

**Role based administration and separation of duty**. It should be possible to delegate just enough Owner privileges to perform some administration task or carry out some duty, without delegating all Owner privileges.

**TPM should support multiple trusted processes**. When a platform has the ability to load and execute multiple trusted processes then the TPM should be able to participate in the protection of secrets and proper management of the processes and their secrets. In fact, the TPM most likely is the root of storage for these values. The TPM should enable the proper management, protection and distribution of values held for the various trusted processes that reside on the same platform.

**Trusted processes may require restrictions.** A fundamental security tenet is the principle of least privilege, that is, to limit process functionality to only the functions necessary to accomplish the task. This delegation model provides a building block that allows a system designer to create single purpose processes and then ensure that the process only has access to the functions that it requires to complete the task.

**Maintain the current authorization structure and protocols**. There is no desire to remove the current TPM Owner and the protocols that authorize and manage the TPM Owner. The capabilities are a delegation of TPM Owner responsibilities. The delegation allows the TPM Owner to delegate some or all of the actions that a TPM Owner can perform. The TPM Owner has complete control as to when and if the capability delegation is in use.

**End of informative comment**

## 29.1 Table Requirements

**Start of informative comment**

**No ocean front property in table** – We want the table to be virtually unlimited in size. While we need some storage, we do not want to pick just one number and have that be the min and max. This drives the need for the ability to save, off the TPM, delegation elements.

**Revoking a delegation, does not affect other** delegations – The TPM Owner may, at any time, determine that a delegation is no longer appropriate. The TPM Owner needs to be able to ensure the revocation of all delegations in the same family. The TPM Owner also wants to ensure that revocation done in one family does not affect any other family of delegations.

**Table seeded by OEM** – The OEM should do the seeding of the table during manufacturing. This allows the OEM to ship the platform and make it easy for the platform owner to startup the first time. The definition of manufacturing in this context includes any time prior to or including the time the user first turns on the platform.

**Table not tied to a TPM owner** – The table is not tied to the existence of a TPM owner. This facilitates the seeding of the table by the OEM.

**External delegations need authorization and assurance of** revocation – When a delegation is held external to the TPM, the TPM must ensure authorization of the delegation when loading the delegation. Upon revocation of a family or other family changes the TPM must ensure that prior valid delegations are not successfully loaded.

**90% case, no need for external store** – The normal case should be that the platform does not need to worry about having external delegations. This drives the need for some NV storage to hold a minimum number of table rows.

**End of informative comment**

## 29.2 How this works

**Start of informative comment**

The existing TPM owner authorization model is that certain TPM commands require the authorization of the TPM Owner to operate. The authorization value is the TPM Owners token. Using the token to authorize the command is proof of TPM Ownership. There is only one token and knowledge of this token allows all operations that require proof of TPM Ownership.

This extension allows the TPM Owner to create a new AuthData value and to delegate some of the TPM Ownership rights to the new AuthData value.

The use model of the delegation is to create an authorization session (DSAP) using the delegated AuthData value instead of the TPM Owner token. This allows delegation to work without change to any current command.

The intent is to permit delegation of selected Owner privileges to selected entities, be they local or remote, separate from the current software environment or integrated into the current software environment. Thus Owner privileges may be delegated to entities on other platforms, to entities (trusted processes) that are part of the normal software environment on the Owner's platform, or to a minimalist software environment on the Owner's platform (created by booting from a CDROM, or special disk partition), for example.

Privileges may be delegated to a particular entity via definition of a particular process on the Owner's platform (by dictating PCR values), and/or by stipulating a particular AuthData value. The resultant TPM_DELEGATE_OWNER_BLOB and any AuthData value must be passed by the Owner to the chosen entity.

Delegation to an external entity (not on the Owner's platform) probably requires an AuthData value and a NULL PCR selection. (But the AuthData value might be sealed to a desired set of PCRs in that remote platform.)

Delegation to a trusted process provided by the local OS requires a PCR that indicates the trusted process. The authorization token should be a fixed value (any well known value), since the OS has no means to safely store the authorization token without sealing that token to the PCR that indicates the trusted process. It is suggested that the value 0x111…111 be used.

Delegation to a specially booted entity requires either a PCR or an authorization token, and preferably both, to recognize both the process and the fact that the Owner wishes that process to execute.

The central delegation data structure is a set of tables. These tables indicate the command ordinals delegated by the TPM Owner to a particular defined environment. The tables allow the distinction of delegations belonging to different environments.

The TPM is capable of storing internally a few table elements to enable the passing of the delegation information from an entity that has no access to memory or storage of the defined environment.

The number of delegations that the tables can hold is a dynamic number with the possibility of adding or deleting entries at any time. As the total number is dynamic, and possibly large, the TPM provides a mechanism to cache the delegations. The cache of a delegation must include integrity and confidentiality. The term for the encrypted cached entity is blob. The blob contains a counter (verificationCount) validated when the TPM loads the blob.

An Owner uses the counter mechanism to prevent the use of undesirable blobs; they increment verificationCount inside the TPM and insert the current value of verificationCount into selected table elements, including temporarily loaded blobs. (This is the reason why a TPM must still load a blob that has an incorrect verificationCount.) An Owner can verify the delegation state of his platform (immediately after updating verificationCount) by keeping copies of the elements that have just been given the current value of verificationCount, signing those copies, and sending them to a third party.

Verification probably requires interaction with a third party because acceptable table profiles will change with time and the most important reason for verification is suspicion of the state of a TOS in a platform. Such suspicion implies that the verification check must be done by a trusted security monitor (perhaps separate trusted software on another platform or separate trusted software on CDROM, for example). The signature sent to the third party must include a freshness value, to prevent replay attacks, and the security monitor must verify that a response from the third party includes that freshness value. In situations where the highest confidence is required, the third party could provide the response by an out-of-band mechanism, such as an automated telephone service with spoken confirmation of acceptability of platform state and freshness value.

A challenger can verify an entire family using a single transport session with logging, that increments the verification count, updates the verification count in selected blobs, reads the tables and obtains a single transport session signature over all of the blobs in a family.

If no Owner is installed, the delegation mechanisms are inoperative and third party verification of the tables is impossible, but tables can still be administered and corrected. (See later for more details.)

To perform an operation using the delegation the entity establishes an authorization session and uses the delegated AuthData value for all HMAC calculations. The TPM validates the

AuthData value, and in the case of defined environments checks the PCR values. If the validation is successful, the TPM then validates that the delegation allows the intended operation.

There can be at least two delegation rows stored in non-volatile storage inside a TPM, and these may be changed using Owner privilege or delegated Owner privilege. Each delegation table row is a member of a family, and there can be at least eight family rows stored in non-volatile storage inside a TPM. An entity belonging to one family can be delegated the privilege to create a new family and edit the rows in its own family, but no other family.

In addition to tying together delegations, the family concept and the family table also provides the mechanism for validation and revocation of exported delegate table rows, as well as the mechanism for the platform user to perform validation of all delegations in a family.

**End of informative comment**

## 29.3 Family Table

**Start of informative comment**

The family table has three main purposes.

1 - To provide for the grouping of rows in the TPM_DELEGATE_TABLE; entities identified in delegate table rows as belonging to the same family can edit information in the other delegate table rows with the same family ID. This allows a family to manage itself and provides an easier mechanism during upgrades.

2 - To provide the validation and revocation mechanism for exported TPM_DELEGATE_ROWS and those stored on the TPM in the delegation table

3 - To provide the ability to perform validation of all delegations in a family

The family table must have eight rows, and may have more. The maximum number of rows is TPM vendor-defined and is available using the TPM_GetCapability command.

As the family table has a limited number of rows, there is the possibility that this number could be insufficient. However, the ability to create a virtual amount of rows, like done for the TPM_DELEGATE_TABLE would create the need to have all of the validation and revocation mechanisms that the family table provides for the delegate table. This could become a recursive process, so for this version of the specification, the recursion stops at the family table.

The family table contains four pieces of information: the family ID, the family label, the family verification count, and the family flags.

The family ID is a 32-bit value that provides a sequence number of the families in use.

The family label is a one-byte field that family table manager software would use to help identify the information associated with the family. Software must be able to map the numeric value associated with each family to the ASCII-string family name displayable in the user interface.

The family verification count is a 32-bit sequence number that identifies the last outside verification and attestation of the family information.

Initialization of the family table occurs by using the TPM_Delegate_Manage command with the TPM_FAMILY_CREATE option.

The verificationCount parameter enables a TPM to check that all rows of a family in the delegate table are approved (by an external verification process), even if rows have been stored off-TPM.

The family flags allow the use and administration of the family table row, and its associated delegate table rows.

**Row contents**

Family ID – 32-bits

Row label – One byte

Family verification count – 32-bits

Family enable/disable use/admin flags – 32-bits

**End of informative comment**

## 29.4 Delegate Table

**Start of informative comment**

The delegate table has three main purposes, from the point of view of the TPM. This table holds:

The list of ordinals allowable for use by the delegate

The identity of a process that can use the ordinal list

The AuthData value to use the ordinal list

The delegate table has a minimum of two (2) rows; the maximum number of rows is TPM vendor-defined and is available using the TPM_GetCapability command. Each row represents a delegation and, optionally, an assignment of that delegation to an identified trusted process.

The non-volatile delegate rows permit an entity to pass delegation rows to a software environment without regard to shared memory between the entity and the software environment. The size of the delegate table does not restrict the number of delegations because TPM_Delegate_CreateOwnerDelegation can create blobs for use in a DSAP session, bypassing the delegate table.

The TPM Owner controls the tables that control the delegations, but (recursively) the TPM Owner can delegate the management of the tables to delegated entities. Entities belonging to a particular group (family) of delegation processes may edit delegate table entries that belong to that family.

After creation of a delegation entry there is no restriction on the use of the delegation in a properly authorized session. The TPM Owner has properly authorized the creation of the delegation so the use of the delegation occurs whenever the delegate wishes to use it.

The rows of the delegate table held in non-volatile storage are only changeable under TPM Owner authorization.

The delegate table contains six pieces of information: PCR information, the AuthData value for the delegated capabilities, the delegation label, the family ID, the verification count, and a profile of the capabilities that are delegated to the trusted process identified by the PCR information.

**Row Elements**

ASCII label – Label that provides information regarding the row. This is not a sensitive item.

Family ID – The family that the delegation belongs to; this is not a sensitive item.

Verification count – Specifies the version, or generation, of this row; version validity information is in the family table. This is not a sensitive value.

Delegated capabilities – The capabilities granted, by the TPM Owner, to the identified process. This is not a sensitive item.

**Authorization and Identity**

The creator of the delegation sets the AuthData value and the PCR selection. The creator is responsible for the protection and dissemination of the AuthData value. This is a sensitive value.

**End of informative comment**

1. The TPM_DELEGATE_TABLE MUST have at least two (2) rows; the maximum number of table rows is TPM-vendor defined and MUST be reported in response to a TPM_GetCapability command
2. The AuthData value and the PCR selection must be set by the creator of the delegation

## 29.5 Delegation Administration Control

**Start of informative comment**

The delegate tables (both family and delegation) present some control problems. The tables must be initialized by the platform OEM, administered and controlled by the TPM Owner, and reset on changes of TPM Ownership. To provide this level of control there are three phases of administration with different functions available in the phases.

The three phases of table administration are; manufacturing (P1), no-owner (P2) and owner present (P3). These three phases allow different types of administration of the delegation tables.

**Manufacturing (P1)**

A more accurate definition of this phase is open, un-initialized and un-owned. It occurs after TPM manufacturing and as a result of TPM_OwnerClear or TPM_ForceClear.

In P1 TPM_Delegate_Manage can initialize and manage non-volatile family rows in the TPM. TPM_Delegate_LoadOwnerDelegation can load non-volatile delegation rows in the TPM.

Attacks that attempt to burnout the TPM's NV storage are frustrated by the NV store's own limits on the number of writes when no Owner is installed.

**No-Owner (P2)**

This phase occurs after the platform has been properly setup. The setup can occur in the platform manufacturing flow, during the first boot of the platform or at any time when the platform owner wants to lock the table settings down. There is no TPM Owner at this time.

TPM_Delegate_Manage locks both the family and delegation rows. This lock can be opened only by the Owner (after the Owner has been installed, obviously) or by the act of removing the Owner (even if no Owner is installed). Thus locked tables can be unlocked by asserting Physical Presence and executing TPM_ForceClear, without having to install an Owner.

In P2, the relevant TPM_Delegate_xxx commands all return the error TPM_DELEGATE_LOCKED. This is not an issue as there is no TPM Owner to delegate commands, so the inability to change the tables or create delegations does not affect the use of the TPM.

**Owned (P3)**

In this phase, the TPM has a TPM Owner and the TPM Owner manages the table as the Owner sees fit. This phase continues until the removal of the TPM Owner.

Moving from P2 to P3 is automatic upon establishment of a TPM Owner. Removal of the TPM Owner automatically moves back to P1.

The TPM Owner always has the ability to administer any table. The TPM Owner may delegate the ability to manipulate a single family or all families. Such delegations are operative only if delegations are enabled.

**End of informative comment**

1. When DelegateAdminLock is TRUE the TPM MUST disallow any changes to the delegate tables
2. With a TPM Owner installed, the TPM Owner MUST authorize all delegate table changes

### 29.5.1 Control in Phase 1

**Start of informative comment**

The TPM starts life in P1. The TPM has no owner and the tables are empty. It is desirable for the OEM to initialize the tables to allow delegation to start immediately after the Owner decides to enable delegation. As the setup may require changes and validation, a simple mechanism of writing to the area once is not a valid option.

TPM_Delegate_Manage and TPM_Delegate_LoadOwnerDelegation allow the OEM to fill the table, read the public parts of the table, perform reboots, reset the table and when finally satisfied as to the state of the platform, lock the table.

Alternatively, the OEM can leave the tables NULL and turn off table administration leaving the TPM in an unloaded state waiting for the eventual TPM Owner to fill the tables, as they need.

Flow to load tables

Default values of DelegateAdminLock are set either during manufacturing or are the result of TPM_OwnerClear or TPM_ForceClear.

TPM_Delegate_Manage verifies that DelegateAdminLock is FALSE and that there is no TPM Owner. The command will therefore load or manipulate the family tables as specified in the command.

TPM_Delegate_LoadOwnerDelegation verifies that DelegateAdminLock is FALSE and no TPM owner is present. The command loads the delegate information specified in the command.

**End of informative comment**

### 29.5.2 Control in Phase 2

**Start of informative comment**

In phase 2, no changes are possible to the delegate tables. The platform owner must install a TPM Owner and then manage the tables, or use TPM_ForceClear to revert to phase 1.

**End of informative comment**

### 29.5.3 Control in Phase 3

Start of informative comment

The TPM_DELEGATE_TABLE requires commands that manage the table. These commands include filling the table, turning use of the table on or off, turning administration of the table on or off, and using the table.

The commands are:

**TPM_Delegate_Manage** – Manages the family table on a row-by-row basis: creates a new family, enables/disables use of a family table row and delegate table rows that share the same family ID, enables/disables administration of a family's rows in both the family table and the delegate table, and invalidates an existing family.

**TPM_Delegate_CreateOwnerDelegation** increments the family verification count (if desired) and delegates the Owner's privilege to use a set of command ordinals, by creating a blob. Such blobs can be used as input data for TPM_DSAP or TPM_Delegate_LoadOwnerDelegation. Incrementing the verification count and creating a delegation must be an atomic operation. Otherwise no delegations are operative after incrementing the verification count.

**TPM_Delegate_LoadOwnerDelegation** loads a delegate blob into a non-volatile delegate table row, inside the TPM.

**TPM_Delegate_ReadTable** is used to read from the TPM the public contents of the family and delegate tables that are stored on the TPM.

**TPM_Delegate_UpdateVerification** sets the verificationCount in an entity (a blob or a delegation row) to the current family value, in order that the delegations represented by that entity will continue to be accepted by the TPM.

**TPM_Delegate_VerifyDelegation** loads a delegate blob into the TPM, and returns success or failure, depending on whether the blob is currently valid.

**TPM_DSAP** – opens a deferred authorization session, using either an input blob (created by TPM_Delegate_CreateOwnerDelegation) or a cached blob (loaded by TPM_Delegate_LoadOwnerDelegation into one of the TPM's non-volatile delegation rows).

**End of informative comment**

## 29.6 Family Verification

**Start of informative comment**

The platform user may wish to have confirmation that the delegations in use provide a coherent set of delegations. This process would require some evaluation of the processes granted delegations. To assist in this confirmation the TPM provides a mechanism to group all delegations of a family into a signed blob. The signed blob allows the verification agent to look at the delegations, the processes involved and make an assessment as the validity of the delegations. The third party then sends back to the platform owner the results of the assessment.

To perform the creation of the signed blob the platform owner needs the ability to group all of the delegations of a single family into a transport session. The platform owner also wants an assurance that no management of the table is possible during the verification.

This verification does not prove to a third party that the platform owner is not cheating. There is nothing to prevent the platform owner from performing the validation and then adding an additional delegation to the family.

Here is one example protocol that retrieves the information necessary to validate the rows belonging to a particular family. Note that the local method of executing the protocol must prevent a man-in-the-middle attack using the nonce supplied by the user.

The TPM Owner can increment the family verification count or use the current family verification count. Using the current family verification count carries the risk that unexamined delegation blobs permit undesirable delegations. Using an incremented verification count eliminates that risk. The entity gathering the verification data requires Owner authorization or access to a delegation that grants access to transport session commands, plus other commands depending on whether verificationCount is to be incremented. This delegation could be a trusted process that can use the delegations because of its PCR measurements, a remote entity that can use the delegations because the Owner has sent it a TPM_DELEGATE_OWNER_BLOB and AuthData value, or the host platform booted from a CDROM that can use the delegations because of its PCR measurements, and TPM_DELEGATE_OWNER_BLOB and AuthData value submitted by the Owner, for example.

Verification using the current verificationCount

The gathering entity requires access to a delegation that grants access to at least the ordinals to perform a transport session, plus TPM_Delegate_ReadTable and TPM_Delegate_VerifyDelegation.

The TPM Owner creates a transport session with the “no other activity” attribute set. This ensures notification if other operations occur on the TPM during the validation process. (If other operations do occur, the validation processes may have been subverted.) All subsequent commands listed are performed using the transport session.

TPM_Delegate_ReadTable displays all public values (including the permissions and PCR values) in the TPM.

TPM_Delegate_VerifyDelegation loads each cached blob, with all public values (including the permissions and PCR values) in plain text.

After verifying all blobs, TPM_ReleaseTransportSigned signs the list of transactions.

The gathering entity sends the log of the transport session plus any supporting information to the validation entity, which evaluates the signed transport session log and informs the platform owner of the result of the evaluation. This could be an out-of-band process.

Verification using an incremented verificationCount

The gathering entity requires Owner authorization or access to a delegation that grants access to at least the ordinals to perform a transport session, plus TPM_Delegate_CreateOwnerDelegation, TPM_Delegate_ReadTable, and TPM_Delegate_UpdateVerification.

The TPM Owner creates a transport session with the “no other activity” attribute set.

To increment the count the TPM Owner (or a delegate) must use TPM_Delegate_CreateOwnerDelegation with increment == TRUE. That blob permits creation of new delegations or approval of existing tables and blobs. That delegation must set the PCRs of the desired (local) process and the desired AuthData value of the process. As noted previously, AuthData values should be a fixed value if the gathering entity is a trusted process that is part of the normal software environment.

If new delegations are to be created, TPM_Delegate_CreateOwnerDelegation must be used with increment == FALSE.

If existing blobs and delegation rows are to be reapproved, TPM_Delegate_UpdateVerification must be used to install the new value of verificationCount into those existing blobs and non-volatile rows. This exposes the blobs’ public information (including the permissions and PCR values) in plain text to the transport session.

TPM_Delegate_ReadTable then exposes all public values (including the permissions and PCR values) of tables to the transport session.

Again, after verifying all blobs, TPM_ReleaseTransportSigned signs the list of transactions.

**End of informative comment**

## 29.7 Use of commands for different states of TPM

**Start of informative comment**

Use the ordinal table to determine when the various commands are available for use

**End of informative comment**

## 29.8 Delegation Authorization Values

**Start of informative comment**

This section describes why, when a PCR selection is set, the AuthData value may be a fixed value, and, when the PCR selection is null, the delegation creator must select an AuthData value.

A PCR value is an indication of a particular (software) environment in the local platform. Either that PCR value indicates a trusted process or not. If the trusted process is to execute automatically, there is no point in allocating a meaningful AuthData value. (The only way the trusted process could store the AuthData value is to seal it to the process’s PCR values, but the delegation mechanism is already checking the process’s PCR values.) If execution of the trusted process is dependent upon the wishes of another entity (such as the Owner), the AuthData value should be a meaningful (private) value known only to the TPM, the Owner, and that other entity. Otherwise the AuthData value should be a fixed, well known, value.

If the delegation is to be controlled from a remote platform, these simple delegation mechanisms provide no means for the platform to verify the PCRs of that remote platform, and hence access to the delegation must be based solely upon knowledge of the AuthData value.

**End of informative comment**

### 29.8.1 Using the authorization value

**Start of informative comment**

To use a delegation the TPM will enforce any PCR selection on use. The use definition is any command that uses the delegation authorization value to take the place of the TPM Owner authorization.

**PCR Selection defined**

In this case, the delegation has a PCR selection structure defined. Each time the TPM uses the delegation authorization value instead of the TPM Owner value the TPM would validate that the current PCR settings match the settings held in the delegation structure. The PCR selection includes the definition of localities and checks of locality occur with the checking of the PCR values. The TPM enforces use of the correct authorization value, which may or may not be a meaningful (private) value.

**PCR selection NULL**

In this case, the delegation has no PCR selection structure defined. The TPM does not enforce any particular environment before using the authorization value. Mere knowledge of the value is sufficient.

**End of informative comment**

## 29.9 DSAP description

**Start of informative comment**

The DSAP opens a deferred auth session, using either a TPM_DELEGATE_BLOB as input parameter or a reference to the TPM_DELEGATE_TABLE_ROW, stored inside the TPM. The DSAP command creates an ephemeral secret to authenticate a session. The purpose of this section is to illustrate the delegation of user keys or TPM Owner authorization by creating and using a DSAP session without regard to a specific command.

A key defined for a certain usage (e.g. TPM_KEY_IDENTITY) can be applied to different functions within the use model (e.g. TPM_Quote or TPM_CertifiyKey). If an entity knows the AuthData for the key (key.usageAuth) it can perform all the functions, allowed for that use model of that particular key. This entity is also defined as delegation creation entity, since it can initiate the delegation process. Assume that a restricted usage entity should only be allowed to execute a subset or a single functions denoted as TPM_Example, within the specific use model of a key. (e.g. Allow the usage of a TPM_IDENTITY_KEY only for Certifying Keys, but no other function). This use model points to the selection of the DSAP as the authorization protocol to execute the TPM_Example command.

To perform this scenario the delegation creation entity must know the AuthData for the key (key.usageAuth). It then has to initiate the delegation by creating a TPM_DELEGATE_KEY_BLOB via the TPM_Delegate_CreateKeyDelegation command. As a next step the delegation creation entity has to pass the TPM_DELEGATE_KEY_BLOB and the delegation AuthData (TPM_DELEGATE_SENSITIVE.authValue) to the restricted usage entity. The specification offers the TPM_DelTable_ReadAuth mechanism to perform this function. Other mechanisms may be used.

The restricted usage entity can now start an TPM_DSAP session by using the TPM_DELEGATE_KEY_BLOB as input.

For the TPM_Example command, the inAuth parameter provides the authorization to execute the command. The following table shows the commands executed, the parameters created and the wire formats of all of the information.

<inParamDigest> is the result of the following calculation: SHA1(ordinal, inArgOne, inArgTwo). <outParamDigest> is the result of the following calculation: SHA1(returnCode, ordinal, outArgOne). inAuthSetupParams refers to the following parameters, in this order: authLastNonceEven, nonceOdd, continueAuthSession. OutAuthSetupParams refers to the following parameters, in this order: nonceEven, nonceOdd, continueAuthSession

In addition to the two even nonces generated by the TPM (authLastNonceEven and nonceEven) that are used for TPM_OIAP, there is a third, labeled nonceEvenOSAP that is used to generate the shared secret. For every even nonce, there is also an odd nonce generated by the system.

| Caller | On the wire | Dir | TPM |
|---|---|---|---|
| Send TPM_DSAP | TPM_DSAP<br>keyHandle<br>nonceOddOSAP<br>entityType<br>entityValue | → | Decrypt sensitiveArea of entityValue<br>If entityValue==TPM_ET_DEL_BLOB verify the integrity of the blob, and if a TPM_DELEGATE_KEY_BLOB is input verify that KeyHandle and entityValue match<br>Create session & authHandle<br>Generate authLastNonceEven<br>Save authLastNonceEven with authHandle<br>Generate nonceEvenOSAP<br>Generate sharedSecret = HMAC(sensitiveArea.authValue., nonceEvenOSAP, nonceOddOSAP)<br>Save keyHandle, sharedSecret with authHandle and permissions |
| Save authHandle, authLastNonceEven<br>Generate sharedSecret = HMAC(sensitiveArea.authValue, nonceEvenOSAP, nonceOddOSAP)<br>Save sharedSecret | authHandle, authLastNonceEven<br>nonceEvenOSAP | ← | Returns |
| Generate nonceOdd & save with authHandle.<br>Compute inAuth = HMAC (sharedSecret, inParamDigest, inAuthSetupParams) | | | |
| Send TPM_Example | tag<br>paramSize<br>ordinal<br>inArgOne<br>inArgTwo<br>authHandle<br>nonceOdd<br>continueAuthSession<br>inAuth | → | Verify authHandle points to a valid session, mismatch returns TPM_AUTHFAIL<br>Retrieve authLastNonceEven from internal session storage<br>HM = HMAC (sharedSecret, inParamDigest, inAuthSetupParams)<br>Compare HM to inAuth. If they do not compare return with TPM_AUTHFAIL<br>Check if command ordinal of TPM_Example is allowed in permissions. If not return TPM_DISABLED_CMD<br>Execute TPM_Example and create returnCode<br>Generate nonceEven to replace authLastNonceEven in session<br>Set resAuth = HMAC(sharedSecret, outParamDigest, outAuthSetupParams) |
| Save nonceEven<br>HM = HMAC( sharedSecret, outParamDigest, outAuthSetupParams)<br>Compare HM to resAuth. This verifies returnCode and output parameters. | tag<br>paramSize<br>returnCode<br>outArgOne<br>nonceEven<br>continueAuthSession<br>resAuth | ← | Return output parameters<br>If continueAuthSession is FALSE then destroy session |

Suppose now that the TPM user wishes to send another command using the same session to operate on the same key. For the purposes of this example, we will assume that the same ordinal is to be used (TPM_Example). To re-use the previous session, the continueAuthSession output boolean must be TRUE.

The following table shows the command execution, the parameters created and the wire formats of all of the information.

In this case, authLastNonceEven is the nonceEven value returned by the TPM with the output parameters from the first execution of TPM_Example.

| Caller | On the wire | Dir | TPM |
|---|---|---|---|
| Generate nonceOdd<br>Compute inAuth = HMAC (sharedSecret, inParamDigest, inAuthSetupParams)<br>Save nonceOdd with authHandle | | | |
| Send TPM_Example | tag<br>paramSize<br>ordinal<br>inArgOne<br>inArgTwo<br>nonceOdd<br>continueAuthSession<br>inAuth | → | Retrieve authLastNonceEven from internal session storage<br>HM = HMAC (sharedSecret, inParamDigest, inAuthSetupParams)<br>Compare HM to inAuth. If they do not compare return with TPM_AUTHFAIL<br>Execute TPM_Example and create returnCode<br>Generate nonceEven to replace authLastNonceEven in session<br>Set resAuth = HMAC(sharedSecret, outParamDigest, outAuthSetupParams) |
| Save nonceEven<br>HM = HMAC( sharedSecret, outParamDigest, outAuthSetupParams)<br>Compare HM to resAuth. This verifies returnCode and output parameters. | tag<br>paramSize<br>returnCode<br>outArgOne<br>nonceEven<br>continueAuthSession<br>resAuth | ← | Return output parameters<br>If continueAuthSession is FALSE then destroy session |

The TPM user could then use the session for further authorization sessions or terminate it in the ways that have been described above in TPM_OIAP. Note that termination of the DSAP session causes the TPM to destroy the shared secret.

**End of informative comment**

1. The DSAP session MUST enforce any PCR selection on use. The use definition is any command that uses the delegation authorization value to take the place of the TPM Owner authorization.

# 30. Physical Presence

**Start of informative comment**

Physical presence is a signal from the platform to the TPM that indicates the operator manipulated the hardware of the platform. Manipulation would include depressing a switch, setting a jumper, depressing a key on the keyboard or some other such action.

TCG does not specify an implementation technique. The guideline is the physical presence technique should make it difficult or impossible for rogue software to assert the physical presence signal.

A PC-specific physical presence mechanism might be an electrical connection from a switch, or a program that loads during power on self-test.

**End of informative comment**

The TPM MUST support a signal from the platform for the assertion of physical presence. A TCG platform specific specification MAY specify what mechanisms assert the physical presence signal.

The platform manufacturer MUST provide for the physical presence assertion by some physical mechanism.

## 30.1 Use of Physical Presence

**Start of informative comment**

For control purposes there are numerous commands on the TPM that require TPM Owner authorization. Included in this group of commands are those that turn the TPM on or off and those that define the operating modes of the TPM. The TPM Owner always has complete control of the TPM. What happens in two conditions: there is no TPM Owner or the TPM Owner forgets the TPM Owner AuthData value. Physical presence allows for an authorization to change the state in these two conditions.

**No TPM Owner**

This state occurs when the TPM ships from manufacturing (it can occur at other times also). There is no TPM Owner. It is imperative to protect the TPM from remote software processes that would attempt to gain control of the TPM. To indicate to the TPM that the TPM operating state can change (allow for the creation of the TPM Owner) the human asserts physical presence. The physical presence assertion than indicates to the TPM that changing the operating state of the TPM is authorized.

**Lost TPM Owner authorization**

In the case of lost, or forgotten, authorization there is a TPM Owner but no way to manage the TPM. If the TPM will only operate with the TPM Owner authorization then the TPM is no longer controllable. Here the operator of the machine asserts physical presence and removes the current TPM Owner. The assumption is that the operator will then immediately take ownership of the TPM and insert a new TPM Owner AuthData value.

**Operator disabling**

Another use of physical presence is to indicate that the operator wants to disable the use of the TPM. This allows the operator to temporarily turn off the TPM but not change the permanent operating mode of the TPM as set by the TPM Owner.

**End of informative comment**

# 31. TPM Internal Asymmetric Encryption

**Start of Informative comment**

For asymmetric encryption schemes, the TPM is not required to perform the blocking of information where that information cannot be encrypted in a single cryptographic operation. The schemes TPM_ES_RSAESOAEP_SHA1_MGF1 and TPM_ES_RSAESPKCSV15 allow only single block encryption. When using these schemes, the caller to the TPM must perform any blocking and unblocking outside the TPM. It is the responsibility of the caller to ensure that multiple blocks are properly protected using a chaining mechanism.

Note that there are inherent dangers associated with splitting information so that it can be encrypted in multiple blocks with an asymmetric key, and then chaining together these blocks together. For example, if an integrity check mechanism is not used, an attacker can encrypt his own data using the public key, and substitute this rogue block for one of the original blocks in the message, thus forcing the TPM to replace part of the message upon decryption.

There is also a more subtle attack to discover the data encrypted in low-entropy blocks. The attacker makes a guess at the plaintext data, encrypts it, and substitutes the encrypted guess for the original block. When the TPM decrypts the complete message, a successful decryption will indicate that his guess was correct.

There are a number of solutions which could be considered for this problem – One such solution for TPMs supporting symmetric encryption is specified in PKCS#7, section 10, and involves using the public key to encrypt a symmetric key, then using that symmetric key to encrypt the long message.

For TPMs without symmetric encryption capabilities, an alternative solution may be to add random padding to each message block, thus increasing the block’s entropy.

This normative was deleted, since it contradicted Part 3: “For a TPM_UNBIND command where the parent key has pubKey.algorithmId equal to TPM_ALG_RSA and pubKey.encScheme set to TPM_ES_RSAESPKCSv15 the TPM SHALL NOT expect a PAYLOAD_TYPE structure to prepend the decrypted data.” The contradiction was the case of a TPM_ES_RSAESPKCSv15 binding key, which does have a payload.

**End of informative comment**

1. The TPM MUST perform the encryption or decryption in accordance with the specification of the encryption scheme, as described below.
2. When a null terminated string is included in a calculation, the terminating null SHALL NOT be included in the calculation.

### 31.1.1 TPM_ES_RSAESOAEP_SHA1_MGF1

1. The encryption and decryption MUST be performed using the scheme RSA_ES_OAEP defined in [PKCS #1v2.0: 7.1] using SHA1 as the hash algorithm for the encoding operation.
2. Encryption
   a. The OAEP encoding P parameter MUST be the 4 character string “TCPA”.

b. While the TCG now controls this specification the string value will NOT change to allow for interoperability and backward compatibility with TCPA 1.1 TPM's

c. If there is an error with the encryption, the TPM must return the error TPM_ENCRYPT_ERROR.

3. Decryption

a. The OAEP decoding P parameter MUST be the 4 character string "TCPA".

b. While the TCG now controls this specification the string value will NOT change to allow for interoperability and backward compatibility with TCPA 1.1 TPM's

c. If there is an error with the decryption, the TPM must return the error TPM_DECRYPT_ERROR.

### 31.1.2 TPM_ES_RSAESPKCSV15

1. The encryption MUST be performed using the scheme RSA_ES_PKCSV15 defined in [PKCS #1v2.0: 7.2].

2. Encryption

a. If there is an error with the encryption, return the error TPM_ENCRYPT_ERROR.

3. Decryption

a. If there is an error with the decryption, return the error TPM_DECRYPT_ERROR.

### 31.1.3 TPM_ES_SYM_CTR

**Start of informative comment**

This defines an encryption mode in use with symmetric algorithms. The actual definition is at

http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf

The underlying symmetric algorithm may be AES128, AES192, or AES256. The definition for these algorithms is in the NIST document Appendix E.

The method of incrementing the counter value is different from that used by some standard crypto libraries (e.g. openSSL, Java JCE) that increment the entire counter value. TPM users should be aware of this to avoid errors when the counter wraps.

**End of informative comment**

1. Given a current counter value, the next counter value is obtained by treating the lower 32 bits of the current counter value as an unsigned 32-bit integer x, then replacing the lower 32 bits of the current counter value with the bits of the incremented integer (x + 1) mod 2^32. This method is described in Appendix B.1 of the NIST document (m=32).

### 31.1.4 TPM_ES_SYM_OFB

**Start of informative comment**

This defines an encryption mode in use with symmetric algorithms. The actual definition is at

http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf

The underlying symmetric algorithm may be AES128, AES192, or AES256. The definition for these algorithms is in the NIST document Appendix E.

**End of informative comment**

## 31.2 TPM Internal Digital Signatures

**Start of informative comment**

These values indicate the approved schemes in use by the TPM to generate digital signatures.

TPM 1.1 included only _SHA1 keys. These allowed the TPM_Sign command to sign a hash with no structure. This signature scheme is retained for backward compatibility.

TPM 1.2 added _INFO keys to ensure that a structure, rather than a plain hash, is always signed. For TPM_Sign, this signature scheme signs a new TPM_SIGN_INFO structure. Other ordinals, such as (e.g., TPM_GetAuditDigestSigned, TPM_CertifyKey, TPM_Quote, etc.) inherently sign a structure, so the _SHA1 and _INFO signature schemes produce an identical result.

**End of informative comment**

The TPM MUST perform the signature or verification in accordance with the specification of the signature scheme, as described below.

### 31.2.1 TPM_SS_RSASSAPKCS1v15_SHA1

**Start of informative comment**

This signature scheme prepends an OID to a SHA-1 digest. The OID, as specified in the normative, is as follows:

PKCS#1 v2.0: 8.1 says to encode the message per PKCS#1 v2.0: 9.2.1.

PKCS#1 v2.0: 9.2.1 says to apply the digest and then add the algorithm ID per Section 11.

PKCS#1 v2.0: Section 11.2.3 for SHA-1 says

{iso(1) identified-organization(3) oiw(14) secsig(3) algorithms(2) 26 }

and also

For each OID, the parameters field associated with this OID in an AlgorithmIdentifier shall have type NULL.

The DER/BER Guide says that the first sub-identifiers are coded as 40 * value1 + value2.

Thus, the OID becomes (with comments):

```
0x30 SEQUENCE
0x21 33 bytes
 0x30 SEQUENCE
 0x09 9 bytes
  0x06 OID
  0x05 5 bytes
   0x2b 43 = 40 * 1 (iso) + 3 (identified-organization)
   0x0e 14 from 11.2.3
   0x03  3 from 11.2.3
   0x02  2 from 11.2.3
   0x1a 26 from 11.2.3
  0x05 NULL (parameters)
  0x00 0 bytes
 0x04 OCTET
 0x14 20 bytes (the SHA-1 digest to follow)
```

**End of informative comment**

1. The signature MUST be performed using the scheme RSASSA-PKCS1-v1.5 defined in [PKCS #1v2.0: 8.1] using SHA1 as the hash algorithm for the encoding operation.

### 31.2.2 TPM_SS_RSASSAPKCS1v15_DER

**Start of informative comment**

This signature scheme is designed to permit inclusion of DER coded information before signing, which is inappropriate for most TPM capabilities

**End of informative comment**

1. The signature MUST be performed using the scheme RSASSA-PKCS1-v1.5 defined in [PKCS #1v2.0: 8.1]. The caller must properly format the area to sign using the DER rules. The provided area maximum size is k-11 octets.
2. TPM_Sign SHALL be the only TPM capability that is permitted to use this signature scheme. If a capability other than TPM_Sign is requested to use this signature scheme, it SHALL fail with the error code TPM_INAPPROPRIATE_SIG

### 31.2.3 TPM_SS_RSASSAPKCS1v15_INFO

**Start of informative comment**

This signature scheme is designed to permit signatures on arbitrary information but also protect the signature mechanism from being misused.

**End of informative comment**

1. The scheme MUST work just as TPM_SS_RSASSAPKCS1v15_SHA1 except in the TPM_Sign command
   a. In the TPM_Sign command the scheme MUST use a properly constructed TPM_SIGN_INFO structure, and hash it before signing

### 31.2.4 Use of Signature Schemes

**Start of informative comment**

The TPM_SS_RSASSAPKCS1v15_INFO scheme is a new addition for 1.2. It causes a new functioning for 1.1 and 1.2 keys. The following details the use of the new scheme and how the TPM handles signatures and hashing

**End of informative comment**

1. For commands that sign a TPM_SIGN_INFO structure (e.g., (TPM_GetAuditDigestSigned, TPM_TickStampBlob, TPM_ReleaseTransportSigned)

a. The TPM MUST create a TPM_SIGN_INFO and sign using the TPM_SS_RSASSAPKCS1v15_SHA1 scheme for either _SHA1 or _INFO keys.

2. For commands that sign a structure defined by the command (e.g., (TPM_CMK_CreateTicket, TPM_CertifyKey, TPM_CertifyKey2, TPM_MakeIdentity, TPM_Quote, TPM_Quote2, TPM_CertifySelfTest, TPM_GetCapabilitySigned)

a. Create the structure as defined by the command and sign using the TPM_SS_RSASSAPKCS1v15_SHA1 scheme for either _SHA1 or _INFO keys.

3. For TPM_Sign:

a. Create the structure as defined by the command and key scheme

b. If key->sigScheme is TPM_SS_RSASSAPKCS1v15_SHA1, sign the 20 byte parameter

c. If key->sigScheme is TPM_SS_RSASSAPKCS1v15_DER, sign the DER value.

d. If key->sigScheme is TPM_SS_RSASSAPKCS1v15_INFO, sign any value using the TPM_SIGN_INFO structure.

4. When data is signed and the data comes from INSIDE the TPM, the TPM MUST do the hash, and prepend the DER encoding correctly before performing the padding and private key operation.

5. When data is signed and the data comes from OUTSIDE the TPM, the software, not the TPM, MUST do the hash.

6. When the TPM knows, or is told by implication, that the hash used is SHA-1, the TPM MUST prepend the DER encoding correctly before performing the padding and private key operation

7. When the TPM does not know, or told by implication, that the hash used is SHA-1, the software, not the TPM) MUST provide the DER encoding to be prepended.

8. The TPM MUST perform the padding and private key operation in any signing operations it does.

# 32. Key Usage Table

**Start of informative comment**

Asymmetric keys (e.g., RSA keys) can do two basic functions: sign/verify and encrypt/decrypt.

TPM_KEY_SIGNING and TPM_KEY_IDENTITY do signature functions.

TPM_KEY_STORAGE, TPM_KEY_BIND, TPM_KEY_MIGRATE, and TPM_KEY_AUTHCHANGE do encryption functions.

**End of informative comment**

This table summarizes the types of keys associated with a given TPM command.

It is the responsibility of each command to check the key usage prior to executing the command

| | | | First Key | | | | | | Second Key | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Name | First Key | Second Key | SIGNING | STORAGE | IDENTITY | AUTHCHG | BIND | LEEGACY | SIGNING | STORAGE | IDENTITY | AUTHCHG | BIND | LEGACY |
| TPM_ActivateIdentity | idKey | | | | x | | | | | | | | | |
| TPM_CertifyKey | certKey | inKey | x | | x | | | x | x | x | x | | x | x |
| TPM_CertifyKey2 (Note 3) | inKey | certKey | x | x | x | | x | x | x | | x | | | x |
| TPM_CertifySelfTest | key | | x | | x | | | x | | | | | | |
| TPM_ChangeAuth | parent | blob | | x | | | | | 2 | 2 | 2 | 2 | 2 | 2 |
| TPM_ChangeAuthAsymFinish | parent | ephemeral | | x | | | | | | | | x | | |
| TPM_ChangeAuthAsymStart | idKey | ephemeral | | | x | | | | | | | x | | |
| TPM_CMK_ConvertMigration | parent | | | x | | | | | | | | | | |
| TPM_CMK_CreateBlob | parent | | | x | | | | | | | | | | |
| TPM_CMK_CreateKey | parent | | | x | | | | | | | | | | |
| TPM_ConvertMigrationBlob | parent | | | x | | | | | | | | | | |
| TPM_CreateMigrationBlob | parent | blob | | x | | | | | 2 | 2 | 2 | 2 | 2 | 2 |
| TPM_CreateWrapKey | parent | | | x | | | | | | | | | | |
| TPM_Delegate_CreateKeyDelegation | key | | x | x | x | x | x | x | | | | | | |
| TPM_DSAP | entity | | x | x | x | x | x | x | | | | | | |
| TPM_EstablishTransport | key | | | x | | | | x | | | | | | |
| TPM_GetAuditDigestSigned | certKey | | x | | x | | | x | | | | | | |
| TPM_GetAuditEventSigned | certKey | | x | | | | | x | | | | | | |
| TPM_GetCapabilitySigned | key | | x | | x | | | x | | | | | | |

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| TPM_GetPubKey | key | | x | x | x | x | x | x | | | | | | |
| TPM_KeyControlOwner | key | | x | x | x | | x | x | | | | | | |
| TPM_LoadKey2 | parent | inKey | | x | | | | | x | x | x | | x | x |
| TPM_LoadKey | parent | inKey | | x | | | | | x | x | x | | x | x |
| TPM_MigrateKey | maKey | | | 1 | | | | | | | | | | |
| TPM_OSAP | entity | | x | x | x | x | x | x | | | | | | |
| TPM_Quote | key | | x | | x | | | x | | | | | | |
| TPM_Quote2 | key | | x | | x | | | x | | | | | | |
| TPM_Seal | key | | | x | | | | | | | | | | |
| TPM_Sealx | key | | | x | | | | | | | | | | |
| TPM_Sign | key | | x | | | | | x | | | | | | |
| TPM_UnBind | key | | | | | | x | x | | | | | | |
| TPM_Unseal | parent | | | x | | | | | | | | | | |
| TPM_ReleaseTransportSigned | key | | x | | | | | | | | | | | |
| TPM_TickStampBlob | key | | x | | x | | | x | | | | | | |

## Notes

1 – Key is not a storage key but TPM_MIGRATE_KEY

2 – TPM unable to determine key type

3 – The order is correct; the reason is to support a single auth version.

TCG Published

# 33. Direct Anonymous Attestation

**Start of informative comment**

TPM_DAA_Join and TPM_DAA_Sign are highly resource intensive commands. They require most of the internal TPM resources to accomplish the complete set of operations. A TPM may specify that no other commands are possible during the join or sign operations. To allow other operations to occur, the TPM does allow the TPM_SaveContext command to save off the current join or sign operation.

Operations that occur during a join or sign result in the loss of the join or sign session in favor of the interrupting command.

**End of informative comment**

1. The TPM MUST support one concurrent TPM_DAA_Join or TPM_DAA_Sign session. The TPM MAY support additional sessions
2. The TPM MAY invalidate a join or sign session upon the receipt of any additional command other than the join/sign or TPM_SaveContext

## 33.1 TPM_DAA_JOIN

**Start of informative comment**

TPM_DAA_Join creates new JOIN data. If a TPM supports only one JOIN/SIGN operation, TPM_DAA_Join invalidates any previous DAA attestation information inside a TPM. The JOIN phase of a DAA context requires a TPM to communicate with an issuer. TPM_DAA_Join outputs data to be sent to an issuing authority and receives data from that issuing authority. The operation potentially requires several seconds to complete, but is done in a series of atomic stages and TPM_SaveContext/TPM_LoadContext can be used to cache data off-TPM in between atomic stages.

The JOIN process is designed so a TPM will normally receive exactly the same DAA credentials from a given issuer, no matter how many times the JOIN process is executed and no matter whether the issuer changes his keys. This property is necessary because an issuer must give DAA credentials to a platform after verifying that the platform has the architecture of a trusted platform. Unless the issuer repeats the verification process, there is no justification for giving different DAA credentials to the same platform. Even after repeating the verification process, the issuer should give replacement (different) DAA credentials only when it is necessary to retire the old DAA credentials. Replacement DAA credentials erase the previous DAA history of the platform, at least as far as the DAA credentials from that issuer are concerned. Replacement might be desirable, as when a platform changes hands, for example, in order to eliminate any association via DAA between the seller and the buyer. On the other hand, replacement might be undesirable, since it enables a rogue to rejoin a community from which he has been barred. Replacement is done by submitting a different “count” value to the TPM during a JOIN process. A platform may use any value of “count” at any time, in any order, but only “counts” accepted by the issuer will elicit DAA credentials from that issuer.

The TPM is forced to verify an issuer’s public parameters before using an issuer’s public parameters. This verification provides proof that the public parameters (which include a public key) were approved by an entity that knows the private key corresponding to that public key; in other words that the JOIN has previously been approved by the issuer. This verification is necessary to prevent an attack by a rogue using a genuine issuer's public parameters, which could reveal the secret created by the TPM using those public parameters. Verification uses a signature (provided by the issuer) over the public parameters.

The exponent of the issuer's key is fixed at 2^16+1, because this is the only size of exponent that a TPM is required to support. The modulus of the issuer's public key is used to create the pseudonym with which the TPM contacts the issuer. Hence, the TPM cannot produce the same pseudonym for different issuers (who have different keys). The pseudonym is always created using the issuer's first key, even if the issuer changes keys, in order to produce the property described earlier. The issuer proves to the TPM that he has the right to use that first key to create a pseudonym by creating a chain of signatures from the first key to the current key, and submitting those signatures to the TPM. The method has the desirable property that only signatures and the most recent private key need be retained by the issuer: once the latest link in the signature chain has been created, previous private keys can be discarded.

The use of atomic operations minimizes the contiguous time that a TPM is busy with TPM_DAA_Join and hence unavailable for other commands. JOIN can therefore be done as a background activity without inconveniencing a user. The use of atomic operations also minimizes the peak value of TPM resources consumed by the JOIN phase.

The use of atomic operations introduces a need for consistency checks, to ensure that the same parameters are used in all atomic operations of the same JOIN process. DAA_tpmSpecific therefore contains a digest of the associated DAA_issuerSettings structure, and DAA_session contains a digest of associated DAA_tpmSpecific and DAA_joinSession structures. Each atomic operation verifies digests to ensure use of mutually consistent sets of DAA_issuerSettings, DAA_tpmSpecific, DAA_session, and DAA_joinSession data.

JOIN operations and data structures are designed to minimize the amount of data that must be stored on a TPM in between atomic operations, while ensuring use of mutually consistent sets of data. Digests of public data are held in the TPM between atomic operations, instead of the actual public data (if a digest is smaller than the actual data). In each atomic operation, consistency checks verify that any public data loaded and used in that operation matches the stored digest. Thus non-secret DAA_generic_X parameters (loaded into the TPM only when required), are checked using digests DAA_digest_X (preloaded into the TPM in the structure DAA_issuerSettings).

JOIN includes a challenge from the issuer, in order to defeat simple Denial of Service attacks on the issuer's server by rogues pretending to be arbitrary TPMs.

A first group of atomic operations generate all TPM-data that must be sent to the issuer. The platform performs other operations (that do not need to be trusted) using the TPM-data, and sends the resultant data to the issuer. The issuer sends values u2 and u3 back to the TPM. A second group of atomic operations accepts this data from the issuer and completes the protocol.

The TPM outputs encrypted forms of DAA_tpmSpecific, v0 and v1. These encrypted data are later interpreted by the same TPM and not by any other entity, so any manufacturer-specific wrapping can be used. It is suggested, however, that enc(DAA_tpmSpecific) or enc(v0) or enc(v1) data should be created by adapting a TPM_CONTEXT_BLOB structure.

After executing TPM_DAA_Join, it is prudent to perform TPM_DAA_Sign, to verify that the JOIN process completed correctly. A host platform may choose to verify JOIN by performing TPM_DAA_Sign as both the target and the verifier (or could, of course, use an external verifier).

**End of informative comment**

## 33.2 TPM_DAA_Sign

**Start of informative comment**

TPM_DAA_Sign responds to a challenge and proves the attestation held by a TPM without revealing the attestation held by that TPM. The operation is done in a series of atomic stages to minimize the contiguous time that a TPM is busy and hence unavailable for other commands. TPM_SaveContext can be used to save a DAA context in between atomic stages. This enables the response to the challenge to be done as a background activity without inconveniencing a user, and also minimizes the peak value of TPM resources consumed by the process.

The use of atomic operations introduces a need for consistency checks, to ensure that the same parameters are used in all atomic operations of the same SIGN process. DAA_tpmSpecific therefore contains a digest of the associated DAA_issuerSettings structure, and DAA_session contains a digest of associated DAA_tpmSpecific structure. Each atomic operation verifies these digests and hence ensures use of mutually consistent sets of DAA_issuerSettings, DAA_tpmSpecific, and DAA_session data.

SIGN operations and data structures are designed to minimise the amount of data that must be stored on a TPM in between atomic operations, while ensuring use of mutually consistent sets of data. Digests of public and private data are held in the TPM between atomic operations, instead of the actual public or private data (if a digest is smaller than the actual data). At each atomic operation, consistency checks verify that any data loaded and used in that operation matches the stored digest. Thus parameters DAA_digest_X are digests (preloaded into the TPM in the structure DAA_issuerSettings) of non-secret DAA_generic_X parameters (loaded into the TPM only when required), for example.

The design enables the use of any number of issuer DAA-data, private DAA-data, and so on. Strictly, the design is that the *TPM* puts no limit on the number of sets of issuer DAA-data or sets of private DAA-data, or restricts what set is in the TPM at any time, but supports only one DAA-context in the TPM at any instant. Any number of DAA-contexts can, of course, be swapped in and out of the TPM using TPM_SaveContext/TPM_LoadContext, so applications do not perceive a limit on the number of DAA contexts.

TPM_DAA_Sign accepts a freshness challenge from the verifier and generates all TPM-data that must be sent to the verifier. The platform performs other operations (that do not need to be trusted) using the TPM-data, and sends the resultant data to the verifier. At one stage, the TPM incorporates a loaded public (non-migratable) key into the protocol. This is intended to permit the setup of a session, for any specific purpose, including doing the same job in TPM_ActivateIdentity as the EK.

**End of informative comment**

## 33.3 DAA Command summary

**Start of informative comment**

The following is a conceptual summary of the operations that are necessary to setup a TPM for DAA, execute the JOIN process, and execute the SIGN process.

The summary is partitioned according to the "stages" of the actual TPM commands. Thus, the operations listed in JOIN under stage-2 briefly describe the operation of TPM_DAA_Join at stage-2, for example.

This summary is in place to help in the connection between the mathematical definition of DAA and this implementation in a TPM.

**End of informative comment**

### 33.3.1 TPM setup

1. A TPM generates a TPM-specific secret S (160-bit) from the RNG and stores S in nonvolatile store on the TPM. This value will never be disclosed and changed by the TPM.

### 33.3.2 JOIN

**Start of informative comment**

This entire section is informative

1. When the following is performed, this process does not increment the stage counter.

a. TPM imports a non-secret values n0 (2048-bit).

b. TPM computes a non-secret value N0 (160-bit) = H(n0).

c. TPM computes a TPM-specific secret DAA_rekey (160-bit) = H(S, H(n0)).

d. TPM stores a self-consistent set of (N0, DAA_rekey)

2. The following is performed 0 or several times: (Note: If the stage mechanism is being used, then this branch does not increment the stage counter.)

a. TPM imports

i. a self consistent set of (N0, DAA_rekey)

ii. a non-secret value DAA_SEED_KEY (2048-bit)

iii. a non-secret value DEPENDENT_SEED_KEY (2048-bit)

iv. a non-secret value SIG_DSK (2048-bit)

b. TPM computes DIGEST (160-bit) = H(DAA_SEED_KEY)

c. If DIGEST != N0, TPM refuses to continue

d. If DIGEST == N0, TPM verifies validity of signature SIG_DSK on DEPENDENT_SEED_KEY with key (DAA_SEED_KEY, e0 (= 2^16 + 1)) by using TPM_Sign_Verify (based on PKCS#1 2.0). If check fails, TPM refuses to continue.

e. TPM sets N0 = H(DEPENDENT_SEED_KEY)

f. TPM stores a self consistent set of (N0, DAA_JOIN)

3. Stage 2

a. TPM imports a set of values, including

i. a non-secret value n0 (2048-bit),

ii. a non-secret value R0 (2048-bit),

iii. a non-secret value R1 (2048-bit),

iv. a non-secret value S0 (2048-bit),

v. a non-secret value S1 (2048-bit),

vi. a non-secret value n (2048-bit),

vii. a non-secret value n1 (1024-bit),

viii. a non-secret value gamma (2048-bit),

ix. a non-secret value q (208-bit),

x. a non-secret value COUNT (8-bit),

xi. a self consistent set of (N0, DAA_rekey).

xii. TPM saves them as part of a new set A.

b. TPM computes DIGEST (160-bit) = H(n0)

c. If DIGEST != N0, TPM refuses to continue.

d. If DIGEST == N0, TPM computes DIGEST (160-bit) = H(R0, R1, S0, S1, n, n1, Γ, q)

e. TPM imports a non-secret value SIG_ISSUER_KEY (2048-bit).

f. TPM verifies validity of signature SIG_ISSUER_KEY (2048-bit) on DIGEST with key (n0, e0) by using TPM_Sign_Verify (based on PKCS#1 2.0). If check fails, TPM refuses to continue.

g. TPM computes a TPM-specific secret f (208-bit) = H(DAA_rekey, COUNT, 0)|| H(DAA_rekey, COUNT, 1) mod q.

h. TPM computes a TPM-specific secret f0 (104-bit) = f mod 2104.

i. TPM computes a TPM-specific secret f1 (104-bit) = f >> 104.

j. TPM save f, f0 and f1 as part of set A.

4. Stage 3

a. TPM generates a TPM-specific secret u0 (1024-bit) from the RNG.

b. TPM generates a TPM-specific secret u'1 (1104-bit) from the RNG.

c. TPM computes u1 (1024-bit) = u'1 mod n1.

d. TPM stores u0 and u1 as part of set A.

5. Stage 4

a. TPM computes a non-secret value P1 (2048-bit) = (R0^f0) mod n and stores P1 as part of set A.

6. Stage 5

a. TPM computes a non-secret value P2 (2048-bit) = P1*(R1^f1) mod n, stores P2 as part of set A and erases P1 from set A.

7. Stage 6

a. TPM computes a non-secret value P3 (2048-bit) = P2*(S0^u0) mod n, stores P3 as part of set A and erases P2 from set A.

8. Stage 7

a. TPM computes a non-secret value U (2048-bit) = P3*(S1^u1) mod n.

b. TPM erases P3 from set A

c. TPM computes and saves U1 (160-bit) = H(U||COUNT||N0) as part of set A.

d. TPM exports U.

9. Stage 8

a. TPM imports ENC_NE (2048-bit).

b. TPM decrypts NE (160-bit) from ENC_NE (2048-bit) by using privEK: NE = decrypt(privEK, ENC_NE).

c. TPM computes U2 (160-bit) = H(U1||NE).

d. TPM erases U1 from set A.

e. TPM exports U2.

10. Stage 9

a. TPM generates a TPM-specific secret r0 (344-bit) from the RNG.

b. TPM generates a TPM-specific secret r1 (344-bit) from the RNG.

c. TPM generates a TPM-specific secret r2 (1024-bit) from the RNG.

d. TPM generates a TPM-specific secret r3 (1264-bit) from the RNG.

e. TPM stores r0, r1, r2, r3 as part of set A.

f. TPM computes a non-secret value P1 (2048-bit) = (R0^r0) mod n and stores P1 as part of set A.

11. Stage 10

a. TPM computes a non-secret value P2 (2048-bit) = P1*(R1^r1) mod n, stores P2 as part of set A and erases P1 from set A.

12. Stage 11

a. TPM computes a non-secret value P3 (2048-bit) = P2*(S0^r2) mod n, stores P3 as part of set A and erases P2 from set A.

13. Stage 12

a. TPM computes a non-secret value P4 (2048-bit) = P3*(S1^r3) mod n, stores P4 as part of set A and erases P3 from set A.

b. TPM exports P4.

14. Stage 13

a. TPM imports w (2048-bit).

b. TPM computes w1 = w^q mod Γ.

c. TPM verifies if w1 = 1 holds. If it doesn't hold, TPM refuses to continue.

d. If it does hold, TPM saves w as part of set A.

15. Stage 14

a. TPM computes a non-secret value E (2048-bit) = w^f mod Γ.

b. TPM exports E.

16. Stage 15

a. TPM computes a TPM-specific secret r (208-bit) = r0 + 2^104*r1 mod q.

b. TPM computes a non-secret value E1 (2048-bit) = w^r mod Γ.

c. TPM exports E1 and erases w from set A.

17. Stage 16

a. TPM imports a non-secret value c1 (160-bit).

b. TPM generates a non-secret value NT (160-bit) from the RNG.

c. TPM computes a non-secret value c (160-bit) = H(c1||NT).

d. TPM save c as part of set A.

e. TPM exports NT

18. Stage 17

a. TPM computes a non-secret value s0 (352-bit) = r0 + c*f0 over the integers.

b. TPM exports s0.

19. Stage 18

a. TPM computes a non-secret value s1 (352-bit) = r1 + c*f1 over the integers.

b. TPM exports s1.

20. Stage 19

a. TPM computes a non-secret value s2 (1024-bit) = r2 + c*u0 mod 21024.

b. TPM exports s2.

21. Stage 20

a. TPM computes a non-secret value s'2 (1024-bit) = (r2 + c*u0) >> 1024 over the integers.

b. TPM saves s'2 as part of set A.

c. TPM exports c

22. Stage 21

a. TPM computes a non-secret value s3 (1272-bit) = r3 + cu1 + s'2 over the integers.

b. TPM exports s3 and erases s'2 from set A.

23. Stage 22

a. TPM imports a non-secret value u2 (1024-bit).

b. TPM computes a TPM-specific secret v0 (1024-bit) = u2 + u0 mod 21024.

c. TPM stores v0 as part of A.

d. TPM computes a TPM-specific secret v'0 (1024-bit) = (u2 + u0) >> 1024 over the integers.

e. TPM saves v'0 as part of set A.

24. Stage 23

a. TPM imports a non-secret value u3 (1512-bit).

b. TPM computes a TPM-specific secret v1 (1520-bit) = u3 + u1 + v'0 over the integers.

c. TPM stores v1 as part of A.

d. TPM erases v'0 from set A.

25. Stage 24

a. TPM makes self-consistent set of all the data (n0, COUNT, R0, R1, S0, S1, n, Γ, q, v0, v1), where the values v0, v1 are secret – they need to be stored safely with the consistent set, and the remaining is non-secret.

b. TPM erases set A.

**End of informative comment**

### 33.3.3 SIGN

**Start of informative comment**

This entire section is informative

1. Stage 0 & 1

a. TPM imports and verifies a self-consistent set of all the data including:

i. a non-secret value n0 (2048-bit),

ii. a non-secret value COUNT (8-bit),

iii. a non-secret value R0 (2048-bit),

iv. a non-secret value R1 (2048-bit),

v. a non-secret value S0 (2048-bit),

vi. a non-secret value S1 (2048-bit),

vii. a non-secret value n (2048-bit),

viii. a non-secret value gamma (2048-bit),

ix. a non-secret value q (208-bit),

x. v0 (1024-bit),

xi. v1 (1520-bit).

xii. If the verification does not succeed, TPM refuses to continue.

b. TPM stores the above values as part of a new set A.

c. TPM computes a TPM-specific secret f0 (104-bit) = f mod 2104.

d. TPM computes a TPM-specific secret f1 (104-bit) = f >> 104.

e. TPM stores f0 and f1 as part of set A.

f. TPM generates a TPM-specific secret r0 (344-bit) from the RNG.

g. TPM generates a TPM-specific secret r1 (344-bit) from the RNG.

h. TPM generates a TPM-specific secret r2 (1024-bit) from the RNG.

i. TPM generates a TPM-specific secret r4 (1752-bit) from the RNG.

j. TPM stores r0, r1, r2, r4, as part of set A.

2. Stage 2

a. TPM computes a non-secret value P1 (2048-bit) = (R0^r0) mod n and stores P1 as part of set A.

3. Stage 3

a. TPM computes a non-secret value P2 (2048-bit) = P1*(R1^r1) mod n, stores P2 as part of set A and erases P1 from set A.

4. Stage 4

a. TPM computes a non-secret value P3 (2048-bit) = P2*(S0^r2) mod n, stores P3 as part of set A and erases P2 from set A.

5. Stage 5

a. TPM computes a non-secret value T (2048-bit) = P3*(S1^r4) mod n.

b. TPM erases P3 from set A.

c. TPM exports T.

6. Stage 6

a. TPM imports a non-secret value w (2048-bit).

b. TPM computes w1 = w^q mod Γ.

c. TPM verifies if w1 = 1 holds. If it doesn't hold, TPM refuses to continue.

d. If it does hold, TPM saves w as part of set A.

7. Stage 7

a. TPM computes a non-secret value E (2048-bit) = w^f mod Γ.

b. TPM exports E and erases f from set A.

8. Stage 8

a. TPM computes a TPM-specific secret r (208-bit) = r0 + 2^104*r1 mod q.

b. TPM computes a non-secret value E1 (2048-bit) = w^r mod Γ.

c. TPM exports E1 and erases w and E1 from set A.

9. Stage 9

a. TPM imports a non-secret value c1 (160-bit).

b. TPM generates a non-secret value NT (160-bit) from the RNG.

c. TPM computes a non-secret value c2 (160-bit) = H(c1||NT) and erases c1 from set A.

d. TPM saves c2 as part of set A.

e. TPM exports NT.

10. Stage 10

a. TPM imports a non-secret value b (1-bit).

b. If b = = 1, TPM imports a non-secret value m (160-bit).

c. TPM computes a non-secret value c (160-bit) = H(c2||b||m) and erases c2 from set A.

d. If b = = 0, TPM imports an RSA public key, eAIK (= 2^16 + 1) and nAIK (2048-bit).

e. TPM computes a non-secret value c (160-bit) = H(c2||b||nAIK) and erases c2 from set A.

f. TPM exports c.

11. Stage 11

a. TPM computes a non-secret value s0 (352-bit) = r0 + c*f0 over the integers.

b. TPM exports s0.

12. Stage 12

a. TPM computes a non-secret value s1 (352-bit) = r1 + c*f1 over the integers.

b. TPM exports s1.

13. Stage 13

a. TPM computes a non-secret value s2 (1024-bit) = r2 + c*v0 mod 21024.

b. TPM exports s2.

14. Stage 14

a. TPM computes a non-secret value s'2 (1024-bit) = (r2 + c*v0) >> 1024 over the integers.

b. TPM saves s'2 as part of set A.

15. Stage 15

a. TPM computes a non-secret value s3 (1760-bit) = r4 + cv1 + s'2 over the integers.

b. TPM exports s3 and erases s'2 from set A.

c. TPM erases set A.

**End of informative comment**

# 34. General Purpose IO

**Start of informative comment**

The GPIO capability allows an outside entity to output a signal on a GPIO pin, or read the status of a GPIO pin. The solution is for a single pin, with no timing information. There is no support for sending information on specific busses like SMBus or RS232. The design does support the designation of more than one GPIO pin.

There is no requirement as to the layout of the GPIO pin, or the routing of the wire from the GPIO pin on the platform. A platform specific specification can add those requirements.

To avoid the designation of additional command ordinals, the architecture uses the NV Storage commands. A set of GPIO NV indexes map to individual GPIO pins. TPM_NV_INDEX_GPIO_00 maps to the first GPIO pin. The platform specific specification indicates the mapping of GPIO zero to a specific package pin.

The TPM does not reserve any NV storage for the indicated pin; rather the TPM uses the authorization mechanisms for NV storage to allow a rich set of controls on the use of the GPIO pin. The TPM owner can specify when and how the platform can use the GPIO pin. While there is no NV storage for the pin value, TRUE or FALSE, there is NV storage for the authorization requirements for the pin.

Using the NV attributes the GPIO pin may be either an input pin or an output pin.

**End of informative comment**

1. The TPM MAY support the use of a GPIO pin defined by the NV storage mechanisms.
2. The GPIO pin MAY be either an input or an output pin.

# 35. Redirection

**Informative comment**

Redirection allows the TPM to output the results of operations to hardware other than the normal TPM communication bus. The redirection can occur to areas internal or external to the TPM. Redirection is only available to key operations (such as TPM_UnBind, TPM_Unseal, and TPM_GetPubKey). To use redirection the key must be created specifying redirection as one of the keys attributes.

When redirecting the output the TPM will not interpret any of the data and will pass the data on without any modifications.

The TPM_SetRedirection command connects a destination location or port to a loaded key. This connection remains so long as the key is loaded, and is saved along with other key information on a saveContext(key), loadContext(key). If the key is reloaded using TPM_LoadKey, then TPM_SetRedirection must be run again.

Any use of TPM_SetRedirection with a key that does not have the redirect attribute must return an error. Use of key that has the redirect attribute without TPM_SetRedirection being set must return an error.

**End of informative comments**

1. The TPM MAY support redirection
2. If supported, the TPM MUST only use redirection on keys that have the redirect attribute set
3. A key that is tagged as a “redirect” key MUST be a leaf key in the TPM Protected Storage blob hierarchy. A key that is tagged as a “redirect” key CAN NEVER be a parent key.
4. Output data that is the result of a cryptographic operation using the private portion of a “redirect” key:
    a. MUST be passed to an alternate output channel
    b. MUST NOT be passed to the normal output channel
    c. MUST NOT be interpreted by the TPM
5. When command input or output is redirected the TPM MUST respond to the command as soon as the ordinal finishes processing
    a. The TPM MUST indicate to any subsequent commands that the TPM is busy and unable to accept additional command until the redirection is complete
    b. The TPM MUST allow for the resetting of the redirection channel
6. Redirection MUST be available for the following commands:
    a. TPM_Unseal
    b. TPM_UnBind
    c. TPM_GetPubKey
    d. TPM_Seal
    e. TPM_Quote

# 36. Structure Versioning

**Start of informative comment**

In version 1.1 some structures also contained a version indicator. The TPM set the indicator to indicate the version of the TPM that was creating the structure. This was incorrect behavior. The functionality of determining the version of a structure is radically different in 1.2.

Most structures will contain a TPM_STRUCTURE_TAG. All future structures must contain the tag, the only structures that do not contain the tag are 1.1 structures that are not modified in 1.2. This restriction keeps backwards compatibility with 1.1.

Any 1.2 structure must not contain a 1.1 tagged structure. For instance the TPM_KEY complex, if set at 1.2, must not contain a PCR_INFO structure. The TPM_KEY 1.2 structure must contain a PCR_INFO_LONG structure. The converse is also true 1.1 structures must not contain any 1.2 structures.

The TPM must not allow the creation of any mixed structures. This implies that a command that deals with keys, for instance, must ensure that a complete 1.1 or 1.2 structure is properly built and validated on the creation and use of the key.

The tag structure is set as a UINT16. This allows for a reasonable number of structures without wasting space in the buffers.

To obtain the current TPM version the caller must use the TPM_GetCapability command.

The tag is not a complete validation of the validity of a structure. The tag provides a reference for the structure and the TPM or caller is responsible for determining the validity of any remaining fields. For instance, in the TPM_KEY structure, the tag would indicate TPM_KEY but the TPM would still use tpmProof and the various digests to ensure the structure integrity.

7. Compatibility and notification

In 1.1 TPM_CAP_VERSION (index 19) returned a version structure with 1.1.x.x. The x.x was for manufacturer information and the x.x also was set version structures. In 1.2 TPM_CAP_VERSION will return 1.1.0.0. Any 1.2 structure that uses the version information will set the x.x to 0.0 in the structure. TPM_CAP_MANUFACTURER_VER (index 21) will return 1.2.x.x. The 1.2 structures do not contain the version structure. The rationale behind this is that the structure tag will indicate the version of the structure. So changing a correct structure will result in a new tag and there is no need for a separate version structure.

For further compatibility, the quote function always returns 1.1.0.0 in the version information regardless of the size of the incoming structure. All other functions may regard a 2 byte sizeofselect structure as indicative of a 1.1 structure. The TPM handles all of the structures according to the input, the only exception being TPM_CertifyKey where the TPM does not need to keep the input version of the structure.

**End of informative comment**

1. The TPM MUST support 1.1 and 1.2 defined structures

2. The TPM MUST ensure that 1.1 and 1.2 structures are not mixed in the same overall structure

a. For instance in the TPM_KEY structure if the structure is 1.1 then PCR_INFO MUST be set and if 1.2 the PCR_INFO_LONG structure must be set

3. On input the TPM MUST ignore the lower two bytes of the version structure

4. On output the TPM MUST set the lower two bytes to 0 of the version structure

# 37. Certified Migration Key Type

**Start of informative comment**

In version 1.1 there were two key types, non-migration and migration keys. The TPM would only certify non-migrating keys. There is a need for a key that allows migration but allows for certification. This proposal is to create a key that allows for migration but still has properties that the TPM can certify.

These new keys are "certifiable migratable keys" or CMK. This designation is to separate the keys from either the normal migration or non-migration types of keys. The TPM Owner is not required to use these keys.

Two entities may participate in the CMK process. The first is the Migration-Selection Authority and the second is the Migration Authority (MA).

**Migration Selection Authority (MSA)**

The MSA controls the migration of the key but does not handle the migrated itself.

**Migration Authority (MA)**

A Migration Authority actually handles the migrated key.

**Use of MSA and MA**

Migration of a CMK occurs using TPM_CMK_CreateBlob (TPM_CreateMigrationBlob cannot be used). The TPM Owner authorizes the migration destination (as usual), and the key owner authorizes the migration transformation (as usual). An MSA authorizes the migration destination as well. If the MSA is the migration destination, no MSA authorization is required.

**End of informative comment**

## 37.1 Certified Migration Requirements

**Start of informative comment**

The following list details the design requirements for the controlled migration keys

**Key Protections**

The key must be protected by hardware and an entity trusted by the key user.

**Key Certification**

The TPM must provide a mechanism to provide certification of the key protections (both hardware and trusted entity)

**Owner Control**

The TPM Owner must control the selection of the trusted entity

**Control Delegation**

The TPM Owner may delegate the ability to create the keys but the decision must be explicit

**Linkage**

The architecture must not require linking the trusted entity and the key user

**Key Type**

The key may be any type of migratable key (storage or signing)

**Interaction**

There must be no required interaction between the trusted entity and the TPM during the key creation process

**End of informative comment**

## 37.2 Key Creation

**Start of informative comment**

The command TPM_CMK_CreateKey creates a CMK where control of the migration is by a MSA or MA. The process uses the MSA public key (actually a digest of the MA public key) as input to TPM_CMK_CreateKey. The key creation process establishes a migrationAuth that is SHA-1(tpmProof || SHA-1(MA pubkey) || SHA-1(source pubkey)).

The use of tpmProof is essential to prove that CMK creation occurs on a TPM. The use of "source pubkey" explicitly links a migration AuthData value to a particular public key, to simplify verification that a specific key is being migrated.

**End of informative comment**

## 37.3 Migrate CMK to a MA

**Start of informative comment**

Migration of a CMK to a destination other than the MSA:

**TPM_MIGRATIONKEYAUTH Creation**

The TPM Owner authorizes the creation of a TPM_MIGRATIONKEYAUTH structure using TPM_AuthorizeMigrationKey command. The structure contains the destination migrationKey, the migrationScheme (which must be set to TPM_MS_RESTRICT_MIGRATE or TPM_MS_RESTRICT_APPROVE) and a digest of tpmProof.

**MA Approval**

The MA signs a TPM_CMK_AUTH structure, which contains the digest of the MA public key, the digest of the destination (or parent) public key and a digest of the public portion of the key to be migrated

**TPM Owner Authorization**

The TPM Owner authorizes the MA approval using TPM_CMK_CreateTicket and produces a signature ticket

**Key Owner Authorization**

The CMK owner passes the TPM Owner MA authorization, the MSA Approval and the signature ticket to the TPM_CMK_CreateBlob using the key owners authorization.

Thus the TPM owner, the key's owner, and the MSA, all cooperate to migrate a key produced by TPM_CMK_CreateBlob.

**End of informative comment**

## 37.4 Migrate CMK to a MSA

**Start of informative comment**

Migrate CMK directly to a MSA

**TPM_MIGRATIONKEYAUTH Creation**

The TPM Owner authorizes the creation of a TPM_MIGRATIONKEYAUTH structure using TPM_AuthorizeMigrationKey command. The structure contains the destination migrationKey (which must be the MSA public key), the migrationScheme (which must be set to TPM_MS_RESTRICT_MIGRATE) and a digest of tpmProof.

**Key Owner Authorization**

The CMK owner passes the TPM_MIGRATIONKEYAUTH to the TPM in a TPM_CMK_CreateBlob using the CMK owner authorization.

**Double Wrap**

If specified, through the MS_MIGRATE scheme, the TPM double wraps the CMK information such that the only way a recipient can unwrap the key is with the cooperation of the CMK owner.

**Proof of Control**

To prove to the MA and to a third party that migration of a key is under MSA control, a caller passes the MA's public key (actually its digest) to TPM_CertifyKey, to create a TPM_CERTIFY_INFO structure. This now contains a digest of the MA's public key.

A CMK be produced without cooperation from the MA: the caller merely provides the MSA's public key. When the restricted key is to be migrated, the public key of the intended destination, plus the CERTIFY_INFO structure are sent to the MSA. The MSA extracts the migrationAuthority digest from the CERTIFY_INFO structure, verifies that migrationAuthority corresponds to the MSA's public key, creates and signs a TPM_RESTRICTEDKEYAUTH structure, and sends that signature back to the caller. Thus the MSA never needs to touch the actual migrated data.

**End of informative comment**

# 38. Revoke Trust

**Start of informative comment**

There are circumstances where clearing all keys and values within the TPM is either desirable or necessary. These circumstances may involve both security and privacy concerns.

Platform trust is demonstrated using the EK Credential, Platform Credential and the Conformance Credentials. There is a direct and cryptograph relationship between the EK and the EK Credential and the Platform Credential. The EK and Platform credentials can only demonstrate platform trust when they can be validated by the Endorsement Key.

This command is called revoke trust because, by deleting the EK, the EK Credential and the Platform Credential are dissociated from the platform, therefore invalidating them and resulting in the revocation of the trust in the platform. From a trust perspective, the platform associated with these specific credentials no longer exists. However, any transaction that occurred prior to invoking this command will remain valid and trusted to the same extent they would be valid and trusted if the platform were physically destroyed.

This is a non-reversible function. Also, along with the EK, the Owner is also deleted, removing all non-migratable keys and owner-specified state.

It is possible to establish new trust in the platform by creating a new EK using the TPM_CreateRevocableEK command. Establishing trust in the platform, however, is more than just creating the EK. The EK Credential and the Platform Credential must also be created and associated with the new EK as described above. (The conformance credentials may be obtained from the TPM and Platform manufacturer.) These credentials must be created by an entity that is trusted by those entities interested in the trust of the platform. This may not be a trivial task. For example, an entity willing to create these credentials my want to examine the platform and require physical access during the new EK generation process.

Besides calling one of the two EK creation functions to create the EK, the EK may be "squirted" into the TPM by an external source. If this method is used, tight controls must be placed on the process used to perform this function to prevent exposure or intentional duplication of the EK. Since the revocation and re-creation of the EK are functions intended to be performed after the TPM leaves the trusted manufacturing process, squirting of the EK must be disallowed after the manufacturing process if the revoke trust command is executed.

**End of informative comment**

1. If TPM_CreateRevokableEK and TPM_RevokeTrust are implemented, one can do an unrestricted number of TPM_CreateRevokableEK / TPM_RevokeTrust pairs until TPM_CreateEndorsementKeyPair is called. After TPM_CreateEndorsementKeyPair is called, the EK becomes irrevocable.
2. After an EK is created the TPM MUST NOT allow a new EK to be "squirted" for the lifetime of the TPM.
3. The EK Credential MUST provide an indication within the EK Credential as to how the EK was created. The valid permutations are:
   a. Squirted, non-revocable

b. Squirted, revocable

c. Internally generated, non-revocable

d. Internally generated, revocable

4. If the method for creating the EK during manufacturing is squirting, the EK may be either non-revocable or revocable. If it is revocable, the method must provide the insertion or extraction of the EKreset value.

# 39. Mandatory and Optional Functional Blocks

**Start of informative comment**

This section lists the main functional blocks of a TPM (in arbitrary order), states whether that block is mandatory or optional in the main TPM specification, and provides brief justification for that choice.

Important notes:

1. The default classification of a TPM function block is “mandatory”, since reclassification from mandatory to optional enables the removal of a function from existing implementations, while reclassification from optional to mandatory may require the addition of functionality to existing implementations.

2. Mandatory functions will be reclassified as optional functions if those functions are not required in some particular type of TCG trusted platform.

3. If a functional block is mandatory in the main specification, the functionality must be present in all TCG trusted platforms.

4. If a functional block is optional in the main specification, each individual platform-specific specification must declare the status of that functionality as either (1) “mandatory-specific” (the functionality must be present in all platforms of that type), or (2) “optional-specific” (the functionality is optional in that type of platform), or (3) “excluded-specific” (the functionality must not be present in that type of platform).

**End of informative comment**

Classification of TPM functional blocks

1. Legacy (v1.1b) features
    a. Anything that was mandatory in v1.1b continues to be mandatory in v1.2. Anything that was optional in v1.1b continues to be optional in v1.2.
    b. V1.2 must be backwards compatible with v1.1b. All TPM features in v1.1b were discussed in depth when v1.1b was written, and anything that wasn't thought strictly necessary was tagged as "optional".
2. Number of PCRs
    a. The platform specific specification controls the number of PCR on a platform. The TPM MUST implement the mandatory number of PCR specified for a particular platform
        i. TPMs designed to work on multiple platforms MUST provide the appropriate number of TPM for all intended platforms. I.e. if one platform requires 16 PCR and the other platform 24 the TPM would have to supply 24 PCR.
    b. For TPMs providing backwards compatibility with 1.1 TPM on the PC platform, there MUST be 16 static PCR.
3. Sessions
    a. The TPM MUST support a minimum of 3 active sessions
        i. Active means currently loaded and addressable inside the TPM

      ii. Without 3 active sessions many TPM commands cannot function

   b. The TPM MUST support a minimum of 16 concurrent sessions

      i. The contextList of currently available session has a minimum size of 16

      ii. Providing for more concurrent sessions allows the resource manager additional flexibility and speed

4. NVRAM

   a. There are 20 bytes mandatory of NVRAM in v1.2 as specified by the main specification. A platform specific specification can require a larger amount of NVRAM

   b. Cost is important. The mandatory amount of NVRAM must be as small as possible, because different platforms will require different amounts of NVRAM. 20 bytes are required for (DIR) backwards compatibility with v1.1b.

5. Keys

   a. The new signing keys are mandatory in v1.2 because they plug a security hole.

   b. The TPM must support a minimum of 2 key slots.

6. Direct Anonymous Attestation

   a. This is optional in v1.2

   b. Cost is important. The DAA function consumes more TPM resources than any other TPM function, but some platform specific specifications (some servers, for example) may have no need for the anonymity and pseudonymity provided by DAA.

7. Transport sessions

   a. These are mandatory in v1.2.

   b. Transport sessions

      i. Enable protection of data submitted to a TPM and produced by a TPM

      ii. Enable proof of the TPM commands executed during an arbitrary session.

8. Resettable Endorsement Key

   a. This is optional in v1.2

   b. Cost is important. Resettable EKs are valuable in some markets segments, but cause more complexity than non-resettable EKs, which are expected to be the dominant type of EK

9. Monotonic Counter

   a. This is mandatory in v1.2

   b. A monotonic counter is essential to enable software to defeat certain types of attack, by enabling it to determine the version (revision) of dynamic data.

10. Time Ticks

   a. This is mandatory in v1.2

   b. Time stamping is a function that is potentially beneficial to both a user and system software.

11. Delegation (includes DSAP)
    a. This is mandatory in v1.2
    b. Delegation enables the well-established principle of least privilege to be applied to Owner authorized commands.
12. GPIO
    a. This is optional in v1.2
    b. Cost is important. Not all types of platform will require a secure intra-platform method of key distribution
13. Locality
    a. The use of locality is optional in v1.2
    b. The structures that define locality are mandatory
    c. Locality is an essential part of many (new) TPM commands, but the definition of locality varies widely from platform to platform, and may not be required by some types of platforms.
    d. It is mandatory that a platform specific specification indicate the definitions of locality on the platform. It is perfectly reasonable to only define one locality and ignore all other uses of locality on a platform
14. TPM-audit
    a. This is optional in v1.2
    b. Proper TPM-audit requires support to reliably store logs and control access to the TPM, and any mechanism (an OS, for example) that could provide such support is potentially capable of providing an audit log without using TPM-audit. Nevertheless, TPM-audit might be useful to verify operation of any and all software, including an OS. TPM-audit is believed to be of no practical use in a client, but might be valuable in a server, for example.
15. Certified Migration
    a. This is optional in v1.2
    b. Cost is important. Certified Migration enables a business model that may be nonsense for some platforms.

# 40. 1.1a and 1.2 Differences

**Start of informative comment**

All 1.2 TPM commands are completely compliant with 1.1b commands with the following known exceptions.

1. TSC_PhysicalPresence does not support configuration and usage in a single step.

2. TPM_GetPubKey is unable to read the SRK unless TPM_PERMANENT_FLAGS -> readSRKPub is TRUE

3. TPM_SetTempDeactivated now requires either physical presence or TPM Operator authorization to execute

4. TPM_OwnerClear does not modify TPM_PERMANENT_DATA -> authDIR[0].

**End of informative comment**