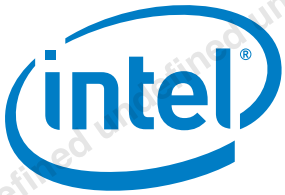


Intel® Trusted Execution Technology (Intel® TXT)

Software Development Guide

Measured Launched Environment Developer's Guide

July 2020

Revision 16.2

Document Number: 315168-16.2

You may not use or facilitate the use of this document in connection with any infringement or other legal analysis concerning Intel products described herein. You agree to grant Intel a non-exclusive, royalty-free license to any patent claim thereafter drafted which includes subject matter disclosed herein.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software, or service activation. Performance varies depending on system configuration. No computer system can be absolutely secure. Check with your system manufacturer or retailer or learn more at intel.com.

Intel technologies may require enabled hardware, specific software, or services activation. Check with your system manufacturer or retailer.

The products described may contain design defects or errors known as errata which may cause the product to deviate from published specifications. Current characterized errata are available on request.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest Intel product specifications and roadmaps

Copies of documents which have an order number and are referenced in this document may be obtained by calling 1-800-548-4725 or visit www.intel.com/design/literature.htm.

Intel, Pentium, Intel Xeon, Intel NetBurst, Intel Core Solo, Intel Core Duo, Intel Pentium D, Itanium, MMX, and VTune are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

*Other names and brands may be claimed as the property of others.

Contact your local Intel sales office or your distributor to obtain the latest specifications and before placing your product order.

Contents

Figures

Tables

§

Revision History

Date	Revision	Description
July 2020	16.2	• LCP_POLICY_LIST2_1 listing will now correctly list KeyAndSignatureOffset field type as UINT16 instead of UINT32
May 2020	16.1	• Document update to correct several errors: • Clarify check for SMX and VMX • Delete description of deprecated register (at offset 0100h) • Correctly name register at offset 0200h • Reflect above changes in listing 3
September 2019	16.0	• Re-write of LCP to reflect architectural changes. • Multiple corrections and updates
November 2017	15.0	• Fixed structure HEAP_EVENT_LOG_POINTER_ELEMENT2(must be HEAP_EVENT_LOG_POINTER_ELEMENT2_1) • Added reserved capability bit 11 • Added CR3 reference to SINIT exit state table • Corrected errors in the last revisions
May 2017	14.0	• Added changes related to Converged BtG / TXT technologies and CNSS Advisory 02-15
August 2016	13.0	• Added each MTRR base must be multiple of that MTTR size Prior to GETSEC[SENTER] execution.
July 2015	12.0	• Added SGX requirement for TXT platform • Updated LCP changes of TPM2.0 transitions • Added TCG compliant TXT event log formats • Documented TPM NV definitions • Inclusion of detailed LCP checklists
May 2014	11.0	• Update TPM_PCR_INFO_SHORT structure and TPMS_QUOTE_INFO structure Endianness
March 2014	10.0	• Corrections to data structures, algorithm detail versus prior versions • Inclusion of TPM 2.0 changes and additions
April 2013	9.0	• Numerous updates from prior author • Added text for LCP details/authorities • Corrected osinitdata offset 84 for versions 6+
June 2011	8.0	• Format updates
March 2011	7.0	• Miscellaneous errata

		• Documented ProcessorIDList support • Described CPU Hotplug handling • Updated TXT configuration registers • Documented new TXT Heap structures • Added LCP_SBIOS_ELEMENT • Documented processor and system state after SENTER/RLP wakeup
December 2009	6.0	• Miscellaneous errata • Added definition of LCP v2 • Multiple processor support
June 2008	5.0	• Updated for latest structure versions and new RLP wakeup mechanism • Added Launch Control Policy information • Removed TEP Appendix • Many miscellaneous changes and additions
August 2007	4.0	• Updated for production platforms • Use MLE terminology
October 2006	3.0	• Added launched environment consideration • Renamed LT to Intel® TXT
August 2006	2.0	• Established public document number • Edited throughout for clarity.
May 2006	1.0	• Initial release.

§

Foreword

Current revision of this document reflects MLE visible changes resulting from converging Boot Guard and Trusted Execution Technologies (Converged BtG/TXT a.k.a. CBnT), and "*CNSS Advisory Memorandum 02-15"*, applicable to platform 2017 and beyond. A short summary of changes can be found in Section 4.6

All obsolete features (i.e., those no longer supported) due to convergence have been removed from this document, starting from revision 014. If you need to review the removed information, it is recommended to consult revision 013, available at: https://cdrdv2.intel.com/v1/dl/getContent/616057.

1.0 Overview

Intel's technology for safer computing, Intel® Trusted Execution Technology (Intel® TXT), defines platform-level enhancements that provide the building blocks for creating trusted platforms.

Whenever the word trust is used, there must be a definition of who is doing the trusting and what is being trusted. This enhanced platform helps to provide the authenticity of the controlling environment such that those wishing to rely on the platform can make an appropriate trust decision. The enhanced platform determines the identity of the controlling environment by accurately measuring the controlling software (see Section 1.1).

Another aspect of the trust decision is the ability of the platform to resist attempts to change the controlling environment. The enhanced platform will resist attempts by software processes to change the controlling environment or bypass the bounds set by the controlling environment.

What is the controlling environment for this enhanced platform? The platform is a set of extensions designed to provide a measured and controlled launch of system software that will then establish a protected environment for itself and any additional software that it may execute.

These extensions enhance two areas:

- The launching of the Measured Launched Environment (MLE)
- The protection of the MLE from potential corruption

The enhanced platform provides these launch-and-control interfaces using Safer Mode Extensions (SMX).

The SMX interface includes the following functions:

- Measured launch of the MLE
- Mechanisms to ensure the above measurement is protected and stored in a secure location
- Protection mechanisms that allow the MLE to control attempts to modify itself

1.1 Measurement and Intel® Trusted Execution Technology (Intel® TXT)

Intel® TXT uses the term *measurement* frequently. Measuring software involves processing the executable in such way that the result is *unique* and *indicative* to any changes in the executable. A cryptographic hash algorithm meets these needs.

A cryptographic hash algorithm is sensitive to even one-bit changes to the measured entity. It also produces outputs that are sufficiently large so the potential for collisions (where two hash values are the same) is extremely small. When the term measurement is used in this specification, the meaning is that the measuring process takes a cryptographic hash of the measured entity.

The controlling environment is provided by system software such as an OS kernel or Virtual Machine Manager (VMM). The software launched using the SMX instructions is known as the Measured Launched Environment (MLE). MLEs provide different launch mechanisms and increased protection (offering protection from possible software corruption).

1.2 Dynamic Root of Trust

A central objective of the Intel® TXT platform is to provide a measurement of the launched execution environment.

One measurement is made when the platform boots, using techniques defined by the Trusted Computing Group (TCG). The TCG defines a Root of Trust for Measurement (RTM) that executes on each platform reset; it creates a chain of trust from reset to the measured environment. As the measurement always executes at platform reset, the TCG defines this type of RTM as a Static RTM (SRTM).

Maintaining a chain of trust for a length of time may be challenging for an MLE meant for use in Intel® TXT; this is because an MLE may operate in an environment that is constantly exposed to unknown software entities. To address this issue, the enhanced platform provides another RTM with Intel® TXT instructions. The TCG terminology for this option is Dynamic Root of Trust for Measurement (DRTM). The advantage of a DRTM (also called the 'late launch' option) is that the launch of the measured environment can occur at any time without resorting to a platform reset. It is possible to launch an MLE, execute for a time, terminate the MLE, execute without virtualization, and then launch the MLE again. One possible sequence is:

1. During the BIOS load: (a) launch an MLE for use by the BIOS, (b) terminate the MLE when its work is done, (c) continue with BIOS processing and hand off to an OS.
2. Then, the OS loads and launches a different MLE.
3. In both instances, the platform measures each MLE and ensures the proper storage of the MLE measurement value.

1.2.1 Launch Sequence

When launching an MLE, the environment must load two code modules into memory. One module is the MLE. The other is known as an authenticated code (AC) module. The AC module (also referred to as ACM) is only in use during the measurement and verification process and is chipset specific. The chipset vendor digitally signs it; the launch process must successfully validate the digital signature before continuing.

With the AC module and MLE in memory, the launching environment can invoke the GETSEC[SENTER] instruction provided by SMX.

GETSEC[SENTER] broadcasts messages to the chipset and other physical or logical processors in the platform. In response, other logical processors perform basic cleanup, signal readiness to proceed, and wait for messages to join the environment created by the MLE. As this sequence requires synchronization, there is an initiating logical processor (ILP) and responding logical processor(s) (RLP(s)). The ILP must be the system bootstrap processor (BSP), which is the processor with IA32_APIC_BASE MSR.BSP = 1. RLPs are also often referred to as application processors (APs).

After all logical processors signal their readiness to join and are in the wait state, the initiating logical processor loads, authenticates, and executes the AC module. The AC module tests for various chipset and processor configurations and ensures the platform has an acceptable configuration. It then measures and launches the MLE.

The MLE initialization routine completes system configuration changes (including redirecting INITs, SMIs, interrupts, etc.); it wakes up the responding logical processors (RLPs) and brings them into the measured environment. At this point, all logical processors and the chipset are correctly configured.

At some later point, it is possible for the MLE to exit and then be launched again, without issuing a system reset.

1.3 Storing the Measurement

SMX operation during the launch provides an accurate measurement of the MLE. After creating the measurement, the initiating logical processor stores that measurement in the trusted platform module (TPM), defined by the TCG. An enhanced platform includes mechanisms that ensure that the measurement of the MLE (completed during the launch process) is properly reported to the TPM.

With the MLE measurement in the TPM, the MLE can use the measurement value to protect sensitive information and detect potential unauthorized changes to the MLE itself.

1.4 Controlled Take-down

Because the MLE controls the platform, exiting the MLE is a controlled process. The process includes: (a) shutting down any guest virtual machines (VMs) if they were created; (b) ensuring that memory previously used does not leak sensitive information.

The MLE cleans up after itself and terminates the MLE control of the environment. If a VMM was running, the MLE may choose to turn control of the platform over to the software that was running in one of the VMs.

1.5 SMX and VMX Interaction

A VM abort may occur while in SMX operation. This behavior is described in the *Intel 64 and IA-32 Software Developer Manual, Volume 3B*. Note that entering authenticated code execution mode or launching of a measured environment affects the behavior and response of the logical processors to certain external pin events.

1.6 Authenticated Code Module

To support the establishment of a measured environment, SMX enables the capability of an authenticated code execution mode. This provides the ability for a special code module, referred to as an authenticated code module (ACM, also frequently referred to as SINIT), to be loaded into internal RAM (referred to as authenticated code execution area or ACEA) within the processor. The AC module is first authenticated and then executed using a tamper resistant mechanism.

Authentication is achieved by a digital signature in the header of the AC module. The processor calculates a hash of the AC module and uses the result to validate the signature. Using SMX, a processor will only initialize processor state or execute the AC module if it passes authentication. Since the authenticated code module is held within the internal RAM of the processor, execution of the module can occur in isolation with respect to the contents of external memory or activities on the external processor bus.

Beside of SINIT, BIOS contains another ACM – the BIOS AC module used for performing subordinate tasks such as TXT Opt-in preparation, clearing the memory, alias checking, etc.

1.7 Chipset Support

One important feature the chipset provides is direct memory access (DMA) protection via Intel® Virtualization Technology (Intel® VT) for Directed I/O (Intel® VT-d). Intel® VT-d, under control of the MLE, allows the MLE to protect itself and any other software such as guest VMs from unauthorized device access to memory. Intel® VT-d blocks access to specific physical memory pages and the enforcement of the block occurs for all DMA access to the protected pages. See Section 1.11 for more information on DMA protection mechanisms.

The Intel® TXT architecture also provides extensions that access certain chipset registers and TPM address space.

Chipset registers that interact with SMX are accessed from two regions of memory by system software using memory read/write protocols. These two memory regions, Intel® TXT public space and Intel® TXT private space, are mappings to the same set of chipset registers but with different read/write permissions depending on which space the memory access came through. The Intel® TXT private space is not accessible to system software until it is unlocked by SMX instructions.

The sets of interface registers accessible within a TPM device are grouped by a locality attribute and are a separate set of address ranges from the Intel® TXT public and private spaces. The following localities are defined:

- Locality 0 : Non-trusted and legacy TPM operation
- Locality 1 : An environment for use by the Trusted Operating System
- Locality 2 : MLE access
- Locality 3 : Authenticated Code Module
- Locality 4 : Intel® TXT hardware use only

Like Intel® TXT public and private space, some of these localities are only accessible via SMX instructions and others are not accessible by software until unlocked by SMX instructions.

1.8 TPM Usage

Intel® TXT makes extensive use of the trusted platform module (TPM) defined by the Trusted Computing Group (TCG) in the "*TPM Main specification for TPM family 1.2"* and "*TPM Library specification for TPM family 2.0"*. The TPM provides a repository for measurements and the mechanisms to make use of the measurements. The system makes use of the measurements to both report the current platform configuration and to provide long-term protection of sensitive information.

The TPM stores measurements in platform configuration registers (PCRs). Each PCR provides a storage area that allows an unlimited number of measurements in a fixed amount of space. They provide this feature by an inherent property of cryptographic hashes. Outside entities never write directly to a PCR register, they "extend" PCR contents. The extend operation takes the current value of the PCR, appends the new value, performs a cryptographic hash on the combined value, and the hash result is the new PCR value. One of the properties of cryptographic hashes is that they are order dependent. This means hashing A then B produces a different result from hashing B then A. This ordering property allows the PCR contents to indicate the order of measurements.

Sending measurement values from the measuring agent to the TPM is a critical platform task. The dynamic root of trust for measurement (DRTM) requires specific messages to flow from the DRTM to the TPM. The Intel® TXT DRTM is the GETSEC[SENTER] instruction and the system ensures GETSEC[SENTER] has special messages to communicate to the TPM. These special messages take advantage of TPM localities 3 and 4 to protect the messages and inform the TPM that GETSEC[SENTER] is sending the messages.

Besides of the measurements, Intel® TXT uses TPM non-volatile (NV) RAM to store launch control policy (LCP) data and for inter-ACM communication. Three main indices used by Intel® TXT are denoted as AUX (auxiliary), PO (platform owner) and SGX

(software guard extension). Detailed information about these indices will be presented in subsequent Sections 3 and 4.5.

With the release of the TPM 2.0 specification and supporting devices, many changes may be required for TXT launch. TPM 2.0 devices can support a variety of cryptographic algorithms, and a single device will often support multiple digest and asymmetric signature algorithms. For the purposes of this document, TPM 1.2 and 2.0 devices will be referred to as two distinct families. The MLE and ACM determine the family of platform TPM. In subsequent discussion, we will refer to actions and structures related to TPM 1.2 or 2.0 as TPM1.2 mode and TPM2.0 mode of operation, respectively.

1.9 Hash Algorithm Support

TPM 2.0 provides PCRs in banks—that is, one bank of PCRs for each supported hash algorithm. For example, a TPM that supports three hashing algorithms will have three banks of PCRs and thus “measuring an object into PCR” implies hashing of that object using each of the three hashing algorithms and extending obtained digests into all banks. The TPM 2.0 specification enumerates all the hash algorithms it allows; of those, the current TXT components support SHA1, SHA256, SHA384, and SM3_256.

TPM 2.0 devices support algorithm agile “event” commands. These commands extend measurements into all existing PCR banks at once. Measuring objects of significant size using event commands may incur performance penalties.

Alternatively, embedded software can be used to compute digests, and the results then extended into PCRs using non-agile “extend” commands. While this may be more efficient, software support for all hashing algorithms supported by a given TPM might be not present. Should this occur, PCRs in banks utilizing algorithms unsupported by the embedded software will be capped with the value “1”

Whether extend calculations will be done using TPM hardware event commands or software implementations is an MLE decision and will be communicated to the launched ACM via the “flags” field, which is described in Table 22.

TPM 1.2 devices support only SHA1 hashing algorithm. To simplify discussion, for both device families, hashing algorithms will be denoted as HASH. For TPM1.2 this means SHA1. For TPM2.0, this means all algorithms supported by the device.

In TPM1.2 mode certain values are extended into PCRs without hashing. Some of them are extended this way due to historical dependencies; others are using extends of zero digests or constant values as an indication of various platform states or events. In such cases, the zero digest is an array of 20 zeros: {0,0, … ,0}

These extends will continue to be supported in TPM1.2 mode without changes, hence ensuring backwards compatibility.

In TPM2.0 this practice is discouraged and simply cannot be supported when maximum agility (MA) extend policy is enforced – see Table 22. Therefore, in all above cases we will not extend constant values as they appear, but we will measure them instead – that is, we will hash these constant values and extend resultant digests into PCRs. All such cases are flagged in the explanation of details/authority measurements below.

1.10 PCR Usage

As part of the measured launch, Intel® TXT will extend measurements of the elements and configuration values of the dynamic root of trust into certain TPM PCRs. The values comprising these measurements (indicated below) are provided in the heap *SinitMleData* data table described in Appendix C.5, and in the event log.

While the MLE may choose to extend additional values into these PCRs, the values described below are those present immediately after the MLE receives control following the GETSEC[SENTER] instruction.

In addition to values explicitly measured by SINIT and MLE GETSEC[SENTER] instruction measures the SINIT module itself by the means of _HASH_START/_HASH_DATA/_HASH_END HW event sequence (will be denoted as _HASH_*). This event sequence is supported by both TPM families and is described in the following TCG specifications: *"TCG TPM Main Specification Version 1.2"*, "TCG Trusted Platform Module Library, Family 2.0", and *"TCG PC Client Platform TPM Profile Specification for TPM 2.0"*.

Since these PCR values are arrived at by a series of measurements, determining their derivation requires a trace or log of the extending steps executed. These steps are recorded in the TPM Event Log, described in Appendix F.

1.10.1 Legacy Usage

Legacy—or original—PCR usage separates the values in the PCRs according to platform elements and MLE. The platform elements of the trusted computing base (TCB), such as SINIT and launch control policy (LCP), are measured into PCR 17 and the MLE is measured into PCR 18.

Legacy (LG) usage support is reported by value of 0 in bit 4 of the *Capabilities* field of both SINIT and MLE headers (see Table 4). Because this reporting needs to be compatible with earlier versions of the *Capabilities* field, for which bit 4 was reserved, inverse logic is used to represent LG usage.

Legacy PCR usage is no longer supported by Intel ® TXT

1.10.2 Details and Authorities Usage

Details and Authorities (DA) PCR usage separates the values in the PCRs according to whether the value extended is the actual measurement of a given entity (a detail) or represents the authority for the given entity (an authority). Details are extended to PCR 17 and authorities to PCR 18. Evaluators who do not care about rollback can use the authorities PCR (18) and it should remain the same even when elements of the TCB are changed.

This usage corresponds to a value of 1 in bit 5 of the *Capabilities* field (see Table 4).

1.10.2.1 PCR 17 (Details)

"Details" measurements include hashes of all components participating in establishing the trusted execution environment and due to very nature of hash algorithm change of any component entail change of the final PCR17 value.

PCR 17 is initialized using the _TPM_HASH_* sequence. As part of this sequence, PCRs 17-23 are reset to 0 before _TPM_HASH_START.

The _HASH_DATA provided in this sequence is the concatenation of digest of the SINIT ACM that was used in the launch process and the 4-byte value of the SENTER parameters (in the EDX register and in *SinitMleData.EdxSenterFlags*). The digest of SINIT is also optionally stored in the *SinitMleData.SinitHash* field of the SINIT to MLE data table (*SinitMleData* – see Appendix C.5).

Result of _HASH_* differs from one CPU family to another and between TPM families. Originally, Intel ® CPUs were computing SINIT digest using SHA1 producing 20-byte output. Newer Intel ® CPUs are computing SINIT digest using SHA256 producing 32 bytes.

Obtained data stream (SINIT digest | EDX - 24 or 36 bytes respectively) is sent to TPM via HASH_DATA. Based on TPM family PCR17 gets updated similarly to how TPM(2)_Extend () command does it.

Respective PCR17 value is reported based on what kind of TPM family device is present in the system.

If TPM 1.2 family device is found in the system, PCR17 value is reported via *SinitMleData* table in which *SinitMleData.SinitHash* field contains value of PCR 17 after the initial extend operation. Alternatively, if TPM 2.0 family device is present, PCR 17 value is reported via the Event Log (see Appendix F.3 for more details).

After the initial SINIT measurement, PCR 17 is extended with the digest values of the items in Table 1:

Table 1. Digest values extending PCR17

Item	Remarks
Digest of MLE	
S-ACM registration data retrieved from the AUX Index	20 bytes in TPM 1.2 mode and 32 bytes in TPM 2.0 mode. In TPM 1.2 mode data extended to TPM as is, in TPM 2.0 mode extended value is $HASH_{HashAlgId}$ (data) using *HashAlgId* of the bank being extended.
Digest of *PO.PolicyControl* field of the used policy.	Retrieved from the TPM NV PO index. Coded as DWORD.
Digest of all matching elements used by the policy[1]	TPM 1.2 only event
Digest of STM	If STM is not enabled, for TPM 1.2 family ZeroDigest is extended, while for the TPM 2.0 family, the digest = $HASH_{HashAlgId}$ (0x0) is extended.
Digest of the *Capability* field of the *OsSinitData* table	TPM 2.0 only event. coded as DWORD.
Digest of the effective LCP details (i.e., all matching elements used by the policy)	TPM 2.0 only event. Construction of byte stream containing digests of all elements contributed to the established policy is discussed in Section 3.4.3.3
Digest of the TPM NV index public info structures	TPM 2.0 only event. Rules how to construct byte stream for hashing are discussed in Section 3.4.4.
Digest of Early TXT BIOS	
Digest of the Key Manifest signature	
Digest of the Boot Policy Manifest signature	
Digest of the ACM boot policy status	Digest of the ACM_POLICY_STATUS register
Pseudo random value	NPW SINIT only event. Renders PCR17 unusable in the production environment.
Unsupported bank capping value	OneDigest is extended into PCR bank using an algorithm that is not supported by SINIT SW

Note: The table above does not reflect the order of extends. Attester might retrieve the exact order from the Event Log.

[1] Construction of byte stream containing digests of all elements contributed to established policy is discussed in Section 3.4. If there is no policy used, a ZeroDigest is extended.

1.10.2.2 PCR 18 (Authorities)

"Authority" measurements include hashes of some unique identifying properties of signing authorities such as public signature verification keys. This enables the same authority to issue an update of component without affecting the final PCR18 value, because the signing authority is unchanged.

The table below shows digests extended to PCR18.

Table 2. Digest values extending PCR18

Item	Remarks
Digest of Processor S-CRTM status	Coded as DWORD; same value as extended to PCR17 in TPM 2.0 mode.
Digest of the PO.PolicyControl field of the platform owner (PO) policy	Coded as DWORD; same value as extended to PCR17
Digest of the *Capability* field of the *OsSinitData* table	Coded as DWORD; same value as extended to PCR17 in TPM 2.0 mode
Digest of SINIT signer key	The value hashed is modulus of the key
Digest of LCP authorities	TPM 1.2 only event. Measurement of LCP lists containing elements contributed to established policy. Byte stream, which is hashed, is built using rules in Section 3.4.2. If there is no policy, ZeroDigest is extended.
Digest of LCP authorities	TPM 2.0 only event. Digest of LCP authorities (i.e., measurement of LCP lists containing elements contributed to established policy). Byte stream which is hashed is built using rules in Section 3.4.3.4.
Digest of the TPM NV index public info structures	TPM 2.0 only event. same value as extended to PCR17 in TPM 2.0 mode. Rules how to construct byte stream for hashing are discussed in Section 3.4.4.
Digest of the Key Manifest descriptor	
Digest of the Boot Policy Manifest descriptor	
Pseudo random value	NPW SINIT only event. Renders PCR18 unusable in the production environment.
Unsupported bank capping value	OneDigest is extended into PCR bank using an algorithm that is not supported by SINIT SW.

Note: As with PCR17, the table above does not reflect the order of extends. Attester might retrieve the exact order from the Event Log.

1.11 DMA Protection

This section briefly describes two chipset mechanisms that can be used to protect regions of memory from DMA access by bus primary devices. More details on these mechanisms can be found in the latest External Design Specification (EDS) of the targeted chipset family and "*Intel® Virtualization Technology for Directed I/O Architecture Specification*".

1.11.1 DMA Protected Range (DPR)

The DMA Protected Range (DPR) is a region of contiguous physical memory whose last byte is the byte before the start of SMRAM segment (TSEG), and which is protected from all DMA access. The DPR size is set and locked by BIOS. This protection is applied to the final physical address after any other translation (e.g,. Intel® VT-d, graphics address remapping table (GART), etc.).

The DPR covers the Intel® TXT heap and SINIT AC Module reserved memory (as specified in the TXT.SINIT.BASE/TXT.SINIT.SIZE registers). On current systems it is no less than 3MB in size, and though this may change in the future it will always be large enough to cover the heap and SINIT regions.

The MLE itself may reside in the DPR if it does not conflict with either SINIT or heap areas. If it does reside in the DPR, then the Intel® VT-d protected memory regions (PMR) do not need to be used to cover it.

1.11.2 Protected Memory Regions (PMRs)

The Intel® VT-d protected memory regions (PMRs) are two ranges of physical addresses that are protected from DMA access. One region must be in the lower 4GB of memory and the other may be anywhere in address space. Either or both may be unused.

The use of the PMRs is not mutually exclusive of DMA remapping. If the MLE enables DMA remapping, it should place the Intel® VT-d page tables within the PMR region(s) to protect them from DMA activity prior to turning on remapping. While it is not required that PMRs be disabled once DMA remapping is enabled, if the MLE wants to manage all DMA protection through remapping tables then it must explicitly disable the PMR(s).

The MLE may reside within one of the PMR regions. If the MLE is not within the DPR region then it must be within one of the PMR regions, else SINIT will not permit the environment to be launched.

For more details of the PMRs, see the "*Intel® Virtualization Technology for Directed I/O Architecture Specification*".

1.12 Intel® TXT Shutdown

1.12.1 Reset Conditions

When an Intel® TXT shutdown condition occurs, the processor or software writes an error code indicating the reason for the failure to the TXT.ERRORCODE register. It then writes to the TXT.CMD.RESET command register, initiating a platform reset. After the write to TXT.CMD.RESET, the processor enters a shutdown sleep state with all external pin events, bus or error events, machine check signaling, and MONITOR/MWAIT event signaling masked. Only the assertion of reset back to the processor takes it out of this sleep state. The Intel® TXT error code register is not cleared by the platform reset; this makes the error code accessible for post-reset diagnostics.

The processor can generate an Intel® TXT shutdown during execution of certain GETSEC leaf functions (for example: ENTERACCS, EXITAC, SENTER, SEXIT), where recovery from an error condition is not considered reliable. This situation should be interpreted as an abort of authenticated execution or measured environment launch.

A legacy IA-32 triple-fault shutdown condition is also converted to an Intel® TXT shutdown sequence if the triple-fault shutdown occurs during authenticated code execution mode or while the measured environment is active. The same is true for other legacy non-SMX specific fault shutdown error conditions. Legacy shutdown to Intel® TXT shutdown conversions are defined as the mode of operation between:

- Execution of the GETSEC functions ENTERACCS issued by software and EXITAC issued by the ACM at completion
- Recognition of the message signaling the beginning of the processor rendezvous after GETSEC[SENTER] and the message signaling the completion of the processor rendezvous

Additionally, there is a special case. If the processor is in VMX operation while the measured environment is active, a triple-fault shutdown condition that causes a guest exiting event back to the VMM supersedes conversion to the Intel® TXT shutdown sequence. In this situation, the VMM remains in control after the error condition that occurred at the guest level and there is no need to abort processor execution.

Given the above situation, if the triple-fault shutdown occurs at the root level of the MLE or a virtual machine extensions (VMX) abort is detected, then an Intel® TXT shutdown sequence is signaled. For more details on a VMX abort, see Chapter 23, “VM Exits,” in the *Intel 64 and IA-32 Software Developer Manuals, Volume 3B*.

§

2.0 Measured Launched Environment

Intel® TXT can be used to launch any type of code. However, this section describes the launch, operation, and teardown of a VMM using Intel® TXT; any other code would have a similar sequence.

2.1 MLE Architecture Overview

Any Measured Launched Environment (MLE) will generally consist of three main sections of code: the initialization, the dispatch routine, and the shutdown. The initialization code is run each time the Intel® TXT environment is launched. This code includes code to setup the MLE on the ILP and join code to initialize the RLPs.

After initialization, the MLE behaves like the unmeasured version would have; in the case of a VMM, this is trapping various guest operations and virtualizing certain processor states.

Finally, the MLE prepares for shutdown by again synchronizing the processors, clearing any state, and executing the GETSEC[SEXIT] instruction.

Table 3 shows the format of the MLE Header structure stored within the MLE image. The SINIT AC module uses the MLE Header structure to set up the correct initial MLE state and to find the MLE entry point. The header is part of the MLE hash.

Table 3. MLE Header structure

Field	Offset	Size (bytes)	Description
UUID (universally unique identifier)	0	16	Identifies this structure
HeaderLen	16	4	Length of header in bytes
Version	20	4	Version number of this structure
EntryPoint	24	4	Linear entry point of MLE
FirstValidPage	28	4	Starting linear address of (first valid page of) MLE
MleStart	32	4	Offset within MLE binary file of first byte of MLE, as specified in page table
MleEnd	36	4	Offset within MLE binary file of last byte + 1 of MLE, as specified in page table
Capabilities	40	4	Bit vector of MLE-supported capabilities
CmdlineStart	44	4	Starting linear address of command line
CmdlineEnd	48	4	Ending linear address of command line

***UUID*:** This field contains a UUID which uniquely identifies this MLE Header Structure. The UUID is defined as follows:

```
ULONG   UUID0;      // 9082AC5A
ULONG   UUID1;      // 74A7476F
ULONG   UUID2;      // A2555C0F
ULONG   UUID3;      // 42B651CB
```

This UUID value should only exist in the MLE (binary) in this field of the MLE header. This implies that this UUID should not be stored as a variable nor placed in the code to be assigned to this field. This can also be ensured by analyzing the binary.

***HeaderLen*:** this field contains the length in bytes of the MLE Header Structure.

Version: this field contains the version of the MLE header, where the upper two bytes are the major version and the lower two bytes are the minor version. Changes in the major version indicate that an incompatible change in behavior is required of the MLE or that the format of this structure is not backwards compatible. Version 2.2 (20002H) is the currently supported version.

***EntryPoint*:** this field is the linear address, within the MLE's linear address space, at which the ILP will begin execution upon successful completion of the GETSEC[SENTER] instruction.

***FirstValidPage*:** this field is the starting linear address of the MLE. This will be verified by SINIT to match the first valid entry in the MLE page tables.

***MleStart / MleEnd*:** these fields are intended for use by software that needs to know which portion of an MLE binary file is the MLE, as defined by its page table. This might be useful for calculating the MLE hash when the entire binary file is not being used as the MLE.

***Capabilities*:** this bit vector represents TXT-related capabilities that the MLE supports. It will be used by the SINIT AC module to determine whether the MLE is compatible with it and as needed for any optional capabilities. The currently defined bits for this are:

Table 4. MLE/SINIT Capabilities Field Bit Definitions

Bit position	Description
0	Support for GETSEC[WAKEUP] for RLP wakeup All MLEs should support this. 1 = supported/requested 0 = not supported

Bit position	Description
1	Support for RLP wakeup using MONITOR address (*SinitMleData.RlpWakeupAddr*) All MLEs should support this. 1 = supported/requested 0 = not supported
2	The ECX register will contain the pointer to the MLE page table on return from SINIT to the MLE EntryPoint 1 = supported/requested 0 = not supported
3	STM support 1 = supported/requested 0 = not supported
4	TPM 1.2 family: Legacy PCR usage support (negative logic is used for backwards compatibility) 0 = supported/requested 1 = not supported No longer supported: reserved/ignored
5	TPM 1.2 family: Details/authorities PCR usage support This usage takes precedence over legacy usage if both are requested 1 = supported/requested 0 = not supported No longer supported: reserved/ignored
7-6	Platform Type 00: legacy / platform undefined 01: client platform ACM 10: server platform ACM 11: reserved / illegal
8	MAXPHYADDR supported 0: 36 bits MTRR masks computed, regardless of actual width 1: actual width MTRR masks computed as reported by CPUID function 0x80000008 Always forced to “1” by an SINIT. See note
9	Supported format of TPM 2.0 event log: = 0 – Original TXT TPM 2.0 Event Log = 1 – *“TCG PC Client Platform. EFI Protocol Specification”* compatible Event Log Always forced to “1”
10	Converged BtG / TXT support (CBnT): = 0 – CBnT is not supported = 1 – CBnT is supported
13:11	Startup ACM specific use. Reserved for SINIT and MLE
31:14	Reserved (must be 0)

Note: Legacy TBOOT computes MTRR masks assuming 36 bits width of an address bus. This may lead to creation of potentially disjoint WB cache ranges and violation of CRAM

protections. To remedy case and support legacy MLE/SINIT behavior the following has been added:

BIT8 of *Capabilities* field in ACM info table and MLE header was defined to indicate use of bus width method. Both MLE and SINIT will examine this bit in counterpart module and amend execution as follows in Truth Table.

Table 5. Truth Table of SINIT / MLE functionality

MLE	SINIT	Functionality
Legacy BIT8 = 0	Legacy BIT8 = 0	Both use 36 bits
New BIT8 = 1	Legacy BIT8 = 0	MLE sees BIT==0 and prepares 36-bit MTRR masks. Legacy SINIT ignores BIT8 in MLE header
Legacy BIT8 = 0	New BIT8 = 1	Legacy MLE ignores BIT8 in SINIT ACM Info Table and prepares 36-bit MTRR masks. SINIT checks MLE header and validates masks as 36 bits
New BIT8 = 1	New BIT8 = 1	Both use actual bus width.

***CmdlineStart / CmdlineEnd*:** these fields are intended for use by software that needs to calculate the MLE hash, for MLEs that include their command lines in their identity. These are linear addresses within the MLE of the beginning and end of a buffer that will contain the command line. The buffer is padded with bytes of 0x0 at the end. MLEs that do not include the command line in their identity should set these fields to 0.

2.2 MLE Launch

At some point system software will start an Intel® TXT measured environment. This may be done at operating system loader time or could be done after the operating system boots. From this point on we will assume that the operating system is starting the Intel® TXT measured environment and refer to this code as the system software.

After the measured environment startup, the application processors (RLPs) will not respond to system inter-processor interrupts (SIPIs) as they did before SENTER. Once the measured environment is launched, the RLPs cannot run the real-mode MP startup code and their startup must be initiated by an alternate method. The new MP startup algorithm does not allow the RLPs to leave protected mode with paging on. The OS may also be required to detect whether a measured environment has been established and use this information to decide which MP startup algorithm is appropriate (the standard MP startup algorithm or the modified algorithm).

This section shows the pseudocode for preparing the system for the SMX measured launch. The following describes the process in several sub-sections:

- Intel® TXT detection and processor preparation
- Detection of previous errors
- Loading the SINIT AC module

- Loading the MLE and processor rendezvous
- Performing a measured launch

2.2.1 Intel® TXT Detection and Processor Preparation

Lines 1 - 4: Before attempting to launch the measured environment, the system software should check that all logical processors support VMX and SMX (the check for VMX support is not necessary if the environment to be launched will not use VMX).

For single processor socket systems, it is sufficient if this action is only performed by the ILP. This includes physical processors containing multiple logical processors. In order to correctly handle multiple processor socket systems, this check must be performed on all logical processors. It is possible that two physical processor within the same system may differ in terms of SMX and VMX capabilities.

For details on detecting and enabling VMX see chapter 19, “Introduction to Virtual-Machine Extensions”, in the *Intel 64 and IA-32 Software Developer Manuals, Volume 3B*. For details on detecting and enabling SMX support see chapter 6, “Safer Mode Extensions Reference”, in the *Intel 64 and IA-32 Software Developer Manuals, Volume 2B*.

Lines 5 - 9: System software should check that the chipset supports Intel® TXT prior to launching the measured environment. The presence of the Intel® TXT chipset can be detected by executing GETSEC[CAPABILITIES] with EAX=0 & EBX=0. This instruction will return the ‘Intel® TXT Chipset’ bit set in EAX if an Intel® TXT chipset is present. The processor must enable SMX before executing the GETSEC instruction.

Lines 10 – 12: System software should also verify that the processor supports all GETSEC instruction leaf indices that will be needed. The minimal set of instructions required will depend on the system software and MLE, but is most likely SENTER, SEXIT, WAKEUP, SMCTRL, CAPABILITIES and PARAMETERS. The supported leaves are indicated in the EAX register after executing the GETSEC[CAPABILITIES] instruction as indicated above.

Listing 1. Intel® TXT Detection Pseudocode

```
//
// Intel TXT detection
// Execute on all logical processors for compatibility with
// multiple processor systems
//
1. CPUID(EAX=1);
2. IF (SMX not supported) OR (VMX not supported) {
3.     Fail measured environment startup;
4. }

//
// Enable SMX on ILP & check for Intel TXT chipset
//
```

```
5. CR4.SMXE = 1;
6. GETSEC[CAPABILITIES];
7. IF (Intel TXT chipset NOT present) {
8.     Fail measured environment startup;
9. }
10.  IF (All needed SMX GETSEC leaves are NOT supported) {
11.      Fail measured environment startup;
12.  }
```

2.2.2 Detection of Previous Errors

In order to prevent a cycle of failures or system resets, it is necessary for the system software to check for errors from a previous launch. Errors that are detected by system software prior to executing the GETSEC[SENTER] instruction will be specific to that software and, if persistent, will be in a manner specific to the software. Errors generated during execution of the GETSEC[SENTER] instruction result in a system reset and the error code being stored in the TXT.ERRORCODE register. Possible remediation steps are described in Section 4.2.

Lines 1 - 3: The error code from an error generated during the GETSEC[SENTER] instruction is stored in the TXT.ERRORCODE register, which is persistent across soft resets. Non-zero values other than 0xC000001 indicate an error. Error codes are specific to an SINIT AC module and can be found in a text file that is distributed with the module. Errors that are not AC-module specific are listed in Appendix H

Lines 9 - 12: If the TXT_RESET.STS bit of the TXT.ESTS register is set, then in order to maintain TXT integrity the GETSEC[SENTER] instruction will fail. System software should detect this condition as early as possible and terminate the attempted measured launch and report the error. The cause of the error must be corrected if possible, and the system should be power cycled to clear this bit and permit a launch.

Listing 2. Error Detection Pseudocode

```
//
// Detect previous GETSEC[SENTER] failures
//
1. IF (TXT.ERRORCODE != 0 && TXT.ERRORCODE != SUCCESS) {
2.     Terminate measured launch ;
3.     Report error ;
4.     If remedial action known {
5.        Take remedial action ;
6.        Power-cycle system ;
7.     }
8. }

//
// Detect previous TXT Reset
//
9.  IF (TXT.ESTS[TXT_RESET.STS] != 0) {
10.     Report error ;
```

```
11.     Terminate measured launch ;
12.  }
```

2.2.3 Loading the SINIT AC Module

This action is only performed by the ILP.

BIOS may already have the correct SINIT AC module loaded into memory or system software may need to load the SINIT code into memory from disk. The system software may determine if an SINIT AC module is already loaded by examining the preferred SINIT load location (see below) for a valid SINIT AC module header.

System software should always use the most recent version of the SINIT AC module available to it. It can determine this by comparing the Date fields in the AC module headers.

System software should also match a prospective SINIT AC module to the chipset before loading and attempting to launch the module. This is described in the next two sections of this document.

System software owns the policy for deciding which SINIT module to load. In many cases, it must load the previously loaded SINIT AC module in order to unseal data sealed to a previously launched environment. If an SINIT AC module is to be changed (e.g., upgraded to the latest version), any secrets sealed to the current measured launch may require migration prior to launching via an updated SINIT AC module. It should be noted that server platforms typically carry an appropriate SINIT AC module within their BIOS, and that a BIOS update may result in an SINIT AC module update outside of system software control. For further discussion on this issue, see Section 4.3.1.

BIOS reserves a region of physically contiguous memory for the SINIT AC module, which it specifies through the TXT.SINIT.BASE and TXT.SINIT.SIZE Intel® TXT configuration registers. By convention, at least 192 KB of physically contiguous memory is allocated for the purpose of loading the SINIT AC module; this has increased to 320 KB for latest generation processors. System software must use this region for any SINIT AC module it loads.

The SINIT AC module must be located on a 4 KB aligned memory location. The SINIT AC module must be mapped WB using the MTRRs and all other memory must be mapped to one of the supportable memory types returned by GETSEC[PARAMETERS]. The MTRRs that map the SINIT AC module must not overlap more than 4 KB of memory beyond the end of the SINIT AC image. See the GETSEC[ENTERACCS] instruction and the Authenticated Code Module Format, Appendix A.1, for more details on these restrictions.

The pages containing the SINIT AC module image must be present in memory before attempting to launch the measured environment. The SINIT AC module image must be loaded in the lower 4 GB of memory. System software should check that the SINIT AC module will fit within the AC execution region as specified by the

GETSEC[PARAMETERS] leaf. System software should not utilize the memory immediately after the SINIT AC module up to the next 4 KB boundary. On certain Intel® TXT implementations, execution of the SINIT AC module will corrupt this region of memory.

2.2.3.1 Matching an AC Module to the Platform

As part of system software loading an SINIT AC module, the system software should first verify that the file to be loaded is really an SINIT AC module. This may be done at installation time or runtime. Lines 1 - 13 in Listing 3 below show how to do this.

Each AC module is designed for a specific chipset or set of chipsets, platform type, and, optionally, processor(s). Software can examine the Chipset ID and Processor ID Lists embedded in the AC module binary to determine which chipsets and processors an AC module supports. Software should read the chipset's TXT.DIDVID register and parse the Chipset ID List to find a matching entry. If the AC module also contains a Processor ID List, then software should also match the AC module against the processor CPUID and IA32_PLATFORM_ID MSR. If the ACM Info Table version is 5 or greater, software should verify that the Platform Type bits within the *Capabilities* field match that of the current platform (server versus client). Attempting to execute an AC module that does not match the chipset and processor, and platform type when specified, will result in AC module failing to complete its normal execution and Intel® TXT Shutdown.

Listing 3. AC Module Matching Pseudocode

```
TXT_ACM_HEADER                *AcmHdr;        // see Table 8.
TXT_CHIPSET_ACM_INFO_TABLE    *InfoTable;     // see Table 10

//
// Find the Chipset AC Module Information Table
//
1. AcmHdr = (TXT_ACM_HEADER *)AcmImageBase;
2. UserAreaOffset = (AcmHdr->HeaderLen + AcmHdr->ScratchSize)*4;
3. InfoTable = (TXT_CHIPSET_ACM_INFO_TABLE *)(AcmBase +
                                              UserAreaOffset);

//
// Verify image is really an AC module
//
4. IF (InfoTable->UUID0 != 0x7FC03AAA) OR
5.    (InfoTable->UUID1 != 0x18DB46A7) OR
6.    (InfoTable->UUID2 != 0x8F69AC2E) OR
7.    (InfoTable->UUID3 != 0x5A7F418D) {
8.      Fail: not an AC module;
9. }

//
// Verify it is an SINIT AC module
//
10.  IF (AcmHdr->ModuleType != 2) OR
11.     (InfoTable->ChipsetACMType != 1) {
```

```
12.     Fail: not an SINIT AC module;
13.  }

//
// Verify that platform type and platform match, if specified
//
14.  IF (InfoTable->Version > 5) {
15.      IF (InfoTable->Capabilities[7:6] != 01 AND
16.          PlatformType == CLIENT) {
17.          Fail: Non-client ACM on client platform
18.      }
19.      IF (InfoTable->Capabilities[7:6] != 10 AND
20.          PlatformType == SERVER) {
21.         Fail: Non-server ACM on server platform
22.      }
23.  }



//
// Verify AC module and chipset production flags match
//
24.  IF (AcmHdr->Flags[15] == TXT.VER.EMIF[31]) {
25.        Fail: production flags mismatch;
26.  }

//
// Match AC module to system chipset
//
TXT_ACM_CHIPSET_ID_LIST   *ChipsetIdList;    // see Table 11
TXT_ACM_CHIPSET_ID        *ChipsetId;        // see Table 12

32.  ChipsetIdList = (TXT_ACM_CHIPSET_ID_LIST *)
33.     (AcmImageBase + InfoTable->ChipsetIdList);

//
// Search through all ChipsetId entries and check for a match.
//
34.  FOR (i = 0; i < ChipsetIdList->Count; i++) {
35.      //
36.      // Check for a match with this ChipsetId entry.
37.      //
38.      ChipsetId = ChipsetIdList->ChipsetIDs[i];
39.      IF ((TXT.DIDVID[VID] == ChipsetId->VendorId) &&
40.          (TXT.DIDVID[DID] == ChipsetId->DeviceId) &&
41.          ((((ChipsetId->Flags & 0x1) == 0) &&
42.            (TXT.DIDVID[RID] == ChipsetId->RevisionId)) ||
43.          (((ChipsetId->Flags & 0x1) == 0x1) &&
44.            (TXT.DIDVID[RID] & ChipsetId->RevisionId != 0)))) {
45.          AC module matches system chipset;
46.          GOTO CheckProcessor;
47.      }
48.  }
```

```
49.  Fail: AC module does not match system chipset;

CheckProcessor:
//
// Match AC module to processor
//
TXT_ACM_PROCESSOR_ID_LIST   *ProcessorIdList;   // see Table 13
TXT_ACM_PROCESSOR_ID        *ProcessorId;       // see Table 13

50.  ProcessorIdList = (TXT_ACM_PROCESSOR_ID_LIST *)
51.     (AcmImageBase + InfoTable->ProcessorIdList);

//
// Search through all ProcessorId entries and check for a match.
//
52.  FOR (i = 0; i < ProcessorIdList->Count; i++) {
53.      //
54.      // Check for a match with this ProcessorId entry.
55.      //
56.      ProcessorId = ProcessorIdList->ProcessorIDs[i];
57.      IF (ProcessorId->FMS ==
58.          (cpuid[1].EAX & ProcessorId->FMSMask)) &&
59.         (ProcessorId->PlatformID ==
60.          (IA32_PLATFORM_ID MSR & ProcessorId->PlatformMask))
61.         AC module matches processor;
62.      }
63.  }
64.  Fail: AC module does not match processor;
```

2.2.3.2 Verifying Compatibility of SINIT with the MLE

Over time, new features and capabilities may be added to the SINIT AC module that can be utilized by an MLE that is aware of those features. Likewise, features or capabilities may be added that require an MLE to be aware of them in order to interoperate properly. In order to expose these features and capabilities and permit the MLE and SINIT to determine whether they support a compatible set, the MLE header contains a *Capabilities* field (see Table 3) that corresponds to the *Capabilities* field in the SINIT AC module Information Table (see Table 10).

In addition, the *MinMleHeaderVer* field in the AC module information table allows SINIT to indicate that it requires a certain minimal version of an MLE. This allows for new behaviors or features requiring MLE support that may not be present in older versions.

Listing 4 shows the pseudocode for the MLE to determine if it is compatible with the provided SINIT AC module.

While lines 4 – 6 may be redundant with current SINIT AC modules if the MLE supports both RLP wakeup mechanisms, they permit graceful handling of future changes.

Listing 4. SINIT/MLE Compatibility Pseudocode

```
//
// Check that SINIT supports this version of the MLE
//
1. IF (InfoTable->MinMleHeaderVer > MleHeader.Version) {
2.     Fail:  SINIT requires a newer MLE
3. }

//
// Check that the known RLP wakeup mechanisms are supported
//
4. IF (MLE does NOT support at least one RLP wakeup mechanism
   specified in InfoTable->Capabilities) {
5.     Fail:  RLP wakeup mechanisms are incompatible
6. }
```

2.2.4 Loading the MLE and Processor Rendezvous

2.2.4.1 Loading the MLE

System software allocates memory for the MLE and MLE page table. The MLE is not required to be loaded into physically contiguous memory. The pages containing the MLE image must be pinned in memory and all these pages must be located in physical memory below 4 GB.

System software creates an MLE page table structure to map the entire MLE image. The pages containing the MLE page tables must be pinned in memory prior to launching the measured environment. The MLE page table structure must be in the format of the IA-32 Physical Address Extension (PAE) page table structure.

The MLE page table has several special requirements:

- The MLE page tables may contain only 4 KB pages.
- A breadth-first search of page tables must produce increasing physical addresses.
- Neither the MLE nor the page tables may overlap certain regions of memory:
  - device memory (PCI, PCIe*, etc.)
  - addresses between [640k, 1M] or above Top of Memory (TOM)
  - ISA hole (if enabled)
  - the Intel® TXT heap or SINIT memory regions
  - Intel® VT-d DMAR tables
- There may not be any invalid (not-present) page table entries after the first valid entry (i.e., there may not be any gaps in the MLE's linear address space).
- The Page Directories must be in a lower physical address than the Page Tables.

- The Page-Directory-Pointer-Table must be in a lower physical address than the Page-Directories.
- The page table pages must be in lower physical addresses than the MLE.

Later, the SINIT AC module will check that the MLE page table matches these requirements before calculating the MLE digest. The second rule above implies that the MLE must be loaded into physical memory in an ordered fashion: a scan of MLE virtual addresses must find increasing physical addresses. The system software can order its list of physical pages before loading the MLE image into memory.

The MLE is not required to begin at linear address 0. There may be any number of invalid/not-present entries in the page table prior to the beginning of the MLE pages (i.e., first valid page). The starting linear address should be placed in the *FirstValidPage* field of the MLE header structure (see Section 2.1).

If the MLE is to use this page table after launch, it needs to ensure that the entry point page is identity-mapped so that when it enables paging post-launch, the physical address of the instruction after paging is enabled will correspond to its linear address in the paged environment.

System software writes the physical base address of the MLE page table's page directory to the Intel® TXT Heap. The size in bytes of the MLE image is also written to the Intel® TXT Heap.

2.2.4.2 Intel® Trusted Execution Technology Heap Initialization

Information can be passed from system software to the SINIT AC module and from system software to the MLE using the Intel® TXT Heap. The SINIT AC module will also use this region to pass data to the MLE.

The system software launching the measured environment is responsible for initializing the following in the Intel® TXT Heap memory (this initialization must be completed before executing GETSEC[SENTER]):

- Initialize contents of the Intel® TXT Heap Memory (see Section C)
- Initialize contents of the *OsMleData* (see Appendix C.3) and *OsMleDataSize* (with the size of the *OsMleData* field + 8H) fields.
- Initialize contents of the *OsSinitData* (see Appendix C.4) and *OsSinitDataSize* (with the size of the *OsSinitData* field + 8H) fields.

The *OsSinitData* structure has fields for specifying regions of memory to protect from DMA (PMR Low/High Base/Size) using Intel® VT-d. As described in Section 1.11, the MLE must be protected from DMA by being contained within either the DMA Protected Range (DPR) or one of the Intel® VT-d Protected Memory Regions (PMRs). If the MLE resides within the DPR then the PMR fields of the *OsSinitData* structure may be set to 0. Otherwise, these fields must specify a region that contains the MLE and the page tables. However, the PMR fields can specify a larger region (and separate region, since

there are two ranges) than just the MLE if there is additional data that should be protected.

If the system software is using Intel® VT-d DMA remapping to protect areas of memory from DMA, then it must disable this before it executes GETSEC[SENTER]. In order to do this securely, system software should determine what PMR range(s) are necessary to cover the entire address range being DMA protected using the remapping tables. It should then initialize the PMR(s) appropriately and enable them before disabling remapping. The PMR values it provides in the *OsSinitData* PMR fields must correspond to the values that it has programmed. Once the MLE has control, it can re-enable remapping using the previous tables (after validating them).

Note: Intel IOMMU HW has an ability and functional need to store certain data in memory protected by PMRs. In order to prevent misuse of this ability, SINIT requires the DMA Address Translation to be disabled before SENTER. Additionally, it forcefully disables Queued Invalidation and Interrupt Remapping features.

Note: SINIT supports the following PMR states on entry:
- PMRs are disabled and no PMR enabling requests exist in OsSinitData structure. SINIT will verify that all required memory regions (MLE code and page tables, LCP data file) reside in DPR;
- PMRs are disabled and PMR enabling requests exist in OsSinitData structure. SINIT will enable PMRs per request;
- PMRs are enabled and their settings are identical to those, requested in OsSinitData structure. SINIT will verify PMR settings against request;
- PMRs are enabled and no PMR enabling requests exist in OsSinitData structure. SINIT will assume that this is an attack and abort execution.

If the MLE or subsequent code will be enabling Intel® VT-d DMA remapping, then the DMAR information that will be needed should be protected from malicious DMA activity until the remapping tables can be established to protect it. The SINIT AC module makes a copy of the DMAR tables in the *SinitMleData* region (located at an offset specified by the *SinitVtdDmarTable* field). Because this region is within the TXT heap, it is protected from DMA by the DPR. If the MLE or subsequent code does not use this copy of the DMAR tables, then it should protect the original tables (within the ACPI area) with the PMR range specified to SINIT. Likewise, the memory range used for the remapping tables should also be protected with the PMRs until remapping is enabled.

If the Platform Owner TXT launch control policy (LCP - see Section 3.2.1 for more details) is of type *POLTYPE_LIST,* then there must be an associated data file that contains the remainder of the policy. This *Policy Data File* must be placed in memory by system software and its starting address and size specified in the LCP PO Base and LCP PO Size fields of the *OsSinitData* structure. The data must be wholly contained within a DMA protected region of memory, either within the DPR (e.g., in the TXT heap) or within the bounds specified for the PMRs.

2.2.4.3 Rendezvousing Processors and Saving State

Listing 5 shows the pseudo-code for rendezvousing and saving states of all processors.

Line 1: If launching the measured environment after operating system boot, then all processors should be brought to a rendezvous point before executing GETSEC[SENTER]. At the rendezvous point each processor will set up for GETSEC[SENTER] and save any state needed to resume after the measured launch. If processors are not rendezvoused before executing SENTER, then the processors that did not rendezvous will lose their current operating state including possibly the fact that an in-service interrupt has not been acknowledged.

Lines 2 – 7: All processors check that they support SMX and enable SMX in CR4.SMXE.

Lines 8 - 10: The MLE should preserve machine check status registers if bit 6 in the TXT Extension Flags returned by GETSEC[PARAMETERS] (see Section 2.2.5.1 for details) is set. If this bit returns 0 or parameter type '5' is not supported, the MLE must log and clear machine check status registers.

Line 11: Check that certain CR0 bits are in the required state for a successful measured environment launch.

Line 12: System software allocates memory to save its state for restoration post measured launch. The *OsMleData* portion of the Intel® TXT Heap is reserved for this purpose (see Appendix C.2), though the size must be set appropriately for the memory to be available.

Line 13: The ILP saves enough state in memory to allow a return to OS execution after the measured launch and then continues launch execution. The RLPs save enough space in memory to allow return to OS execution after measured launch then execute HLT or spin waiting for transition to the measured environment.

Certain MSRs are modified by executing the GETSEC[SENTER] instruction. For example, bits within the IA32_MISC_ENABLE and IA32_DEBUGCTL MSRs are set to predetermined values. It may be desirable to restore certain bits within these MSRs to their pre-launch state after the MLE launch. If this is desired, then before executing GETSEC[SENTER], software should save the contents of these MSRs in the *OsMleData* area. The launched software can restore the original values into these MSRs after the GETSEC[SENTER] returns or, alternatively, the MLE can restore these MSRs with their original values during MLE initialization.

It is expected that most MLEs will want to restore the MTRR and *IA32_MISC_ENABLE* MSR states after the MLE launch, to provide optimal performance of the system.

Listing 5. Pseudo-code for Rendezvousing Processors and Saving State

```
1. Rendezvous all processors;

   //
   // The following code is run on all processors
```

```
   //
   // Enable SMX
   //

2. CPUID(EAX=1);
3. IF (SMX not supported) OR (VMX not supported) {
4.    Fail measured environment startup;
5. } ELSE {
6.    CR4.SMXE = 1;
7. }

8. IF (GETSEC[PARAMETERS](type=5)[6] == 1) {
9.     Clear Machine Check Status Registers;
10.    }
11.    Ensure CR0.CD=0, CR0.NW=0, and CR0.NE=1;

   //
   // Save current system software state in Intel TXT Heap
   //

12.    Allocate memory for OsMleData;
13.    Fill in OsMleData with system software state (including
   MTRR and IA32_MISC_ENABLE MSR states);
```

2.2.5 Performing a Measured Launch

2.2.5.1 MTRR Setup Prior to GETSEC[SENTER] Execution

System software must set up the variable range MTRRs to map all of memory (except the region containing the SINIT AC module) to one of the supported memory types as returned by GETSEC [PARAMETERS] before executing GETSEC [SENTER]. System software first saves the current MTRR settings in the *OsMleData* area and verifies that the default memory type is one of the types returned by GETSEC [PARAMETERS] (default memory type is specified in the IA32_MTRR_DEF_TYPE MSR). Next, the variable range MTRRs are set to map the SINIT AC module as WB, making sure that each variable MTRR base is a multiple of that MTRR's size. The SINIT AC module must be covered by the MTRRs such that no more than (4K-1) bytes after the module are mapped WB. For example, if an SINIT AC module is 11KB bytes in size, an 8KB and a 4KB or three 4KB MTRRs should be used to map it, not a single 32KB MTRR. Any unused variable range MTRRs should have their valid bit cleared. If the 8th bit of the ACM's Info Table *Capabilities* field is clear, the mask MTRRs covering the SINIT AC module should not set any bits beyond bit 35 (which corresponds to a 36-bit physical address space), or SINIT will treat this as an error condition. If that bit is set, the mask should cover the full range indicated by the *MAXPHYADDR* MSR

Listing 6 shows the pseudo-code for correctly setting the ILP and RLP MTRRs. This code follows the recommendation in the *IA-32 Software Developer's Manual*.

After MTRR setup is complete, the RLPs mask interrupts (by executing CLI), signal the ILP that they have interrupts masked, and execute halt. Before executing GETSEC [SENTER], the ILP waits for all RLPs to indicate that they have disabled their interrupts. If the ILP executed a GETSEC [SENTER] while an RLP was servicing an interrupt, the interrupt servicing will not complete, possibly leaving the interrupting device unable to generate further interrupts.

Listing 6. MTRR Setup Pseudo-code

```
//
// Pre-MTRR change
//

1. Disable interrupts (via CLI);
2. Wait for all processors to reach this point;
3. Disable and flush caches (i.e. CRO.CD=1, CR0.NW=0, WBINVD);
4. Save CR4
5. IF (CR4.PGE == 1) {
6.     Clear CR4.PGE
7. }
8. Flush TLBs
9. Disable all MTRRs (i.e. IA32_MTRR_DEF_TYPE.e=0)

//
// Use MTRRs to map SINIT memory region as WB, all other regions
// are mapped to a value reported supportable by
// GETSEC[PARAMETERS]
//

10.  Set default memory type (IA32_MTRR_DEF_TYPE.type) to one
     reported by GETSEC[PARAMETERS];
11.  Disable all fixed MTRRs (IA32_MTRR_DEF_TYPE.fe=0);
12.  Disable all variable MTRRs (clear valid bit);
13.  Read SINIT size from the SINIT AC header;
14.  Program variable MTRRs to cover the AC execution region,
     memtype=WB (re-enable each one used),make sure each variable
     MTRRs base must be a multiple of that MTRR's size;
//
// Post-MTRR changes
//
15.  Flush caches (WBINVD);
16.  Flush TLBs;
17.  Enable MTRRs (i.e. MTRRdefType.e=1);
18.  Enable caches (i.e. CRO.CD=0, CR0.NW=0);
19.  Restore CR4;
20.  Wait for all processors to reach this point;
21.  Enable interrupts;
//
// RLPs stop here
//

22.  IF (IA32_APIC_BASE.BSP != 1) {
```

```
23.      CLI;
24.      set bit indicating we have interrupts disabled;
25.      HLT;
26.  }

27.  Wait for all RLPs to signal that they have their interrupts
     disabled
```

2.2.5.2 Selection of Launch Capabilities

System software must select the capabilities that it wishes to use for the launch. It must choose a subset of the capabilities supported by the SINIT AC module. For mandatory capabilities, such as the RLP wakeup mechanism, one of the supported options must be chosen.

```
28.  OsSinitData.Capabilities = selected capabilities;
```

2.2.5.3 TPM Preparation

System software must ensure that the TPM is ready to accept commands and that there is no currently active locality (*TPM.ACCESS_x.activeLocality* bit is clear for all localities) before executing the GETSEC[SENTER] instruction.

```
29.  Read TPM Status Register until it is ready to accept a
     command
30.  For all localities x, ensure that ACCESS_x.activeLocality is
     0
```

2.2.5.4 Intel® Trusted Execution Technology Launch

The ILP is now ready to launch the measuring process. System software executes the GETSEC[SENTER] instruction. See chapter 6, “Safer Mode Extensions Reference”, in the *Intel 64 and IA-32 Software Developer Manuals, Volume 2B* for the details of GETSEC[SENTER] operation.

```
31.  EBX = Physical Base Address of SINIT AC Module
32.  ECX = size of the SINIT AC Module in bytes
33.  EDX = 0
34.  GETSEC[SENTER]
```

2.3 MLE Initialization

This section describes the initialization of the MLE. Listing 7 shows the pseudo-code for MLE initialization.

The MLE initialization code is executed on the ILP when the SINIT AC module executes the GETSEC[EXITAC] instruction—the MLE initialization code is the first MLE code to run after GETSEC[SENTER] and within the measured environment. The SINIT AC module obtains the MLE initialization code entry point from the *EntryPoint* field in the MLE Header data structure whose address is specified in the *OsSinitData* entry in the

Intel® TXT Heap. The MLE initialization code is responsible for setting up the protections necessary to safely launch any additional environments or software. The initialization includes Intel® TXT hardware initialization, waking and initializing the RLPs, MLE software initialization and initialization of the STM (if one is being used). This section describes the details of MLE initialization.

During MLE initialization, the ILP executes the GETSEC[WAKEUP] instruction, bringing all RLPs into the MLE initialization code. Each RLP gets its initial state from the MLE JOIN data structure (see the *Intel 64 and IA-32 Software Developer Manual, Volume 2B, Table 6-11*). The ILP sets up the MLE JOIN data structure and loads its physical address in the TXT.MLE.JOIN register prior to executing GETSEC[WAKEUP]. Generally, the RLP initialization code will be very similar to the ILP initialization code.

If the MLE restores any state from the environment of the launching system software, it must first validate this state before committing it. This is because state from the environment prior to the GETSEC[SENTER] instruction is not considered trustworthy and could lead to loss of MLE integrity.

Lines 1 – 8: The MLE loads CR3 with the MLE page directory physical address and enables paging. The SINIT AC module has just transferred control to the MLE with paging off, now the MLE must setup its own environment. The MLE's GDT is loaded at line 3 and the MLE does a far jump to load the correct MLE CS and cause a fetch of the MLE descriptor from the GDT. At line 5 a stack is setup for the MLE initialization routine and, at line 6, the MLE segment registers are loaded. Next the MLE loads its IDT and initializes the exception handlers.

All instructions and data that were used before paging was enabled must reside on the same physical page as the MLE entry point and must access data with relative addressing. This is because the page tables may have been subverted by untrusted code prior to launch and so the MLE entry point's page may have been copied to a different physical address than the original. The MLE must also verify that this page is identity mapped prior to enabling paging (to ensure that the linear address of the instruction following enabling of paging is the same as its physical address).

If the MLE cannot guarantee that it was loaded at a fixed address, then it must create the identity mapping dynamically. Because the physical address of the identity page could overlap with the virtual address range that the MLE wants to use in its page tables, the MLE may need to create a trampoline page table. In such case, the trampoline page table would consist of an identity-mapped page for the physical address of the MLE entry point and a virtual address mapping of that page which is guaranteed not to be within the desired address range (i.e., a trampoline page). That virtual address mapping would also need to be added to the page table that the MLE ultimately wants to run on. This way the MLE can enable paging to the trampoline page table (at the identity mapped address) and then jump to the trampoline page's address and then switch page tables (CR3's) to the final table where it will begin executing at the virtual address of the trampoline page but in the final page table.

If the MLE does not enable paging, then it must also validate that the physical addresses specified in the page table used for the launch are the expected ones. And as above, it must do this in code that resides on the same physical page as the MLE entry point and must use only relative addressing. The reason for this validation is that the page table could have been altered to place the MLE pages at different physical addresses than expected, without having altered the MLE measurement.

Because the MLE page table that was used for measurement does not contain pages other than those belonging to the MLE, if it wishes to continue to run in a paged environment it will need to either extend the page tables to map the additional address space needed (e.g., TXT configuration space, etc.) or create new page tables. This should be done after it has finished establishing a safe environment. The cacheability requirements for the address space of any MLE-established page tables must follow the guidelines below.

Line 9: The MLE checks the MTRRs which were saved in the *OsMleData* area of the Intel® TXT Heap (see Appendix C.3). It looks for overlapping regions with invalid memory type combinations and variable MTRRs describing non-contiguous memory regions. If either of these checks fails, the MLE should fail the measured launch or correct the failure.

Before the original MTRRs are restored, the MLE must ensure that all its own pages will be mapped with defined memory types once the variable MTRRs are enabled. The MLE must ensure that the combined memory type as specified by the page table entry and variable MTRRs results in a defined memory type.

The MLE must also ensure that the TXT Device Space (0xFED20000 – 0xFED4FFFF) is mapped as UC so that accesses to these addresses will be properly handled by the chipset.

Line 10: The MLE should check that the system memory map that it will use is consistent with the memory regions and types as specified in the Memory Descriptor Records (MDRs) returned in the *SinitMleData* structure. Alternately, the MLE may use this table as its map of system memory. This check is necessary as the system memory map is most likely generated by untrusted software and so could contain regions that, if used for trusted code or secrets, might lead to compromise of that data. If the MLE will be using PCI Express* devices, it should verify that it is accessing their configuration space through the address range specified by the PCIE MDR type (3).

Line 11: The MLE should also verify that the Intel® VT-d PMR settings that were used by SINIT to program the Intel® VT-d engines, as specified in the *OsSinitData* structure, contain the expected values. While the MLE can only be launched if the settings cover itself and its page tables (or the pages fall within the DPR), settings beyond these regions could have been subverted by untrusted code prior to the launch.

Line 12: The ILP must re-enable SMIs that were disabled as part of the SENTER process; most systems will not function properly if SMIs are disabled for any length of time. It is recommended that the MLE enable SMIs on the ILP before enabling them on

the RLPs, since some BIOS SMI handlers may hang if they receive an SMI on an AP and cannot generate one on the BSP to rendezvous all threads. Newer CPUs may automatically enable SMIs on entry to the MLE; for such CPUs there is no harm in executing GETSEC[SMCTRL].

Lines 13 – 17: If this is the ILP then the MLE does the one-time initialization, builds the MLE JOIN data structure and wakes the RLPs. This structure contains the physical addresses of the MLE entry point and the MLE GDT, along with the MLE GDT size and must be located in the lower 4GB of memory. The ILP writes the physical address of this structure to the TXT.MLE.JOIN register. An RLP will read the startup information from the MLE JOIN data structure when it is awakened. The MLE writer should ensure that the MLE JOIN data structure does not cross a page boundary between two non-contiguous pages. The MLE image must be built or loaded such that its GDT is located on a single page. Enough of the RLP entry point code must be on a single page to allow the RLPs to enable paging.

Lines 18 – 27: The MLE must look at the *OsSinitData.Capabilities* field to see which RLP wakeup mechanism was chosen by the pre-SENTER code and thus used by SINIT. If the MLE wants to enforce that certain capabilities or wakeup mechanism was used, then it can choose to error if it finds that not to be the case. For future compatibility, MLEs should support both RLP wakeup mechanisms.

Line 30: The MLE checks several items to ensure they are consistent across all processors:

- All processors must have consistent SMM Monitor Control MSR settings. The processors must all be opt-in and have the same MSEG region or the processors must be all opt-out.
- Ensure all processors have compatible VMX features. The compatible VMX features will depend on the specific MLE implementation. For example, some implementation may require all processors support Virtual Interrupt Pending.
- Ensure all processors have compatible feature sets. Some MLE implementations may depend on certain feature being available on all processors. For example, some MLE implementation may depend on all processors supporting SSE2.

  If the MLE will use VMX then it should verify that bit 1 (VMX in SMX operation) in the *IA32_FEATURE_CONTROL* MSR is set. Bit 2 (VMX outside SMX operation) may also be set depending on the BIOS being used and on whether TXT has been enabled.
- Ensure all processors have a valid microcode patch loaded or all processors have the same microcode patch loaded. This check will depend on the specific MLE implementation. Some MLE implementations may require the same patch be loaded on all processors, other MLE implementations may contain a microcode patch revocation list and require all processors have a microcode patch loaded which is not on the revocation list.

Line 31: The MLE must wakeup the RLPs while the memory type for the SINIT AC module region is WB cache-able. This is a requirement of the MONITOR mechanism for RLP wakeup. Since this is not guaranteed to be true of the original MTRRs, it is safest to wait until after the RLPs have been awakened before restoring the MTRRs to their pre-SENTER values. Alternatively, the MLE could ensure that this is the case and adjust the MTRRs if it is not. It could then restore the MTRRs before waking the RLPs. In either case, when restoring the MTRRs they should be made the same for each processor.

Line 32: The MLE should restore the *IA32_MISC_ENABLE* MSR to the value saved in the *OsMleData* structure. This MSR was set to predefined values as part of SENTER in order to provide a more consistent environment to the authenticated code module. Most MLEs should be able to safely restore the previous value without any need to verify it. The MLE should wait until the RLPs are awakened before restoring the MSR in case the original MSR did not have the Enable MONITOR FSM bit (18) set. See Appendix E for the processor state of the ILP after SENTER and the states of the RLPs after waking.

Line 33: The machine-check exceptions flag (CR4.MCE) is cleared by the GETSEC[SENTER] instruction. If the MLE supports the machine-check architecture, then it should initialize the exception mechanism and enable exception reporting.

Line 34: The MLE enables SMIs on each RLP.

Line 35: The MLE enables VMX in the CR4 register. This is required before any VMX instruction can be executed.

Line 36: The MLE allocates and sets up the root controlling VMCS then executes VMXON, enabling VMX root operation.

Lines 37 – 41: The MLE sets up the guest VM. At line 37 the MLE allocates memory for the guest VMCS. This memory must be 4KB aligned. The MLE executes VMCLEAR with a pointer to this VMCS in order to mark this VMCS clear and allow a VMLAUNCH of the guest VM. At line 39 the MLE executes VMPTRLD so that it can initialize the VMCS at line 40. Now at line 41 the guest VM is launched for the first time.

Note: On the last extend of the TPM by the SINIT AC module, it may not wait to see if the command is complete – so the MLE needs to make sure that the TPM is ready before using it.

Listing 7. MLE Initialization Pseudo-code

```
//
// MLE entry point – ILP and RLP(s) enter here
//

1. Load CR3 with MLE page table pointer (OsSinitData.MLE
   PageTableBase);
2. Enable paging;
3. Load the GDTR with the linear address of MLE GDT;
4. Long jump to force reload the new CS;
5. Load MLE SS, ESP;
```

```
6. Load MLE DS, ES, FS, GS;

7. Load the IDTR with the linear address of MLE IDT;
8. Initialize exception handlers;

//
// Validate state
//
9. Check MTRR settings from OsMleData area;
10. Validate system memory map against MDRs
11. Validate VT-d PMR settings against expected values

//
// Enable SMIs
//
12. execute GETSEC[SMCTRL];

//
// Wake RLPs
//
13. IF (ILP) {
14.     Initialize memory protection and other data
   structures;
15.     Build JOIN structure;
16.     TXT.MLE.JOIN = physical address of JOIN structure;
17.     IF (RLP exist) {
18.         IF (OsSinitData.Capabilities is set to MONITOR
   wakeup mechanism) {
19.             SinitMleData.RlpWakeupAddr = 1;
20.         }
21.         ELSE IF (OsSinitData.Capabilities is set to GETSEC
   wakeup mechanism) {
22.             GETSEC[WAKEUP];
23.         }
24.         ELSE {
25.             Fail: Unknown RLP wakeup mechanism;
26.         }
27.     }
28. }

29. Wait for all processors to reach this point;
30. Do consistency checks across processors;
31. Restore MTRR settings on all processors;
32. Restore IA32_MISC_ENABLE MSR from OsMleData
33. Enable machine-check exception handling
34. RLPs execute GETSEC[SMCTRL]


//
// Enable VMX
//
35. CR4.VMXE = 1;
```

```
//
// Start VMX operation
//
36. Allocate and setup the root controlling VMCS, execute
   VMXON(root controlling VMCS);

//
// Set up the guest container
//
37. Allocate memory for and setup guest VMCS;
38. VMCLEAR guest VMCS;
39. VMPTRLD guest VMCS;
40. Initialize guest VMCS from OsMleData area;

//
// All processors launch back into guest
//
41. VMLAUNCH guest;
```

2.4 MLE Operation

The dispatch routine is responsible for handling all VMExits from the guest. The guest VMExits are caused by various situations, operations or events occurring in the guest. The dispatch routine must handle each VMExit appropriately to maintain the measured environment. In addition, the dispatch routine may need to save and restore some of processor state not automatically saved or restored during VM transitions. The MLE must also ensure that it has an accurate view of the address space and that it restricts access to certain of the memory regions that the GETSEC[SENTER] process will have enabled. The following subsections describe various key components of the MLE dispatch routine.

2.4.1 Address Space Correctness

It is likely that most MLEs will rely on the e820 memory map to determine which regions of the address space are physical RAM and which of those are usable (e.g., not reserved by BIOS). However, as this table is created by BIOS it is not protected from tampering prior to a measured launch. An MLE, therefore, cannot rely on it to contain an accurate view of physical memory.

After a measured launch, SINIT will provide the MLE with an accurate list of the actual RAM regions as part of the *SinitMleData* structure of the Intel® TXT Heap (see Appendix C.5). The *SinitMDR* field of this data structure specifies the regions of physical memory that are valid for use by the MLE. This data structure can also be used to accurately determine SMRAM and PCIe extended configuration space, if the MLE handles these specifically.

2.4.2 Address Space Integrity

There are several regions of the address space (both physical RAM and Intel® TXT chipset regions) that have special uses for Intel® TXT. Some of these should be reserved for the MLE and some can be exposed to one or more guests/VMs.

2.4.3 Physical RAM Regions

There are two regions of physical RAM that are used by Intel® TXT and are reserved by BIOS prior to the MLE launch. These are the SINIT AC module region and the Intel® TXT Heap. Each region's base address and size The Intel® TXT configuration registers (e.g., TXT.SINIT.BASE and TXT.SINIT.SIZE) specify each region's base address and size.

The SINIT and Intel® TXT Heap regions are only required for measured launch and may be used for other purposes afterwards. However, if the measured environment must be re-launched (e.g., after resuming from the S3 state), the MLE may wish to preserve and protect these regions.

2.4.4 Intel® Trusted Execution Technology Chipset Regions

There are two Intel® TXT chipset regions: Intel® TXT configuration register space and Intel® TXT Device Space. These regions are described in Appendix B.

2.4.4.1 Intel® Trusted Execution Technology Configuration Space

The configuration register space is divided into public and private regions. The public region generally provides read only access to configuration registers and the MLE may choose to allow access to this region by guests. The private region allows write access, including to the various command registers. This region should be reserved to the MLE to ensure proper operation of the measured environment.

2.4.4.2 Intel® Trusted Execution Technology Device Space

The Intel® TXT Device Space supports access to TPM localities. Localities three and four are not usable by the MLE even after the measured environment has been established, and so do not need any special treatment. Locality two is unlocked when the Intel® TXT private configuration space is opened during the launch process. Locality one is not usable unless it has been explicitly unlocked (via the TXT.CMD.OPEN.LOCALITY1 command). If the MLE wants to reserve access to locality two for itself then it needs to ensure that guest/VM access to these regions behaves as a TPM abort, as defined by TCG for non-accessible localities. This behavior is that memory reads return FFh and writes are discarded. The MLE can provide this behavior by any one of the following:

- Trapping guest/VM accesses to the regions and emulating the defined behavior.
- Instead, it could map these regions onto one of the hardware-reserved localities (three or four) and let the hardware provide the defined behavior.

- If the MLE does not need access to locality 2 then it can close this locality (TXT.CMD.CLOSE.LOCALITY2) so neither itself nor guests will have access to it.

Note: Addresses FED45000H – FED7FFFFH are Intel reserved for expansion.

2.4.5 Device Assignment

If the MLE exposes devices to untrusted VMs, it must take care to completely protect itself from any affects (either intentional or otherwise) of these devices. For devices which use DMA to access memory, the MLE can protect itself using Intel® VT for Directed IO (Intel® VT-d) to prevent unwanted access to memory and through VMX to manage access to device configuration space. For other types of devices, their interactions with the system, and how they affect it, should be fully understood before allowing an untrusted VM to access them.

2.4.6 Protecting Secrets

If there is data in memory whose confidentiality must be maintained, then the MLE should set the Intel® TXT secrets flag so that the Intel® TXT hardware will maintain protections even if the measured environment is lost before performing a shutdown (e.g., hardware reset). Writing to the TXT.CMD.SECRETS configuration register can do this. The teardown process will clear this flag once it has scrubbed memory and removed any confidential data.

2.4.7 Model Specific Register Handling

Model Specific Registers (MSRs) pose challenges for a measured environment. Certain MSRs may directly leak information from one guest to another. For example, the Extended Machine Check State registers may contain secrets at the time a machine check is taken. Other MSRs might be used to indirectly probe trusted code. The non-trusted guest could use the Performance Counter MSRs, for example, to determine secrets (e.g., keys) used by the trusted code. Other MSRs can modify the MLE's operation and destroy the integrity of the measured environment.

The VMX architecture allows the MLE to trap all guest MSR accesses. Certain VMX implementations will also allow the MLE to use a bitmap to selectively trap MSR accesses. The MLE must use these VMX features to check certain guest MSR accesses, ensuring that no secrets are leaked and that MLE operation is not compromised.

An MLE might virtualize some of the MSRs. The VMX architecture provides a mechanism to automatically save selected guest MSRs and load selected MLE MSRs on VMEXIT. Selected guest MSRs may be automatically loaded on VMENTER. These features allow the MLE to virtualize MSRs, keeping a separate MSR copy for the guest and MLE. Note that using this feature will slow VMEXIT and VMENTER times. The VMX architecture provides a separate set of VMCS registers for the automatic saving and restoring of the fast system call MSRs.

There is a limit to the number of MSRs that can be swapped during a VMX transition. Bits 27:25 of the VMX_BASE_MSR+5 indicate the recommended maximum number of MSRs that can be saved or loaded in VMX transition MSR-load or MSR-store lists. Specifically, if the value of these bits is N, then 512 * (N + 1) is the recommended maximum number of MSRs referenced in each list. If the limit is exceeded, undefined processor behavior may result (including a machine check during the VMX transition).

There are certain MSRs that cannot be included in the MSR-load or MSR-store lists. In the initial VMX implementations, *IA32_BIOS_UPDT_TRIG* and *IA32_BIOS-SIGN_ID* may not be loaded as part of a VM-Entry or VM-Exit. The list of MSRs that cannot be loaded in VMX transitions is implementation specific.

The MLE must contain a built-in policy for handling guest MSR accesses. This MSR handling policy must deal with all architectural MSRs that might be accessed by guest code. The built-in MSR policy must deny access to all non-architectural MSRs.

2.4.8 Interrupts and Exceptions

To preserve the integrity of the measured environment, the MLE must be careful in how it handles exceptions and interrupts. It needs to ensure that its IDT (Interrupt Descriptor Table) has a handler for all exceptions and interrupts. The MLE should also ensure that if it uses interrupts for internal signaling that it does so securely. Likewise, it is best if exception handlers do not try and recover from the exception but instead properly terminate the environment.

2.4.9 ACPI Power Management Support

Certain ACPI power state transitions may remove or cause failure to the Intel® TXT protections. The MLE must control such ACPI power state transitions. The following sections describe the various ACPI power state transitions and how the MLE must deal with these state transitions.

2.4.9.1 T-state Transitions

T-states allow reduced processor core temperature through software-controlled clock modulation. T-state transitions do not affect the Intel® TXT protections, so the MLE does not need to control T-state transitions. The MLE may wish to control T-state transitions for other purposes (e.g., to enforce its own power management or performance policies).

2.4.9.2 P-State Transitions

P-state transitions allow software to change processor operating voltage and frequency to improve processor utilization and reduce processor power consumption. On some systems, where the processor does not enforce allowed combinations, the MLE must ensure software does not write an invalid combination into the GV3 MSRs.

2.4.9.3 C-State Transitions

C-states allow the processor to enter lower power state. The C0 state is the only C-state where the processor is actually executing code – in the remaining C-states the processor enters a lower power state and does not execute code. In these lower power C-states the Intel® TXT protections remain intact; therefore, the MLE does not need to monitor or control the C-state transitions. The MLE may wish to control C-state transitions for other purposes (e.g., to enforce its own power management or scheduling policy).

2.4.9.4 S-State Transitions

The S0 state is the system working state – the remaining S-states are low-power, system-wide sleep states. Software transitions from the S0 working state to the other S-states by writing to the PM1 control register (PM1_CNT) in the chipset. Since the Intel® TXT protections are removed when the system enters the S3, S4 or S5 states, and the BIOS will gain control of the system on resume from these states, the MLE must remove secrets from memory before allowing the system to enter one of these sleeps states. Note that entering S1 does not remove Intel® TXT protections and Intel chipsets do not support the S2 sleep state.

The Intel® TXT chipset provides hardware to detect when the software attempts to enter a sleep state while the secrets flag is set (the TXT.SECRETS.STS bit of the TXT.E2STS register). The Intel® TXT chipset will reset the system if it detects a write to the PM1_CNT register that will force the system into S3, S4 or S5 while the secrets flag is set. If the Intel® TXT chipset does detect this situation and resets the system, then the BIOS AC module or code within Trusted BIOS ensures that memory is scrubbed before passing control onward. To avoid this reset and scrubbing process, the MLE should remove secrets from memory and teardown the Intel® TXT environment before allowing a transition to S3, S4 or S5.

Before tearing down the Intel® TXT environment, the MLE may remove secrets from memory (clearing pages with secrets) or encrypt secrets for later use (e.g., for a later measured environment launch). Once this operation is complete the MLE must issue the TXT.CMD.NO-SECRETS command to clear the secrets flag. After this command is issued, the MLE may allow a transition to S3, S4 or S5 sleep state. The MLE teardown procedure is described in more detail in Section 2.5.

2.4.9.4.1 S3

The S3 state provides special challenges for the MLE because the resume process uses the in-memory state from when S3 was entered. This means that unlike a normal boot process where trust is established as each component launches, the trust that existed at entry to S3 must be maintained/verified on resume.

Since the TXT environment must be torn down before entering S3, it will have to be re-established on resume. This part of the S3 resume process is nearly identical to the

original launch. Because S3 resume should leave the platform in the same state as before S3 was entered, the PCRs should also have the same values. This means that the MLE launched on resume should be the same as the initial one launched, which means that the code/data being measured cannot include any state from before entering S3. If some state from before entering S3 is needed on resume, then it must be validated post-launch (since it is not being measured).

The MLE needs to ensure that the integrity of the TCB will be maintained across the transition. There are two possible sources for loss of integrity across S3: malicious DMA and compromise of the in-memory BIOS image. The initial, pre-S3 TXT launch process protects against DMA and so the S3 resume process should maintain such protection. Most BIOS will execute portions of their S3 resume code from their in-memory image without first re-copying it from flash. Since this in-memory image could have been modified by any privileged software or firmware that executed as part of the original, pre-S3 boot process (e.g., option ROMs, bootloader, etc.), this too needs to be defended against.

The TXT launch process done as part of S3 resume ensures integrity and DMA protection for the measured part of the MLE itself. For the remainder of the TCB this can be accomplished by creating a memory integrity code for the TCB and sealing it to the MLE's launch-time measurements just before the TXT protections are removed prior to entering S3 (the sealed data creation attributes should include a locality that is only available to the TCB). On resume and after a successful launch, the MLE can re-calculate the value for the memory image and compare it with the sealed value to determine if the memory image has been compromised).

If there were additional measurements extended to the DRTM PCRs as part of the original boot process, these will need to be re-established since these PCRs are cleared when the MLE is re-launched. The entity that makes the measurements should seal the measurement values (not the resulting PCR values) to the PCRs values in effect just before it extends the measurements into the PCRs (the sealed data creation attributes should include a locality that is only available to software trusted by the entity). On resume from S3, when that entity is resumed it will unseal the values and re-extend them into the appropriate PCRs.

It may be necessary for the MLE to seal additional information that is required to securely re-establish the trusted environment. For instance, the portion of the TCB needed to re-establish final DMA protections (e.g., with VT-d DMA remapping) will need to be DMA protected by the MLE as part of the post-resume launch. The MLE may need to save the bounds of this region prior to entering S3 (both in plain text and sealed). It would then use the plain text saved bounds to determine the PMR values to specify as part of the re-launch. Post-launch the MLE would unseal the bounds information and verify it against the bounds specified in the launch.

Because the boot process on resuming from an S3 state does not re-measure the elements of the SRTM, software prior to entering the S3 state must execute the appropriate TPM command for the current TPM mode to inform the TPM to preserve

the state of its PCRs: for 1.2 this is TPM_SaveState; for 2.0 this is TPM2_Shutdown (requesting state). Upon resume from S3, the BIOS must provide a flag to TPM_Startup to indicate that the TPM is to restore the saved state. If the TPM's state is not saved prior to entering S3, then the TPM will be non-functional after resuming. Normally an OS TPM driver would perform the TPM state preservation command when the OS indicated that it was entering S3. However, if the MLE cannot be sure that the environment it establishes will perform this command, it may wish to do so itself prior to entering S3. If the MLE alters the TPM state (e.g., extending to PCRs, etc.) after TPM state preservation command has been issued then the TPM may invalidate the previously saved state. In such cases, the MLE must also perform this command and it should be the last TPM command that is executed, in order to ensure that the state is not changed afterwards. There is no harm if this command is executed multiple times prior to S3.

2.4.10 Processor Capacity Addition (aka CPU Hotplug)

VMMs and OS kernels not accommodating or executing within an Intel® TXT measured launch assume control of application processors during boot using INIT-SIPI-SIPI mechanism. Upon receipt of a SIPI, the processor resumes execution at the specified SIPI vector.

On the other hand, an MLE issues GETSEC[WAKEUP] (or a write to the wakeup address) to assume control of application processors (RLPs) following SENTER. The RLPs begin execution at an address pointed to by the MLE JOIN data structure. Intel® TXT for multiprocessor platforms enables processor capacity addition (also known as CPU hotplug or hotadd) after the Intel® TXT environment has been launched. Processor capacity addition can be a result of physical addition of a processor package to a running system or bringing a processor package online that was previously inactive. The Intel® TXT processor capacity addition flow makes use of INIT-SIPI-SIPI as the RLP wakeup mechanism, but the hot added logical processors use the MLE JOIN data structure to determine their entry point.

The processor capacity addition flow for an MLE is documented below:

4. New processors are released from reset. They execute measured BIOS code.
5. BIOS configures the new processors. At the end of configuration, BIOS clears the BSP flag in the IA32_APIC_BASE register of new processors and leaves them in a CLI/HLT loop.
6. The MLE is notified of this event via the standard ACPI mechanism.
7. Some MLEs may choose to allow write access to the Intel® TXT public configuration region by guests. However, during the processor capacity addition flow, the MLE must prevent guests from writing to the public region in order to prevent them from modifying the TXT.MLE.JOIN register in the middle of this flow.
8. The MLE must prepare the MLE JOIN data structure for the new processors. It may choose to use the same values as it did for the initial RLPs, or it may use different ones. In either case, they are subject to the same restrictions as for the initial RLP

wakeup. The MLE then writes the physical address of the JOIN data structure into the TXT.MLE.JOIN register.

9. The MLE issues the INIT–SIPI-SIPI sequence to each newly added logical processor to take control of these processors.
10. In response to the SIPI, each new processor will detect that this is a capacity addition to an existing measured environment and resume execution at the entry point specified in the MLE JOIN data structure. The processor will ignore the SIPI vector that may have been supplied. The new processor gets its initial state from the MLE JOIN data structure just like an RLP would during the MLE launch process.

2.5 MLE Teardown

This section describes an orderly measured environment teardown. This occurs when the guest OS or the MLE decides to tear down the measured environment (for example prior to entering an ACPI sleep state such as S3). The listing below shows the pseudo-code for teardown of the measured environment.

Line 1: Rendezvous all processors at "exiting Intel® TXT environment" point in guest. No need for the guests to save their state as their state will be stored in a VMCS on VMEXIT to the monitor.

Lines 2 and 3: After all processors in the guest rendezvous, all processors execute a VMCALL to the teardown routine in the MLE. Once in the MLE, each processor increments a counter in trusted memory. All processors except the BSP/ILP (the processor with IA32_APIC_BASE MSR.BSP=1) wait on a memory barrier. The ILP waits for all other processors to enter MLE teardown routine then signals the other processors to resume with teardown.

If one or more processors fail to reach the rendezvous in the guest, the ILP may timeout and VMCALL to the MLE teardown routine. If one or more processors fail to arrive in the MLE teardown routine, the ILP forces all other processors into the MLE with an NMI IPI. Both these conditions are treated as errors – the ILP should proceed with the measured environment teardown but log an error.

At line 4, each processor reads all guest state from its VMCS and stores this data in memory, since after VMXOFF the processors will no longer be able to access data in their VMCS. This state will be needed to restore the guest execution after teardown.

The MLE automatically saves certain guest state (general purpose registers which are not part of the VMCS guest area) on VM Exit. The MLE may need to restore this state when it reenters the guest after the GETSEC[SEXIT].

Line 5: Once all processors are in the MLE and have saved guest state from the VMCS, all processors clear their appropriate registers to remove secrets from these registers.

Lines 6: All processors flush VMCS contents to memory using VMCLEAR. The MLE must flush any VMCS that might contain secrets – this would include all guest VMCSes in a multi-VM environment.

Line 7: The processors wait until all processors have reached this point before resuming execution. This allows all the VMCS flushes to complete before the ILP encrypts or scrubs secrets. Processors should execute an SFENCE to ensure all writes are completed before continuing.

Line 9: The ILP encrypts and stores exposed secrets from all trusted VMs. Note that encrypted secrets will have to be stored in memory until the OS can put them to disk. This will require extra memory above and beyond the memory holding secrets. This step assumes that the RLPs do not have secrets that are not visible to the ILP. Therefore, when the ILP scrubs/encrypts all secrets, this will deal with secrets in the RLP caches also.

Line 10: The ILP again clears appropriate registers to remove any secrets from those registers.

Line 11: The ILP scrubs all trusted memory (except the teardown routine itself and encrypted memory). Note that the scrub itself clears secrets still held in the cache.

Line 12: The ILP executes WBINVD to invalidate its caches (to ensure last few pages of zeros get to memory).

Lines 13 - 16: If the MLE is going to enter the S3 state, the ILP calculates a memory integrity code and seals it.

Line 17: The ILP caps, or extends, the dynamic PCRs with some value. This prevents an attacker from unsealing the secrets after the teardown using the same PCRs, since the dynamic PCRs are not reset after GETSEC[SEXIT].

Line 18: The ILP writes the NoSecrets in memory command. (TXT.CMD.NO-SECRETS)

Line 19: The ILP should unlock the system memory configuration (TXT.CMD.UNLOCK-MEM-CONFIG) (that was locked by SINIT) once secrets have been removed from memory. This will facilitate re-launching the MLE and may be necessary for a graceful shutdown of the system.

Line 20: The ILP closes Intel® TXT private configuration space.

Line 23: The RLPs wait while the ILP encrypts and scrubs secrets from memory.

Line 25: Each processor then disables processor virtualization. If an STM was launched, it must be torn down before VMX is disabled. See *Intel 64 and IA-32 Software Developer Manual, Volume 3B,* Section 25.15.7 for more information.

Lines 26 - 31: The RLPs wait on a memory barrier while the ILP executes the GETSEC[SEXIT] instruction to initiate the teardown of SMX operation.

At end of GETSEC[SEXIT], the ILP simply continues to the next instruction (still running in monitor's context – paging on). The ILP signals the RLPs to continue.

Lines 32 and 33: The former monitor code now restores guest state left behind when the guest executed the VMCALL to enter the MLE teardown routine. All processors perform the transition to guest OS, now operating as normal environment rather than guest.

The guest MSRs must be restored when restarting the guest OS. The MLE can restore the MSRs with information in the VMCS (VM-exit MSR store count) and the VM-exit MSR store area, or the guest OS could save important MSR settings before calling the teardown routine and restore its own MSR settings after resuming after teardown.

If the MLE is going to return control to a designated guest after tearing down, then the MLE must ensure that no interrupts are left pending or not serviced before returning control to the designated guest. Any interrupts left pending or not serviced may prevent further interrupt servicing once the designated guest is restarted.

Listing 8. Measured Environment Teardown Pseudo-code

```
1. Rendezvous processors in guest OS;
2. All processors VMCALL teardown in MLE;
3. Rendezvous all processors in MLE teardown routine;
4. All processors read guest state from VMCS, store values in
   memory;

//
// Remove and encrypt all secrets from registers and memory
//
5. All processors clear their appropriate registers;
6. All processors flush VMCS contents to memory using VMCLEAR;
7. Wait for all processors to reach this point;
8. IF (ILP) {
9.    Encrypt and store secrets in memory;
10.    Again, clear appropriate registers to remove secrets;
11.    Scrub all trusted memory;
12.    WBINVD caches;
13.    IF (S3) {
14.       Create memory integrity code
15.       Seal memory integrity code
16.    }
17.    "cap" dynamic TPM PCRs;
18.    Write to TXT.CMD.NO-SECRETS;
19.    Unlock memory configuration
20.    Close private Intel TXT configuration space;
21.    Signal RLPs that scrub is complete;
22. } ELSE { // RLP
23.    Wait for ILP to signal completion of memory scrub;
24. }

//
// Stop VMX operation
```

```
//
25. VMXOFF;


//
// RLPs wait while ILP executes SEXIT
//
26. IF (ILP) {
27.    GETSEC[SEXIT];
28.    signal completion of SEXIT;
29. } ELSE {
30.    wait for ILP to signal completion of SEXIT;
31. }

//
// Transition back to the guest OS

32. Restore guest OS state from device memory;
33. Transition back to guest OS context;
```

2.6 Other Considerations

2.6.1 Saving MSR State across a Measured Launch

Execution of the GETSEC[SENTER] instruction loads certain MSRs with pre-defined values. For example, GETSEC[SENTER] will load *IA32_DEBUGCTL* MSR with 0H and will load the *GV3* MSR with a predetermined value. The software can deal with this in several different ways. The launching software may save the state of these MSRs before measured launch and restore the state after the launch returns. In this case the MLE will need to check the values that are restored. Another approach is to have the launch software save the desired state and have the MLE restore the values before resuming the guest. The software could also leave these MSRs in the state established by GETSEC[SENTER].

The *IA32_MISC_ENABLE* MSR should be saved and restored around measured launch and teardown.

§

3.0 Verifying Measured Launched Environments

Launch Control Policy (LCP) is the verification mechanism for the Intel® TXT verified launch process. LCP is used to determine whether the current platform configuration or the environment to be launched meets specified criteria. Policies may be defined by the Platform Owner, as well as leveraged from Boot Guard (BtG) verifications.

LCP allows the Platform Owner (PO) to specify environments that may be launched by the GETSEC[SENTER] instruction.

The addition of LCP to the platform indirectly now makes them more manageable. Measured Launched Environments (MLEs) no longer must worry about sealing secrets to several platform configurations when they know a policy is in place to check these before their launch takes place; they just need to seal to a representation of this policy. This becomes useful when the platform's configuration changes (e.g., the BIOS is replaced), because the new BIOS configurations can be added to the policy the value to which any MLE secret is sealed to need not change – thus reducing the burden of migrating the MLEs secrets if another component in the platform changes.

LCP requirements can be summarized as:

- Provide the Platform Owner with the ability to control which measured environments can be immediately launched by the GETSEC[SENTER] instruction.
- Provide the Platform Owner with the ability to control which platform configurations, as measured by the Static Root of Trust, are permitted to perform GETSEC[SENTER].
- Provide the Platform Owner with an ability to control which STM is permitted to run in SMRAM and MLE suppliers with an ability to require STM presence.
- Provide this mechanism in a manner which prevents it from being undermined by untrusted software.
- If no policy is present on the platform then the platform shall continue the Intel® TXT launch process allowing any configuration and MLE to be launched.

The policy described in this section applies to TXT-capable platforms produced in 2017 and later. Policy definitions for earlier platforms can be found in earlier versions of this document.

Note: LCP contains number of alternative structures used in TPM 1.2 and TPM 2.0 modes of operation with similar names and differing only by an additional suffix.
In all cases when differences between those structures are unimportant for the content being discussed, they will be referred to as group in which additional suffix is included in parenthesis.
Examples: LCP_POLICY_LIST2 and LCP_POLICY_LIST2_1 will be referred to as

LCP_POLICY_LIST2(_1); LCP_POLICY and LCP_POLICY2 will be referred to as LCP_POLICY(2) and so on.

3.1 Overview

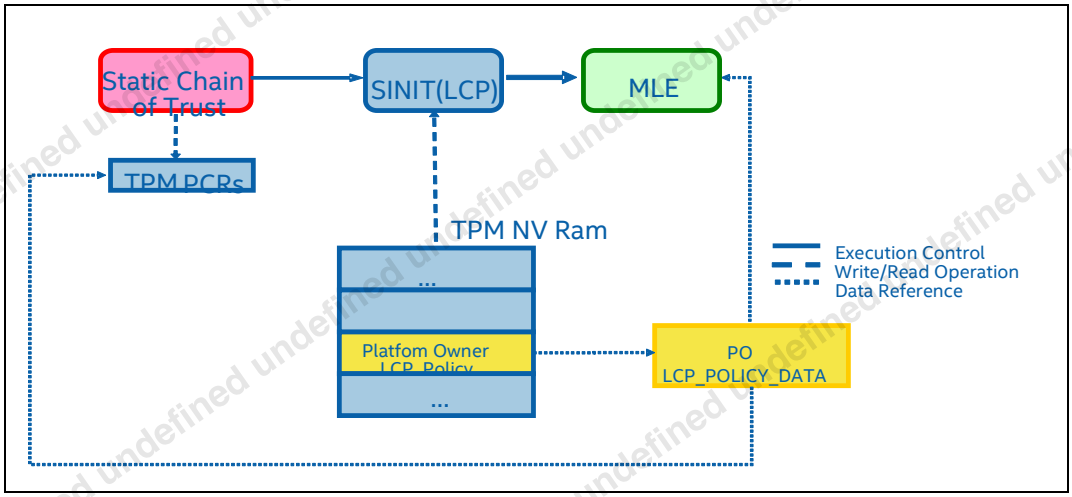
The Launch Control Policy architecture consists of the following components:

- LCP Policy Engine. This is part of the SINIT ACM which enforces the policies stored on the platform.
- LCP Policy. This policy takes a form of structure residing in the TPM PO NV index. This structure defines some of the policies and creates a linkage to LCP *Policy Data File*.
- LCP *Policy Data File*. Linked to a policy structure in the TPM PO index, is structured to be a nested collection of lists and valid policy elements, such as measurements of MLEs, platform configurations, and other objects.

Figure 1 shows how these components relate to each other.

When the platform boots, its state is measured and recorded by the Static Root of Trust for Measurement (SRTM) and the other components which make up the static chain of trust; these events occur from when the platform is powered on until the Intel® TXT measured launch (or until some component breaks the static chain). At this point GETSEC[SENTER] is invoked, and control is passed through the Authenticated Code Execution Area (ACEA) to the SINIT authenticated code module. The LCP engine in SINIT reads the PO LCP Policy Index in the TPM NV, decides which policy to use, and checks the Platform Configuration and the Measured Launched Environment as required by the chosen policy. The measured environment is then launched if the policy is satisfied.

Figure 1. Launch Control Policy Components



3.1.1 Versions of LCP Components

The following Section 3.2 provides detailed description of LCP components and data structures they operate.

Appendix D contains all the minutia of structure formats.

Since introduction of CBnT two versions of LCP structures used in TPM 1.2 mode and TPM 2.0 mode of operation were essentially morphed together though some of the structures remain TPM family specific. Thus, in the following text, where applicable, each of the structures supporting just as single TPM family is clearly marked as such.

3.2 LCP Components. General provisions

The policy that the SINIT AC module implements is named as Platform Owner (PO) policy. PO policy is stored in the non-volatile memory of the Trusted Platform Module (TPM NV). By storing the policy in the TPM NV, access controls can be applied to it; it also enables the policy to persist across platform power cycles.

Besides of PO TPM NV index, Intel® TXT also uses other TPM NV indices: The Auxiliary TXT index (AUX) and the Software Guard Extension index (SGX).

The AUX index is used as internal inter-ACM communication area. Main data it contains is:

- BIOS ACM registration data to be extended by SINIT into PCR17 since BIOS ACM is in TXT Trusted Computing Base (TCB).
- Revocation data described in Section 3.7
- Digests of various components passed from S-ACM to SINIT

The SGX index is used as BIOS / MLE inter-communication area. Its usage is described in Section 4.5

3.2.1 LCP Policy

The *LCP_POLICY* structure (for a full listing see Appendix D.1.2) is used for the Platform Owner policy. The size of the structure currently needs to be kept to a minimum in order to preserve the scarce resources of the TPM NV storage, which is why additional structures for Platform Owner policy (*LCP_POLICY_DATA*) that can persist elsewhere are provided to handle additional information.

Figure 2. LCP_POLICY (2) Structure

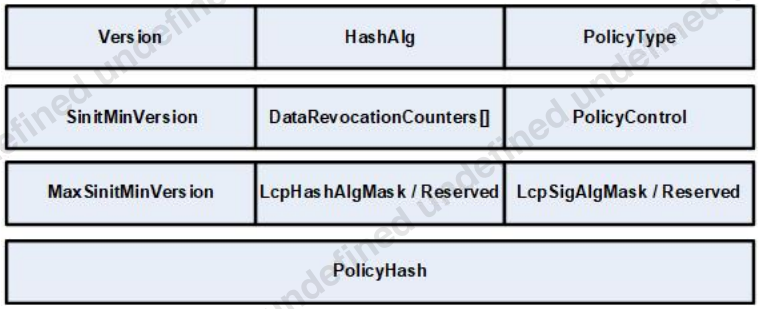


Figure 2 diagrammatically illustrates main fields of the LCP_POLICY and LCP_POLICY2 structures (fields not to scale):

Version specifies the version of the structures and, implicitly, of the policy engine semantics. It is of the format *<major>.<minor>* where the major version is the MSB of the field and the minor version is the LSB. All minor versions of a given major version will be backwards compatible. If new fields are added they will be at the end and the semantics of all previous minor versions are maintained (though they can be extended). Major versions are not guaranteed to be backwards compatible with each other thus SINIT will fail to launch if it finds a major version that it is not compatible with.

The version of the LCP_POLICY structure is defined as 2.4 (0x0204) and the version of the LCP_POLICY2 structure is 3.2 (0x0302)

HashAlg identifies the hashing algorithm used for the *PolicyHash* field. If the algorithm type is not supported by ACM processing the policy, then it shall stop processing the policy and fail.

PolicyType indicates whether an additional LCP_POLICY_DATA structure is required.

- If the *PolicyType* field is LCP_POLTYPE_ANY, then the value in the *PolicyHash* field is ignored and the environment to be launched is simply measured before execution control is passed to it. No corresponding LCP_POLICY_DATA is expected.

- If the *PolicyType* field is LCP_POLTYPE_LIST, then the value of *PolicyHash* is the result of computing a hash over the LCP_POLICY_DATA structure per the rules below.

If the type specified is not supported by the ACM processing the policy, then it shall stop processing the policy and fail.

SinitMinVersion specifies the minimum version of SINIT that can be used. This value corresponds to the *AcmVersion* field in the AC module Information Table (see Table 10). This value must be less than or equal to the value of *AcmVersion* in the executing SINIT image for that SINIT to continue; otherwise SINIT will fail the launch. If there is an LCP_MLE_ELEMENT or LCP_MLE_ELEMENT2 element in a policy, then the *SinitMinVersion* in that element will be combined with the value in the LCP_POLICY, per the description in Appendix D.4.4 and D.4.7. This is done to allow MLE supplier to request a minimum version of SINIT that MLE requires. If there is no *LCP_MLE_ELEMENT* element in the policy, then the value in the *LCP_POLICY* will be used.

DataRevocationCounters is an array of elements corresponding to an array of policy lists in the LCP policy data file. This array provides a mechanism to revoke signed lists in that object. Index of elements in this array corresponds to LCP List (LCP_POLICY_LIST2, LCP_POLICY_LIST2_1) number in the LCP policy data file (LCP_POLICY_DATA) and contains a minimum *RevocationCounter* value in the signature of this Policy List (LCP_SIGNATURE2, LCP_SIGNATURE2_1) which will be accepted by a policy engine. If the value in the *RevocationCounter* field is less than this value, then SINIT will fail the launch.

Example: if an LCP policy list which is LCP_POLICY_DATA[#N] is signed and has its *RevocationCounter* field set to M, *DataRevocationCounters* [#N] element must be less or equal to M, otherwise SINIT will fail.

For any LCP_POLICY_LISTs that are not signed, the corresponding *DataRevocationCounters* index will be ignored. Values in *DataRevocationCounters* indices not corresponding to existing lists in the policy data file will be ignored as well.

The *PolicyControl* field contains policy bits with global LCP impact. Its content is deciphered in Appendix D.1.1. Field is essentially simplified with only two bits left:

- Bit 1 tells a Non-Production Worthy SINIT that it can run. This puts Owner in control of what kind of SINIT can be executed in production environment
- Bit 3 signifies how platform configuration via PCONF element is enforced – just platform owner's selected configuration or both platform owner's and platform supplier's configurations. Detailed description can be found in Appendix J.2.3

Note: Bit 1 Identifies whether the platform will allow AC Modules marked as pre-production to be used to launch the MLE. If this bit is 0 and a pre-production AC Module has been invoked, it will cause a TXT reset during GETSEC[SENTER]. The use of any pre-production AC Module will result in PCRs 17 and 18 being capped with random values.

The *MaxSinitMinVersion* field specifies the maximum value of *AUX.AUXRevocation.SinitMinVer* this policy will allow a revocation utility to set. The value of "0" in this field means the SINIT minimum version may not be changed and "0xFF" allows unconditional changes to the SINIT minimum version.

Two new fields *LcpHashAlgMask* and *LcpSignAlgMask* are defined in LCP_POLICY2 structure. They specify hash and signature algorithm sets Platform Owner tells SINIT to grant. If LCP engine encounters a list with unauthorized signature algorithm, it will abort.

If the LCP engine encounters elements with unauthorized hash algorithms, it will skip them not allowing to match but will record the fact that given element type exists and will require a match for this element type to be satisfied upon the end of a scan.

PolicyHash field is described in Section 3.2.1.1

3.2.1.1 PolicyHash Field for LCP_POLTYPE_LIST

For policies of type LCP_POLTYPE_LIST, the LCP_POLICY_DATA may contain multiple lists, some of which are signed and some of which are not. In order to realize the value of signed policies, *PolicyHash* can't be a simple hash over the entire *LCP_POLICY_DATA* or even changes to signed policy lists would cause a change in the measurement of the policy.

The measurement of a policy list depends on whether the list is signed.

For a list signed using any algorithm supported by the ACM, the measurement is the hash (as specified by the *HashAlg* field of the corresponding LCP_POLICY) of the public (verification) key in the *PubkeyValue* member (it is called *Modulus* in RSA_PUBLIC_KEY structure of LCP_SIGNATURE2_1) of the *Signature* field for RSA, or the *Qx* and *Qy* public coordinates for elliptic curve signatures.

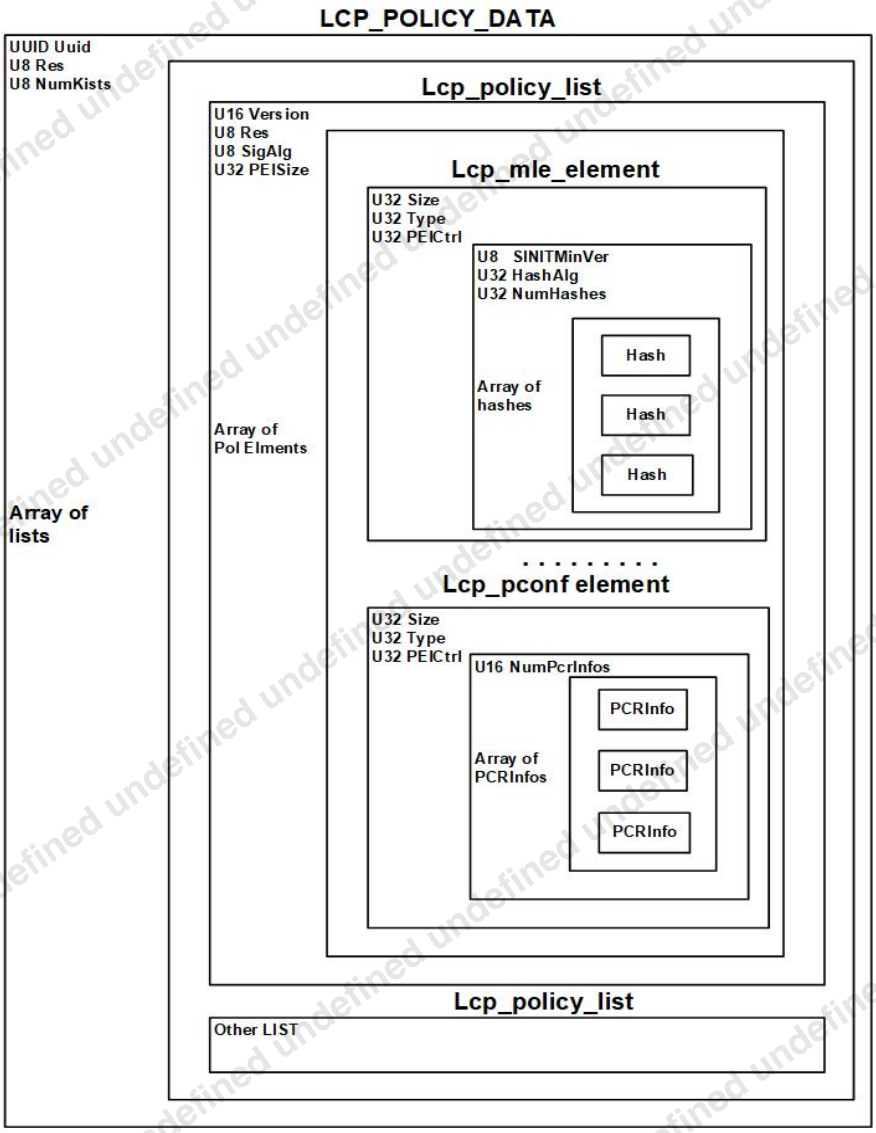
For unsigned lists (with *SigAlgorithm* = LCP_POLSALG_NONE or *KeySignatureOffset* = 0 – see Appendix D.3.1.1 and D.3.1.2 respectively), the measurement is the hash specified by the *HashAlg* field of the corresponding LCP_POLICY(2)) of the entire list (LCP_POLICY_LIST).

The value of the *PolicyHash* field will be the hash of all policy list measurements concatenated (there is no end padding if the number of lists present is less than LCP_MAX_LISTS), even if there is only one list measurement. For example, if there is only a single list then the value of *PolicyHash* will be $HASH_{HashAlg}(HASH_{HashAlg}(list))$.

3.2.2 LCP Policy Data

The purpose of the LCP_POLICY_DATA structure is to provide any additional data needed to enforce the policy but in a separate entity that doesn't have to consume TPM NV space. A full description of LCP_POLICY_DATA can be found in Appendix D.

Figure 3. LCP_POLICY_DATA Structure



The *PolicyLists[]* field allows the policy data file to contain a maximum of *LCP_MAX_LISTS*(currently, 8 lists) LCP policy lists. An LCP policy list may be either signed or unsigned and can contain any type of LCP_POLICY_ELEMENT structures (e.g., for MLE policy, platform configuration policy, etc.).

The structure of LCP policy list corresponds to the LCP_LIST structure – see Appendix D.3.2.3.

3.2.3 LCP Policy Element

Policy elements are self-describing entities that contain the actual policy conditions. Since they are self-describing, policy engine can ignore the elements that it fails to understand or does not support. This allows for adding new element types without breaking backwards compatibility. Element structures for all supported element types are fully described in Appendix D.4.

Figure 4. LCP_POLICY_ELEMENT structure

Size	Type	PolEltControl	Data[Size - 12]

Figure 4 illustrates the LCP_POLICY_ELEMENT structure (fields not to scale):

Size is the size (in bytes) of the entire LCP_POLICY_ELEMENT structure, including the type-specific *Data* and the *Size* field itself. The policy engine can use this to skip over an element that it does not understand or support.

Type is the type of the element. Currently CBnT supports the following types of elements: MLE, PCONF, STM, and Custom.

The *PolEltControl* provides several control bits divided into two groups of 16 bits each. One is specific to the element type and the other applies to all element types in a list (see Appendix D.4)

The contents of the *Data[]* field depend on the type of element (see Appendix D.4).

Each of the element types features two forms – for TPM 1.2 and TPM 2.0 families of devices. TPM 2.0 elements are enhancements to the TPM1.2 elements with augmentations for algorithm agility.

LCP policy lists can be comprised of both TPM 1.2 and TPM 2.0 element types. This minimizes space requirements for NV RAM, simplifies processing logic and allows to maintain the same number of authorities (8) since no authority will have to supply separate lists for TPM 1.2 and TPM 2.0 devices.

3.2.4 Signed Policies

The purpose of signed policies is to provide a mechanism that allows policy authors to update the list of permissible environments without having to update the TPM NV (note that if revocation is used, then the TPM NV must be updated to increment the revocation counter). This allows updates to be a simple file pushes rather than physical or remote platform touches. It also facilitates sealing against the policy, as sealed data does not have to be migrated when the policy is updated.

The use of this mechanism places certain responsibilities on policy authors:

- The private signature key always needs to be kept secure and under the control of the key owner.
- The private signature key needs to be strong enough for the full lifetime of the policy (this period has been estimated to be up to seven years).

3.2.5 Supported Cryptographic Algorithms

TPM 1.2

The following algorithms are defined for TPM 1.2 mode of execution of the Launch Control Policy:

Hashing – SHA1

Signature – RSASSA PKCS V1.5 / 2048- and 3072-bit keys / SHA1

TPM 2.0

- List Signatures

LCP lists may use the following signature algorithms:

- RSASSA PKCS V1.5 / 2048- and 3072-bit keys / SHA256 and SHA384
- RSAPSS / 2048- and 3072-bit keys / SHA256 and SHA384
- ECDSA, curves P256 and P384
- SM2

3.2.5.1 RSASSA, RSAPSS, and ECDSA

Support will be limited to versions RSASSA-PKCS1-v1_5, RSASSA-PSS and ECDSA as defined in "*NIST, FIPS PUB 186-4, Federal Information Processing Standards Publication, Digital Signature Standard (DSS), July 2013*"

Signatures will be supported only with the following approved hash functions: TPM_ALG_SHA1; TPM_ALG_SHA256; TPM_ALG_SHA384.

Supported key sizes will be 2048 and 3072 bits.

3.2.5.2 SM2

Implementation will follow the definition in "*State Cryptography Administration, Public Key Cryptographic Algorithm SM2 Based on Elliptic Curves, December 2010*"

Since SM2 is a variation of ECC, it will share format of signature structure used for ECCDSA with the following differences:

- It will use the SM3 hash algorithm

- It will default to Fp-256 curve as specified in "*Recommended Curve Parameters for Public Key Cryptographic Algorithm SM2 Based on Elliptic Curves*"

It is the responsibility of the policy author to ensure that their policy uses an algorithm supported by the version of the AC module being used. If the policy specifies an unsupported algorithm, it will fail and, depending on the ACM evaluating the policy, the environment will not be permitted to launch, or the platform will not boot.

3.2.5.3 List Signatures

Additional restriction of an applicable signature algorithms is associated with the *LcpHashAlgMask* and *LcpSignAlgMask* fields of the LCP_POLICY2 structure described in Section 3.2.1

Not only signature algorithm of a signed list must be permitted by *LcpSignAlgMask*, but *HashAlgIDs* used in signatures when creating an encoded message will be limited to those included in the *LcpHashAlgMask*.

Consider the following scenario: if the RSASSA signature is to use SHA256 to prepare encoded message, SHA256 must be permitted by *LcpHashAlgMask.* Otherwise the list will be skipped and not participate in establishing the LCP policy.

3.3 Policy Engine Logic

3.3.1 Policies

ALL TPM Modes

Before evaluating a policy, the policy engine must first verify the policy's integrity. For a policy of type LCP_POLTYPE_LIST, the engine must verify each LCP_POLICY_LIST2(_1) in the LCP_POLICY_DATA. In the signed list case, the signature must be verified as well. For an unsigned list, the hash of the list must be calculated. The hash of the LCP_POLICY_DATA structure is calculated per Section 3.2.1.1 above and then compared with the hash in the LCP_POLICY(2) that was read from the TPM NVRAM.

The policy engine must scan the policy for each policy element it supports. When a policy contains multiple lists in its LCP_POLICY_DATA, the policy engine will evaluate each list sequentially. As soon as it finds a match that satisfies the policy element being evaluated (e.g., MLE, platform configuration, etc.) it will stop evaluating further elements and lists. Exception of this rule is evaluation of LCP_PCONF_ELEMENT(2) which depends on the *PO.PolicyControl.Pconf_Enforced* enforcement flag per description in Appendix J.2.3.

For a policy of type LCP_POLTYPE_ANY, the policy engine will treat that policy as successfully evaluating every policy element type.

For a policy of type LCP_POLTYPE_LIST, for every policy element type supported by the ACM evaluating the policy that is present in any of the lists, at least one instance of that element type must evaluate successfully for the policy to succeed. If a particular policy element type is not in any of the lists, then that condition is not evaluated, and any state is accepted. For instance:

If SINIT is processing a policy that contains two lists, the first containing only an LCP_MLE_ELEMENT(2) and the second containing only an LCP_PCONF_ELEMENT(2), then the MLE being launched must appear in the first list's LCP_MLE_ELEMENT(2) and the current platform configuration must satisfy the second list's LCP_PCONF_ELEMENT(2); otherwise the launch will fail.

If SINIT is processing a policy that contains two lists, each containing only an LCP_MLE_ELEMENT(2) element, then the MLE being launched must appear in at least one list's LCP_MLE_ELEMENT(2). Since no other element types are present, any other platform condition or state is acceptable (e.g., any PCR values).

While Policy Engine logic described so far is applicable to any mode of operation, its logic in TPM 2.0 mode has additional peculiarities.

TPM 2.0 mode

ACMs' ability to evaluate LCP elements present in policy data file depends on its hashing capabilities originated from two sources - embedded software and TPM hashing commands. Based on these two sources ACM builds lists of supported hashing algorithms following logic described in Appendix G. Using these lists for evaluation of different element types is described below.

- Evaluation of MLE and STM policy elements

The evaluation of MLE and STM policy elements is determined by the *EFF_HashAlgIDList* as described in Appendix G.3. Elements of the above types hashed with *HashAlgIDs* not in this list will cause the policy engine abort.

Use of list is not *Extend Policy* dependent.

- Evaluation of PCONF LCP elements

The evaluation of PCONF elements depends on TPM capabilities and will be limited only by PCR banks supported by a given TPM (i.e., by *PCR_HashAlgIDList* as described in Appendix G.3).

Hash algorithm used to compute composite digest will be also determined by *LCP_TPM_HashAlgIDList* as described in Appendix G.3 ensuring that the *HashAlgID* associated with such elements is permitted by PO policy and that the PCRs implementing the PCONF *HashAlgID* indeed exist.

Use of *LCP_TPM_HashAlgIDList* list is not *Extend Policy* dependent.

3.3.2 Processing of Policy Data Files

Policy Engine in TPM1.2 mode processes all policy lists looking for TPM 1.2 style elements only. All unrecognized elements including all TPM 2.0 style elements are ignored in this mode.

Policy Engine in TPM2.0 mode processes only TPM 2.0 style policy elements. All unrecognized elements including all TPM 1.2 style elements are ignored in this mode.

Policy Engine scans the data file in two phases – the first phase is used to perform integrity verification (validation) of the file, second phase is used to enforce (evaluate) the policy.

During every scan, the policy engine processes all lists in the data file and all of the elements in list in the order of appearance.

3.3.2.1 Integrity Verification

During the verification phase the policy engine must validate all kinds of signatures of the policy data lists. The policy engine must exit with an error if signature cannot be validated for any reason.

Note: This is needed to thwart the following attack: if the policy engine skipped a signed list with an unrecognized signature algorithm, an adversary might simply change the signature algorithm identifier to something not supported by SINIT causing the entire list, which would otherwise be evaluated, to be skipped thus ignoring intended policies.

During the integrity verification phase, the overall hash of the policy data file is computed and compared to the value of the *PolicyHash* field, stored in the PO index.

The policy engine will check hash algorithms of all elements in all lists not skipped due the reasons mentioned earlier and abort with error if it finds one not supported by ACM. This requirement is aimed to thwart a special form of ACM substitution attack when an ACM is substituted by an analogous ACM which lacks support of some of the crypto-algorithms.

Other checks performed are verification of structure versions, sizes of all components TPM capabilities etc.

Any discovered integrity error during this phase causes platform reset.

TPM NV Indices

The policy engine checks TPM NV index attributes and sizes. In TPM 2.0 mode, it will run additional checks outlined below:

- *nameAlg* strength check.
  - When each of the indices is verified, the policy engine will compare nameAlg of the index to the minimal algorithm required and abort with error if it is not of the required strength.

  - The minimal required nameAlg is SHA256 or SM3, provided they are supported by current TPM. If the only TPM supported algorithm is SHA1, it will be accepted by the policy engine and no error will be generated.

- The policy engine will also ensure the following, as a matter of integrity:
  - PO.LcpHashAlgMask, PO.LcpSignAlgMask are not empty:
  - each bit set in a PO.LcpHashAlgMask mask means that a relevant hash algorithm is permitted; each bit set in PO.LcpSignAlgMask mask means that relevant combination of signature algorithm, key size, hash algorithm and curve is permitted.
  - All hash algorithms permitted by PO.LcpHashAlgMask and PO.LcpSignAlgMask are supported by ACM;
  - PO.HashAlg is permitted by PO.LcpHashAlgMask

3.3.2.2 Policy Enforcement

During the policy enforcement phase, the policy engine seeks elements in the following order: MLE; PCONF; STM. In short, the second phase is comprised of three sequential scans.

The policy engine will check revocation counters associated with lists and will skip any revoked lists and ignore their content.

In the TPM 2.0 mode, when evaluating list signatures, if a combination of signature algorithm, key size, hash algorithm, and curve is not permitted by *LcpSignAlgMask,* the policy engine will skip such list and ignore its content.

During each scan, elements of each type are evaluated in the order of appearance in the policy data file, irrespective to hash algorithms they are using.

In TPM 2.0 mode for each found element the policy engine will consult *LcpHashAlgMask*. It will process the element only if its *HashAlgID* is permitted by the mask, otherwise it will be skipped.

The policy engine will apply the following enforcement logic:

- It will record each new element type as “required” when it is first discovered.
- In the TPM 2.0 mode only, the “required” assertion is recorded only if the element’s hash algorithm is permitted by *LcpHashAlgMask*.
  Element types filtered out by *LcpHashAlgMask* will not trigger the “required” assertion.
- If a match for a “required” element type is not found at the end of evaluation, an error will be generated.

There are three possible results of policy enforcement:

- Successful policy evaluation (i.e., all required matches were found). Control is passed to further SINIT tasks;

- Assumed successful policy evaluation. Some of the element types were not found at all and are assumed to be successfully evaluated. Control is passed to further SINIT tasks;
- No match found error. Platform is reset.

3.4 Measuring the Enforced Policy

The LCP engine in SINIT will extend to PCR 17 a hash value which represents the policy or policies against which the environment was launched. This hash value is determined by the rules in the following sections.

It is important that for all policy cases a measurement will always be extended to PCR 17 in order to prevent the MLE from later extending a value of a policy that was not evaluated. This is an issue because PCR 17 is open to locality 2 extends and the MLE executes with locality 2 access. If this was not done, such MLE could get access to data that was sealed against some known policy by another MLE.

The above consideration is correct in general but was especially important for a legacy implementation of Intel® TXT when the event log creation was an optional SINIT function.

With CBnT the situation changed because:

- The event log creation is a mandatory SINIT function;
- There are several extends SINIT performs as a matter of integrity regardless of the LCP policy type;
- Last event in the extend sequence is always extended to both PCR 17 and 18 and is *not optional*. This makes it impossible for any MLE to deduce and extend into PCR any data, which would allow it to arrive at any predefined PCR value another MLE might have used to seal secrets.

3.4.1 No Policy Data and Allow Any Policy

When no owner policy is executed (i.e., the PO index is not provisioned, is provisioned with LCP_POLTYPE_LIST policy, but none of its policy elements are understood by the ACMs' policy engine), it contains no policy elements, or an owner policy exists and is of type LCP_POLTYPE_ANY, then:

- In TPM 1.2 mode *ZeroDigest* will is extended to PCR 17 and PCR 18;
- In TPM 2.0 mode number of extends are made to PCR 17 and PCR 18 with the last one *not optional* event.

3.4.2 Policy with LCP_POLICY_DATA

Because the measurement may contain the measurements of more than one policy list, it is important that SINIT ACMs for all platforms order the list measurements in the same way so that identical policy evaluations will extend PCR 17 with the same value.

The policy engine will order the policy list measurements according to the order in which it evaluates policy elements. For this version of the specification, the following policy element types are evaluated in this order: LCP_POLELT_TYPE_MLE, LCP_POLELT_TYPE_PCONF and LCP_POLTYPE_STM. If additional policy element types are supported in the future, their evaluation order will be specified.

The policy engine will not allow more than one list to be signed with the same signature key and will abort if such lists are found. This is because measuring of such lists produces the same digest, which may allow substitute attacks when lists are swapped.

The above principles are applicable to all generations of policy engines – legacy and combined, while the combined policy engine logic extends them to accommodate TPM 2.0 algorithm agility, new format of LCP structures, and new data combining possibilities.

3.4.3 Effective Policies

Effective LCP policies are those enforced by the policy engine. They fall into two categories: Effective LCP Policy Details, and Effective LCP Policy Authorities.

The effective LCP policy details value is the collection of all matching elements discovered by the policy engine. The value of effective LCP policy details is measured into PCR 17 – the “details” PCR. To achieve it, SINIT concatenates digests and other pertinent data describing such elements, hashes the obtained byte stream and extends the hash into PCR 17.

The effective value of LCP policy authorities is the collection of all lists, containing matching elements discovered by the policy engine. The value of effective LCP policy authorities is measured into PCR 18 – the “authorities” PCR. To achieve it SINIT concatenates list digests and other relevant data describing such lists, hashes the obtained byte stream and extends the hash into PCR 18.

Construction of the two-byte streams in TPM 1.2 and TPM 2.0 modes is, in principle, similar but due to the need to support algorithm agility there are some differences in low level technical details.

3.4.3.1 TPM 1.2 LCP Policy Details

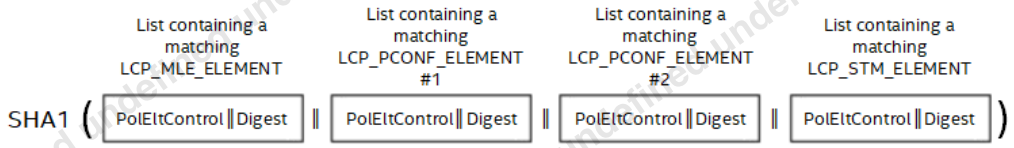
The effective LCP policy details hash represents all elements contributing to the currently established policy. Each of the elements will be represented by a descriptor being a concatenation of corresponding *PolEltControl* DWORD and a SHA1 digest.

Absent elements authenticated by default will be represented by the zero descriptor - 24 bytes of zeros.

Descriptors of elements will be concatenated in order: MLE, PCONF#1, PCONF#2, and STM, where PCONF #1 and PCONF #2 correspond to a configuration where the *Pconf_Enforce* bit in *PO.PolicyControl* field is set to 1.

The obtained byte stream is then hashed and extended into PCR 17. The process is diagrammatically depicted below:

Figure 5. TPM 1.2 LCP Policy Details Data Stream



3.4.3.2 TPM 1.2 LCP Policy Authorities

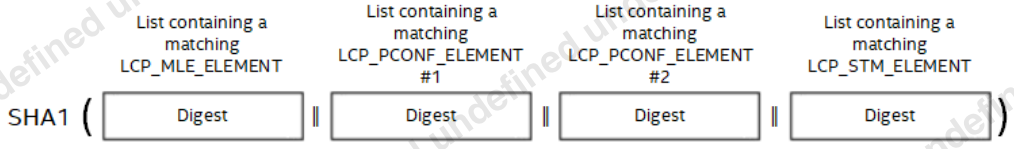
The effective LCP authorities hash provides aggregated information about all authorities (policy lists) contributing to the currently established policy. It is computed as follows: each authority is represented by a digest of the entire list if it is unsigned, or a digest of a public key otherwise.

Digests of lists containing matching elements will be concatenated in the same order as with LCP policy details: MLE, PCONF#1, PCONF#2, and STM.

None of the list digests will be inserted more than once (i.e., if list #N contains two kinds of matching elements its digest is inserted only once). Missing list digest is not replaced by any dummy data – it is skipped.

The obtained byte stream is then hashed and extended into PCR 18. The process is diagrammatically depicted below.

Figure 6. TPM 1.2 LCP Policy Authorities Data Stream



3.4.3.3 TPM 2.0 Effective LCP Policy Details

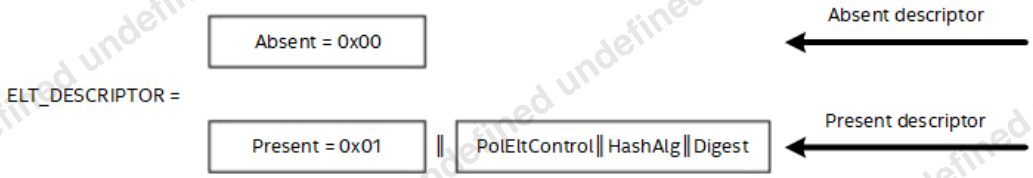
The effective LCP policy details hash represents all elements contributing to the currently established policy. Each of the elements is represented by a descriptor per element type. All descriptors are concatenated into a byte stream, which is then hashed using the rule described below and extended into PCR 17.

Format of the byte stream differs from the one used with TPM 1.2 since the size of the encountered digests may vary, and it is no longer possible to allocate a single fixed size for all descriptors.

The following solution addresses this issue:

Descriptors now have two forms: *Absent Descriptor*, and *Present Descriptor*. Difference between them is the value of *Existence Indicator*. If an element was matched during the enforcement scan, its *Present Descriptor* will be present in *EffLcpPolicyDetailsData.* If an element was not matched and was satisfied by default, its descriptor is reduced to just an *Absent Descriptor*. Form of descriptors is depicted below:

Figure 7. Element Descriptor



The descriptor contains: one byte of *Existence Indicator* concatenated with DWORD *PolEltControl,* WORD of *Hash Algorithm ID*, and variable size *Digest* corresponding to hash algorithm.

Formal definition of the structures is presented below:

```
#define ELT_IND UINT8
#define POL_CONTROL UINT32

typedef struct {
    ELT_IND    EltIndicator; // Boolean presence indicator ==
"1"
    POL_CONTROL   PolControl;
    TPMT_HA    EffEltDigest; // Standard TPM 2.0 library
structure
} EFF_ELT_PRESENT_DESCRIPTOR;

typedef struct {
    ELT_IND    EltIndicator; // Boolean presence indicator ==
"0"
} EFF_ELT_ABSENT_DESCRIPTOR;
typedef union {
    EFF_ELT_PRESENT_DESCRIPTOR PresDescr;
    EFF_ELT_ABSENT_DESCRIPTOR  AbsDescr;
}EFF_ELT_DESCRIPTOR, E_E_D;
```

EltIndicator is a byte size Boolean value initialized to "1" to indicate that given element type has contributed to the established policy. In this case it is followed by details of the element satisfying the policy. *EltIndicator* is initialized to "0" to indicate that the given element type was not found in policy files and was satisfied by default.

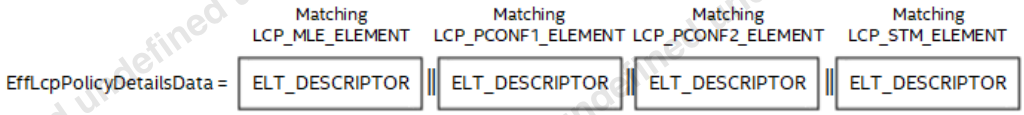
PolControl is *PolEltControl* field of the element satisfying the policy.

EffEltDigest is a standard TPM 2.0 library structure describing the matching digest value of the element satisfying the policy. It contains *HashAlgId* and a digest value.

For each of the elements involved, this last structure will be initialized to {1, *EffPolicyControl*, *EffEltDigest*} if matched, or {0} otherwise.

After all matching element descriptors are constructed, the overall form of *Effective LCP Policy Details Data* (*EffLcpPolicyDetailsData*) is depicted below:

Figure 8. TPM 2.0 LCP Policy Details Data Stream



LCP_PCONF1_ELEMENT and LCP_PCONF2_ELEMENT *Descriptors* represent first and second instance of PCONF element provided that *PO.PolicyControl.PCONF_Enforced* flag is set per Appendix J.2.3.

If LCP_PCONF2_ELEMENT is not available for any reason, it must be represented by an *Absent Descriptor*.

Above description can be formally expressed as:

$$EffLcpPolicyDetailsData = E\_E\_D_{MLE} \mid E\_E\_D_{PCONF1} \mid E\_E\_D_{PCONF2} \mid E\_E\_D_{STM}$$

Effective LCP Policy Details Digest is computed as:

$$LcpEffPolicyDetails = HASH_{HashAlgId}(EffLcpPolicyDetailsData)$$

Here, the hashing algorithm is determined by a PCR bank being extended (i.e., *HashAlgId* is the Hash Algorithm ID of extended PCR bank).

Finally, PCR 17 will be extended as:

```
PCR 17 = TPM2_Extend (LcpEffPolicyDetails, HashAlgId)
```

If LCP policy was “ANY”, a single byte of “0” will be measured into PCR 17 instead of using a relevant PCR bank’s HashAlgId.

3.4.3.4 TPM 2.0 Effective LCP Policy Authorities

The Effective LCP Authorities Hash provides aggregated information about all authorities (policy lists) contributing to the currently established policy. It will be computed as follows: each authority will be described using the LIST_SIGN_DSCR structure defined below; all authority descriptors will be concatenated into a byte

stream; that byte stream will be hashed using the rule described below and extended into PCR 18.

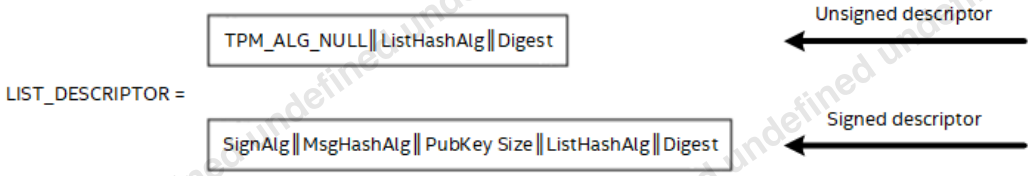
By the same reason as with *Effective LCP Policy Details Data* - since size occupied by each of the digests is variable, it is no longer possible to allocate fixed size for each of the *Descriptors*.

Solution is analogous to the one used for *Effective LCP Policy Details*:

Descriptors have two forms: *Signed Descriptor*, and *Unsigned Descriptor*. Difference between them is depicted below. Unsigned descriptor is required to have *SignAlg* hardcoded to be TPM_ALG_NULL. It is also missing all signature attributes.

The same rule as for TPM1.2 applies here: if more than one matching element is found in the *same* list, *List's Descriptor* is inserted only *once*. Missing *List Descriptor* is not replaced by any dummy data – it is skipped.

Figure 9. List Descriptor



Each signed list will be represented using the following structure:

```
typedef struct {
    UINT16  SignAlg;
    UINT16  HashAlg;
    UINT16  PubKeySize;
    TPMT_HA EffAuthDigest;
} LIST_SIGN_DSCR;
```

SignAlg is one of the supported signature algorithms, either TPM_ALG_RSASSA, TPM_ALG_RSAPSS, TPM_ALG_ECDSA or SM2.

HashAlg is the algorithm paired with the signature algorithm used to compute message digest.

PubKeySize is the public key size in bytes.

EffAuthDigest is a standard TPM2.0 library structure describing the digest of the public key of this authority. It includes the hash algorithm ID used to hash this Authority, concatenated with the digest value of the public key field used for the list's signature in memory.

- For RSASSA/RSAPSS signatures, the *PubKeyValue*[*PubkeySize*] field of LCP_RSA_SIGNATURE is hashed using the *HashAlg* specified in the PO NV Index. The size of the hashed data is *PubkeySize*.

- For SM2/ECDSA signatures, the *Qx*[*PubkeySize*] and *Qy*[*Pubkeysize*] components of *LCP_ECC_SIGNATURE* are hashed using the *HashAlg* specified in the PO TPM NV index. The size of the hashed data in *PubkeySize* * 2.

Each unsigned list will be represented using the following structure:

```
typedef struct {
    UINT16  SignAlg;
    TPMT_HA EffAuthDigest;
} LIST_UNSIGN_DSCR;
```

SignAlg is TPM_ALG_NULL.

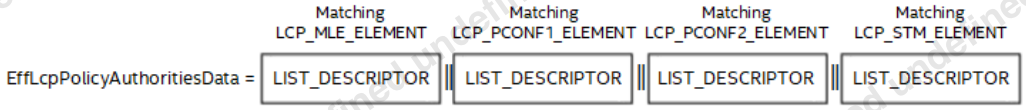
EffAuthDigest is a standard TPM 2.0 library structure describing the digest of the entire unsigned list. The entire list will be hashed using the *HashAlg* in the PO TPM NV Index. The size of the hashed data is the size of the unsigned list itself.

Any list will be represented by the following structure:

```
typedef union {
    LIST_SIGN_DSCR   SignedList;
    LIST_UNSIGN_DSCR UnsignedList;
} EFF_LIST_DSCR, E_L_D;
```

After all matching list descriptors are constructed, the overall form of *Effective LCP Policy Authorities Data* (*EffLcpPolicyAuthoritiesData*) is depicted below. Concatenation must be exactly in the shown order.

Figure 10. TPM 1.2 LCP Policy Authorities Data Stream



The same important note as to TPM 1.2 case – if more than one matching element is found in the *same* list, list's digest inserted only *once.* Missing list digest is not replaced by zero digest – it is skipped.

Above description can be formally expressed as:

$$EffLcpPolicyAuthoritiesData = E\_L\_D_{MLE} \mid E\_L\_D_{PCONF1} \mid E\_L\_D_{PCONF2} \mid E\_L\_D_{STM}$$

Note: Authorities providing a match of PCONF elements *Policy Data File* are included individually. This is done to cover the case when two PCONF elements are required to match – see *PO.PolicyControl.Pconf_Enforced* bit description in Appendix J.2.3

All signed LCP lists in each of the LCP *Policy Data Files* must use unique signing keys. If authority needs to supply more than one signed LCP list, it must maintain unique signing key for each of the lists.

Effective LCP Policy Authorities Digest then is computed as:

$$LcpEffPolicyAuhorities = HASH_{HashAlgId}(EffLcpPolicyAuthoritiesData)$$

Here used hashing algorithm is determined by PCR bank being extended (i.e., *HashAlgId* is the Hash Algorithm ID of extended PCR bank).

Finally, PCR 18 will be extended as:

```
PCR 18 = TPM2_Extend (LcpEffPolicyAuhorities, HashAlgId)
```

If LCP policy is evaluated to ANY, single byte with value 0 will be measured to PCR18 instead using relevant PCR bank *HashAlgId*.

3.4.4 Effective TPM NV info Hash

TPM 2.0 mode relevant only

For an attester, evaluating the trustworthiness of a platform/environment, the properties of the TPM NV indices used by Intel® TXT are of considerable interest. Therefore, SINIT will extend these properties into PCRs 17 and 18 and log the extension process into the event log in a form that facilitates inspection.

TPMS_NV_PUBLIC structure is used as the descriptor of TPM NV index properties. Based on that, the following structures are built:

```
#define IDX_IND UINT8

typedef struct {
    IDX_IND          IndexIndicator;
    TPMS_NV_PUBLIC   IndexProperties;
} EFF_IDX_PRESENT_DESCRIPTOR;
typedef struct {
    IDX_IND       IndexIndicator;
} EFF_IDX_ABSENT_DESCRIPTOR;
```

IndexIndicator is a Boolean byte size value initialized to “1” to indicate that index has been provisioned and to “0” if not.

IndexProperties is a standard TPM 2.0 library structure fully describing index properties.

```
typedef union {
    EFF_IDX_PRESENT_DESCRIPTOR PresDescr;
    EFF_IDX_ABSENT_DESCRIPTOR  AbsDescr;
} EFF_IDX_DESCRIPTOR, E_I_D;
```

For each of the indices:

- If present, its E_I_D = {1, PresDescr}
- If absent, its E_I_D = {0}

The data to be hashed will be:

$LcpEffNvInfoData = E\_I\_D_{AUX} \mid E\_I\_D_{PO}$

Where $E\_I\_D_{AUX}$ and $E\_I\_D_{PO}$ are descriptors of AUX and PO index, respectively.

Note: SGX index is intended for BIOS use and its descriptor is not included in the above data stream.

The data extended into PCR 17 and 18 will be:

$LcpEffNvInfoHash = HASH_{HashAlgId}\ (LcpEffNvInfoData)$

3.5 TPM NV RAM

As mentioned in Section 1.8 Intel® TXT and its CBnT version intensively uses TPM shielded storage to pass required information from Startup ACM to SINIT and to house Launch Control Policy data. Two indices AUX and PO are allocated for these purposes.

Usage of indices in TPM 1.2 and TPM 2.0 modes is analogous, but attributes, sizes and layout differ. Properties and owning authorities change as well, as described below. Complete index provisioning data can be found in Table 33 and Table 34

3.5.1 Auxiliary Index

TPM 1.2

The following are main properties of a TPM 1.2 mode Auxiliary (AUX) index:

It is defined with 'D' bit set, which means that index is defined for a lifetime of a platform and cannot be deleted.

Content of an index is readable by any SW at any locality, but writable only via TPM locality 3 (i.e., only by an ACM)

Its layout is described in Appendix D.5 and attributes in Table 33

TPM 2.0

TPM 2.0 mode AUX index features properties analogous to its TPM 1.2 mode counterpart but also has notable differences:

Like the TPM 1.2 mode index, its content is readable by any SW at any locality and writable only via locality 3 (i.e., ACM only).

Its layout is similar and thus described by the same Appendix D.5

Its size is larger to accommodate larger digests produced by SHA256 and SHA384 algorithms.

The biggest difference from its TPM 1.2 counterpart is an option to delete this index under Platform Supplier's policy – details can be found in Table 34 and notes that follow this table.

Another important difference is associated with a *nameAlg* – one of the major index parameters defining strength of data protection. Full discussion is in Section 3.5.3 below.

3.5.2 Platform Owner Index

Platform Owner (PO) index purpose is to house Platform Owner's LCP Policy. Full discussion of and index layout can be found in Section 3.2.1

Despite that TPM 1.2 and TPM 2.0 properties are very similar, notable differences still exist:

TPM 1.2

PO index is controlled by Platform Owner's credentials. It can be read by any SW but can be written / deleted by a Platform Owner only. All reads and writes are possible at any locality.

Complete index provisioning data can be found in Table 33

The size of PO index is fixed, and its full layout can be found in Appendix D.1.2

TPM 2.0

In the TPM 2.0 mode, the PO index is controlled by Platform Owner's credentials. It also can be read by any SW and written / deleted by a Platform Owner only.

Index provisioning data are specified in Table 34

Index size can vary (but can also be fixed at maximum value) based on the sizes of the digests stored in the *PolicyHash* area (see Appendix D.1.3).

Index contains two new fields *LcpSignAlgMask* and *LcpHashAlgMask* also described in Appendix D.1.3

Like TPM 2.0 mode AUX index, it requires careful selection of *nameAlg* – *see* Section 3.5.3 below.

3.5.3 nameAlg Support

When defining a TPM NV index, the *nameAlg* parameter specifies the Hashing algorithm used by the policy controlling that index and therefore the cryptographic strength used to protect that index's data.

There must be constraints on what *nameAlg* may be used for TPM NV indices. The Platform Owner is free to select one algorithm out of those supported by the TPM (*TPM_HashAlgIDList* from Appendix G.3) that meets cryptographic strength constraints for its intended use.

PO or AUX indices cannot contain any settings controlling the strength of their own protection, as these would be self-referential and impossible to configure initially. An insufficiently strong *nameAlg* may lead to insufficient policy protection of index access, making the contents of such index vulnerable.

ACMs will be built requiring a minimum bit-length for allowed hash algorithms. For example, if this built-in minimum requirement is 256 bits, then SHA256 and SM3 satisfy it.

The *nameAlg* parameter for AUX and PO indices must meet this requirement, or the ACM will generate a reset. The exception to this is when the *nameAlg* is the only supported algorithm on the platform, even if it does not meet the minimum.

Note that when the *nameAlg* for an index meets the minimum requirement, algorithms used within the index need not; that index's contents will be trusted, irrespective of the strength of internal protections for contained items.

3.6 PCR Extend Policy

TPM 2.0 mode relevant only

TPM 2.0 family of devices support different sets of commands to extend measurements into PCR banks:

- The *Extend* command allows to extend already computed digest(s) into one or several selected PCR banks;
- The *Event* and *Event Sequence* commands allow sending data stream to TPM which will then compute digests and extend them into all existing banks of PCRs at once.

Since expected performance of these sets of commands is different, ACM will support *Extend Policy* prescribing set of commands used to extend PCRs

Two policy settings will be supported:

1 Maximum Agility (MA). When this policy is selected ACM will extend PCRs using commands TPM2_PCR_Event; TPM2_HashSequenceStart; TPM2_HashUpdate;

TPM2_EventSequenceComplete. These commands will extend all existing PCR banks at the expense of possible performance loss.

2 Maximum Performance (MP). When this policy is selected ACM will use embedded SW to compute hashes and then will use TPM2_PCR_Extend commands to extend them into PCRs. If PCRs utilizing hash algorithms not supported by SW are discovered, they will be capped with "1" value. This policy will ensure maximum possible performance at the expense of possible capping of some of the PCRs.

PCR Extend Policy is delivered to SINIT via OS to SINIT Data heap table – see Appendix C.4

3.6.1 SINIT Policy Selection

ISVs supplying MLE solutions can retrieve information of SINIT supported embedded SW algorithms from TPM Info List Table (see Table 15) and examine capabilities of installed TPM to make a comprehensive choice.

Currently MLE and SINIT operations are not perceived to be time critical and therefore MA *Extend Policy* setting is envisioned to be suitable in all cases and can be statically chosen.

Nevertheless, possibility to apply installation time logic based on discovered capabilities remains a possibility for MLE developers. If all TPM PCR banks are supported by SINIT embedded SW, MP *Extend Policy* can be chosen. Otherwise MA Extend Policy must be a choice. Chosen policy may be delivered to MLE via configuration setting.

3.7 Revocation

3.7.1 SINIT Revocation

LCP also enables a limited self-revocation mechanism for SINIT. SINIT itself enforces this on launch and a failure (i.e., the executing SINIT was revoked) results in a TXT reset with an error code stored in the TXT.ERRORCODE register. The algorithm or process that SINIT uses to determine whether it has been revoked is described in the *SinitMinVersion* field in Section 3.2.1. The rationale for this decision-making process follows.

The ACM Info Table of a SINIT module contains that module's version number, per Table 10. The *SinitMinVersion* field of the PO Index defines the minimum version of a SINIT module that is allowed to execute to completion. SINIT verifies that its version meets that required and performs a self-revocation (via LT-RESET) if it does not. This mechanism allows the Platform Owners to restrict execution of SINIT versions prior to one they specify.

The upper boundary of *SinitMinVersion* field of the PO Index might need to be controlled too. The case when it might be needed is when Platform Owner wants to prevent SINIT update as part of the OS update push.

The *MaxSinitMinVersion* field in the PO Index allows the platform owner to control such *SinitMinVersion* update. Unless this field is set to 0xFF, the value given is the maximum value to which *SinitMinVersion* may be set to by a revocation ACM.

A SINIT version satisfying a platform *SinitMinVersion* may be found to have a security flaw resulting in platform vulnerabilities affecting *all* released platforms and all of the Platform Owners. An orthogonal self-revocation mechanism is provided for this case. In addition to its version number, each SINIT has a *security version number* (SVN). The platform AUX Index contains a reference value this field can be verified against. If the SVN is below the required version, SINIT performs a self-revocation as above.

This AUX Index field is only updatable from locality 3, and hence can only be modified by a special “revocation” ACM or by adding of respected functionality to select set of BIOS / Startup ACM functions (called collectively “revocation module”). Such revocation module would increase the AUX Index SVN requirement to a new target value, precluding execution of SINIT modules with lower SVN values, effectively revoking them.

§

4.0 Development and Deployment Considerations

4.1 Launch Control Policy Creation

Depending on the usage model, it may be desirable to create a Launch Control Policy at the time the MLE is built. This would apply in the cases where only one MLE is expected to run on the system. In such cases, the policy can be pre-created and provisioned during the installation of the MLE.

If multiple MLEs are expected to run on the system, or if there is to be a platform configuration policy, then it is likely that the policy will need to be created at the time of deployment. As each element type is allowed in a policy list, each MLE will have its own list if any of its list elements must differ from those of others.

In either case, it is advisable that the policy should contain a *SinitMinVersion* value that corresponds to the lowest versioned SINIT that is required. In the case of a system that supports multiple MLEs, a different *SinitMinVersion* may be specified with each MLE's policy element. The effective *SinitMinVersion* value will be the highest of the values in the PO policy, and matching MLE element (for each one that exists). This effective *SinitMinVersion* will be compared to the *AcmVersion* of the SINIT being used. Setting the *SinitMinVersion* value in the policy prevents an attacker from substituting an older version of SINIT (if there is one for a given platform) that may have security issues.

4.2 Launch Errors and Remediation

If there is an error during a measured launch, the platform will be reset, and an error code will be coded in the TXT.ERRORCODE register. It is important for MLE vendors to consider how their software will handle such errors and allow users or administrators to remediate them.

If an MLE is launched automatically either as part of the boot process or as part of an operating system's launch process, it needs to be able to detect a previous failure in order to prevent a continuous cycle of boot failures. Such failures may occur as part of the loading and preparation for the GETSEC[SENTER] instruction or they may occur during the processing of that instruction before the MLE is given control.

In the former case, it is the MLE launching software that is detecting the error condition (e.g., a mismatched SINIT ACM, TXT not being enabled, etc.) and that software can use whatever mechanism it chooses to persist the error or to handle it at that time. In the latter case, the system will be reset before the MLE launching software can handle the error so that software should be able to detect the error in the TXT.ERRORCODE register and take appropriate action.

What remediation action will be required depends on the error itself. If the MLE launch happens early in the boot process, the launching software may need a way of booting into a remediation operating system. If the launch happens within an operating system environment, the software may be able to remediate errors in that environment.

4.3 Determining Trust

While a TXT Launch Control Policy can be used to prevent software use of TPM locality 2 and access to TXT private space, it is not a general mechanism to prevent unwanted software from executing on a system. Consequently, an MLE cannot itself determine whether it is running in a TXT measured environment (it could be running on an emulator that spoofs PCR values, chipset registers, etc.).

In order to gain trust in an MLE (or rather, in software that uses TXT with the intention of being an MLE), the PCR values for the MLE (PCRs 17 and 18) must be used to make the trust "decision". The trust "decision" must either be made by a party external to the MLE's system (i.e., remote attestation) or by the release of some data that is not available to untrusted software (i.e., local attestation).

Remote attestation involves a remote party requesting the MLE to provide a TPM quote of the PCRs needed to determine trust (at least 17 and 18) and that remote party verifying the quote and making a trust determination of the PCR values. The remote party can then act on the trust level in various ways (disconnect the MLE system from the network, not provide it with network credentials, etc.). The details involved in the remote attestation process can be found at the Trusted Computing Group's (TCG) website (http://www.trustedcomputinggroup.org/).

Local attestation, also known as *SEALing*, uses the TPM to encrypt some data bound to certain PCR values (and/or locality). The data is typically SEALed by the MLE system when the system is in a trusted state (there are multiple ways to establish an initial trusted state) and the PCR/binding made to values that represent the desired trusted state (the initial and final states don't have to be the same as long as they are both trusted). The SEALed data is then persistently stored, so it can be retrieved and UNSEALed when the trusted MLE is running. The specifics of SEALing can also be found on the TCG website.

4.3.1 Migration of SEALed Data

If SEALing/local attestation is used to protect data, then the MLE must be able to accommodate the upgrading/changing of components whose measurements are in the PCRs being SEALed to. Since PCRs 17 and 18 contain the TCB for the MLE, at a minimum this would include SINIT, LCP, the MLE, etc. (see Section 1.10 for the complete contents of these PCRs). If data is SEALed to additional PCRs then changes to the entities that are measured into these other PCRs must also be handled. While data may be sealed to PCRs, locality, or an auth value, or any combination thereof, migration is only an issue when PCRs are sealed to.

When one of the elements of the SEALed data's PCRs is changed, the TPM will no longer UNSEAL that data. So, if no migration or backup of the plaintext data is made, then after the next measured launch that data will not be available to the new MLE. And unless the original MLE can be re-launched, the data will be lost. Thus, some provision for making the data available to the new MLE must be made while the data is still available in plaintext form (UNSEALed) (i.e., in the original MLE).

The most seamless and secure method for migrating the data to the new environment is for the original environment to re-SEAL the data to the new environment. This requires the original environment to calculate what the PCR values will be in the new environment. For PCRs 17 and 18, this can be done using the information included in Section 1.10, coupled with knowing or being able to calculate the constituent values for the new components. Alternately, if the new environment were run on a trusted system (so that nothing would tamper with the measurements), the PCR values could then be collected from that system and used directly as the new values without having to calculate them from the components.

Other methods, such as password-based recovery, key escrow, etc. would be the same as for any other encrypted data and have similar tradeoffs.

4.4 Deployment

4.4.1 LCP Provisioning

4.4.1.1 TPM Ownership

In the following discussion term "ownership" shall be understood broadly as a set of credentials allowing owners to control TPM. This is real ownership for TPM 1.2 family of devices and set of owner policies and authentication values for TPM 2.0 devices.

Because creation and writing to the Platform Owner policy TPM NV index requires the TPM owner authorization credential, the installation program should accommodate differing IT policies for how and when TPM ownership is established. In some enterprises, IT may take ownership before a system is deployed to an end user. In others, the TPM may be un-owned until the first application that requires ownership establishes it.

Since the TPM owner authorization credential will be required to modify the Platform Owner policy, if the installation program creates the credential it should provide a mechanism for securely saving that credential either locally or remotely.

4.4.1.2 Policy Provisioning

The Platform Owner policy TPM NV index will need to be created by the MLE installation program (or other TPM management software) if does not already exist.

This can be done with the TPM_NV_DefineSpace or TPM2_NV_DefineSpace command or corresponding higher-level TPM interface (e.g., via a TCG Software Stack or TSS).

Once the index has been created, the installation program can write the policy into the index using the TPM_NV_WriteValue or TPM2_NV_Write command or corresponding higher-level TPM interface (e.g., via a TCG Software Stack or TSS).

Because creating and writing to the Platform Owner policy index requires the TPM owner authorization credential, care should be taken to protect the credential when it is being used and to erase or delete it from memory as soon as it is no longer needed.

Ideally, policy provisioning would occur in a secure environment or be performed by an agent that can be verified as trustworthy. An example of the former would be on an isolated network immediately after receiving the system. Another would be booting from a CD containing provisioning software. An example of the latter would be to use Intel® TXT to launch a provisioning MLE or agent that was then attested to by a remote entity which could provide the owner authorization credential upon successful attestation.

4.4.2 SINIT Selection

Because the SINIT AC module is specific to a chipset, different platforms may have different SINIT ACMs. If an MLE is intended to run on multiple platforms with different chipsets, the MLE installation program will need to determine which SINIT ACM to install if the platform BIOS does not provide the SINIT ACM or if the MLE wants to use a later version.

Comparing the chipset compatibility information in the SINIT ACM's Chipset ID List with the corresponding information for the platform accomplishes this. This would be identical to the process for verifying SINIT ACM compatibility at launch time, as described in Section 2.2.3.1.

4.5 SGX Requirement for TXT Platform

Software Guard Extensions (SGX) is a set of instructions and mechanisms for memory accesses added to Intel® Architecture processors. These extensions allow an application to instantiate a protected container, referred to as an enclave. An enclave is a protected area in the application's address space, which provides confidentiality and integrity even in the presence of privileged malware. Accesses to the enclave memory area from any software not resident in the enclave are prevented. For more details about SGX, please refer to *Intel SGX Programming Reference Guide.*

Since TXT ACMs are in Trusted Computing Base (TCB) of SGX, SGX SW must be made aware of the SGX Security Version Numbers (SGX SVN) of all ACMs in the system. Passing of ACM SGX SVNs to SGX SW is performed via dedicated BIOS_SE_SVN MSR programming of which is the BIOS responsibility. This BIOS task is trivial if all ACMs in

the system are carried in BIOS flash but presents a special challenge to client platforms carrying SINIT ACM on HDD where it is not available for BIOS examination.

In order to avoid ungraceful resets resulted from mismatch of expected and actual SVN (security version number) of SINIT ACM during launch of SGX and TXT, requirements of SGX, TXT and BIOS interface developed for client platforms should be followed.

The interface allows the TXT runtime software to convey to BIOS the value of SINIT SVN to be used during next POST. In worst case, one additional system reboot after update of SINIT in TXT software stack is required.

The Interface is based on a mailbox mechanism implemented as SGX TPM NV index with unrestricted read and write capabilities. TXT runtime software will write into this index with SGX SVN discovered in SINIT ACM and BIOS will read this index and program into the SINIT_SVN field of BIOS_SE_SVN MSR.

SGX TPM NV Index is mandatory on the client platforms supporting TXT. If this index doesn't exist, SINIT_SVN field of BIOS_SE_SVN MSR may remain at default 0xFF value or be arbitrarily chosen by BIOS and this might cause TXT shutdown when GETSEC[SENTER] is executed after first non-faulty SGX instruction. The Index size is 8 bytes.

Details of SGX index definition for TPM 1.2 and TPM 2.0 platforms can be found in Table 33 and Table 34 respectively.

The following must be noted for TPM 2.0 index definition:

- *authValue* of index by convention must remain at default value "empty buffer". This will enable unrestricted access by any agent from any locality.
- Deletion of the SGX index is possible only in TPM 2.0 and can only be performed under control of the OEM policy – analogously to other TPM 2.0 TXT indices.

Table 6 diagrammatically illustrates the SGX index structure.

Table 6. SGX Index Content

Bytes 1 - 7. Reserved, must be zero.	Byte 0. SGX SINIT_SVN. Saved by MLE of specially defined tool. Must be initialized to default value "1"

Table 7 shows content of IA32_SE_SVN_STATUS MSR used by software components. It is a read-only MSR.

Table 7. IA32_SE_SVN_STATUS MSR (0x500)

Bits	Name	Comment
0	Lock	0 – BIOS_SE_SVN MSR can be floated down. Launching a properly signed ACM will not lead to LT shutdown irrespective of its SE SVN 1 - BIOS_SE_SVN MSR is locked. Launching of ACM with SGX SVN lower than value of SINIT_SVN field will LT reset platform.
15:1	Reserved	
23:16	SINIT_SVN	Reflect values of bits 23:16 of BIOS_SE_SVN MSR (SINIT_SVN) on CPUs that enumerate CPUID.feature_flags.SMX as 1. On CPUs that enumerate CPUID.feature_flags.SMX as 0, these bits are reserved (0).
63:24	reserved	

Line 1: SINIT may be present in BIOS flash as is practice for server platforms, and workstations. In this case BIOS is responsible for finding the SINIT module, decompressing it and placing it in heap SINIT memory. It also has to program *BiosSinitSize* field in the BIOS Data table – see Table 20.

Lines 2, 3: If SINIT is present in BIOS flash as signaled by value of *BiosSinitSize* field, MLE may exit this flow since BIOS has all necessary information to program *BIOS_SE_SVN MSR. SGX* index may not exist and is ignored if it exists. MLE shall not generate any SGX index related errors.

Lines 4, 5: MLE shall exit this flow if SGX is not supported by CPU

Line 7: SGX SVN value is in SINIT header in a word at offset 0x1E

Lines 8, 9: Architectural *IA32_SE_SVN_STATUS* MSR carries the same SINIT SGX information which is programmed into *BIOS_SE_SVN* MSR. MLE shall avoid accessing non-architectural *BIOS_SE_SVN* MSR. It shall retrieve programmed SINIT SGX SVN value from *IA32_SE_SVN_STATUS* MSR instead. If value retrieved from *IA32_SE_SVN_STATUS* MSR differs from SINIT SGX SVN, value of SGX index must be updated and system reset to let BIOS use the updated information.

Lines 10, 11: If BIT 0 of *IA32_SE_SVN_STATUS* MSR is “0” SGX is not active and MLE may continue to launch SINIT. If BIT 0 of *IA32_SE_SVN_STATUS* MSR is “1” SGX is active and attempt to launch SINIT will lead to ungraceful platform reset. MLE shall avoid it by one of the possible actions: It may post message to user and reset platform after a short delay to let BIOS use correct SINIT SGX SVN number at next boot. Alternatively, it may post message to user and let him reset platform manually at a proper time.

Listing 9. SGX Support on TXT Platform Pseudo code

```
1.  Find appropriate SINIT for platform.
2.  If SINIT is already present in SINIT memory
3.      Exit this flow
//
// Enumerate SGX support. Analyze CPUID.feature_flags.SE bit
// (leaf 7, sub-leaf 0, EBX.bit2).
//
4.  If CPUID.feature_flags.SE bit is not set {
5.      Exit this flow
6.  }
7.  Read SINIT_SVN 16-bit value from SINIT header offset 0x1E
8.  If IA32_SE_SVN_STATUS[23:16] != SINIT_SVN {
//
// Write SINIT_SVN value from ACM header into SGX TPM NV index
//
9.      SGX TPM NV index ← SINIT_SVN
//
// Read IA32_SE_SVN_STATUS MSR, lock bit (Bit 0)
//
10.     If IA32_SE_SVN_STATUS MSR[0] == 1 { // SGX is active
11.         Reset platform
12.     }
13. }
```

4.6 Converged BtG/TXT impact on TXT Platform

Both Server TXT and Boot Guard (BtG) technologies require a Startup ACM to be executed at platform reset. Intel® CPUs can support only a single such ACM and therefore combining a BtG ACM with a Startup ACM is inevitable for platforms supporting both technologies. This combining requirement triggered the whole set of upgrades targeted to better align both technologies, and their mutual benefits.

Short list of the most fundamental changes follows. Complete information can be found in *"Converged Boot Guard and Intel® Trusted Execution Technologies. BIOS Specification"*.

1. BtG, Startup and BIOS ACMs are combined into a single binary referred to as Startup ACM (S-ACM). The Startup function of the S-ACM performs all steps required by BtG and TXT.
2. Client and Server TXT flavors are unified. Client reset attack mitigation was changed to be carried out by BIOS and not by the SCLEAN function.
3. Information about OEM-established policies consumed by the Startup function is delivered using a BtG Key and Boot Policy manifests (KM and BPM). Delivering information using FIT table records was eliminated.
4. KM and BPM are included into MLE TCB and are measured into dynamic PCRs
5. Platform Supplier LCP is removed. Instead TXT uses results of BtG BIOS verifications that has been found to be analogous to "Signed BIOS" LCP policy. As a

result, the PS index, its associated Policy Data File and SBIOS element are eliminated.

6. Supported cryptography has been reduced per recommendation of "*CNSS Advisory Memorandum 02-15"*

Majority of these changes occur in pre-boot space and are of little interest for MLE but some of the changes impact MLE and other may be of interest for future updates. The following list contains all MLE-visible changes resulted from conversion, used or not:

1. Platform Supplier's policy is now handled by BtG. Respectively no PS LCP data file will be present, and no PS index will be provisioned. Therefore no "merging" of PS and PO policies is needed, and overall LCP handling is simplified.
2. PO index content is simplified:
   MaxBiosacMinVersion field was removed since now S-ACM revocation is handled using BtG style;
   PolicyControl flags were essentially simplified – all of them are reserved except of NPW_OK and *Pconf_Enforced* – see Section 3.2.1.
   Consequently, version of PO index data structure changed to 2.4 and 3.2 respectively for TPM 1.2 and TPM 2.0
3. Functionality associated with *Pconf_Enforced* bit changes. Previously it required to satisfy PCONF element matches in both PS and PO LCP policies. Now it requires matching of PCONF elements in two consecutive policy lists. Complete explanation is in Appendix J.2.3
4. SBIOS element is removed and associated element types 2, 0x12 are reserved
5. New capability flag is defined in *"ACM Info Table"* to indicate ACM compliance to *"Converged BtG and TXT Technologies"* requirements. Similar flag is defined in the capabilities field of MLE header. Respectively, *"ACM Info Table"* version is incremented to 7 and MLE Header version is incremented to 2.2. See Table 4
6. BIOS data table no longer contains *LcpPdBase* and *LcpPdSize* fields since these fields were associated with currently removed PS LCP Data file – see Appendix C.2
7. *ProcessorSCRTMStatus* field of *SinitMleData* heap table will be set to "1" when with BtG/TXT PCR0 measurement is performed by Startup ACM function – see Appendix C.5
8. MLE will see new events logged into TXT event log. Full list of new events can be found in Table 29.
9. Three new registers have been defined in TXT device space: new ACM_POLICY_STATUS register at offset 0x378 has been defined to carry combination of BtG and TXT polies. MLE may benefit from it if needed;
   ACM_ERROR_STATUS register at offset 0x328 carries pre-boot error code information instead or TXT.ERRORCODE register (Layout of this register is not disclosed since it is of no value for OS). TXT.ERRORCODE register is left to exclusive use of SINIT and CPU uCode; BOOTSTATUS register at offset 0xA0 caries status flags of Startup function execution. Detailed information of layout of these registers

can be found in B and in *"Converged Boot Guard and Intel® Trusted Execution Technologies. BIOS Specification for Client CNL and Server ICX platforms"*.

10. TXT has been extended to handle ACMs with 3072-bit signatures and new crypto algorithms.

§

Appendix A Intel® TXT Execution Technology Authenticated Core Modules

A.1 Authenticated Code Module Format

An authenticated code module (AC module) is required to conform to a specific format. At the top level, the module is composed of three sections: module header, internal working scratch space, and user code and data. The module header contains critical information necessary for the processor to properly authenticate the entire module, including the encrypted signature and RSA-based public key. The processor also uses other fields of the AC module for initializing the remaining processor state after authentication.

The format of the authenticated-code module is in Table 8. This definition represents revisions 0.0 and 3.0 of the AC module header version (defined in the *HeaderVersion* field). ACMs with version 3.0 support converge of Boot Guard and TXT.

Table 8. Authenticated Code Module Format

Field	Offset version 0.0 / 3.0	Size (bytes) version 0.0 / 3.0	Description
ModuleType	0	2	2 = Module type
ModuleSubType	2	2	Module sub-type 0 – TXT ACM 1 – S-ACM
HeaderLen	4	4	Header length (in multiples of four bytes) 161 - for version 0.0 224 – for version 3.0
HeaderVersion	8	4	Module format version 0.0 – for SINIT ACM before 2017 3.0 – for SINIT ACM of converge of BtG and TXT
ChipsetID	12	2	Module release identifier
Flags	14	2	Module-specific flags
ModuleVendor	16	4	Module vendor identifier
Date	20	4	Creation date (BCD format: year.month.day)

Field	Offset version 0.0 / 3.0	Size (bytes) version 0.0 / 3.0	Description
Size	24	4	Module size (in multiples of four bytes)
TXT SVN	28	2	TXT Security Version Number
SE SVN	30	2	Software Guard Extensions (Secure Enclaves) Security Version Number
CodeControl	32	4	Authenticated code control flags
ErrorEntryPoint	36	4	Error response entry point offset (bytes)
GDTLimit	40	4	GDT limit (defines last byte of GDT)
GDTBasePtr	44	4	GDT base pointer offset (bytes)
SegSel	48	4	Segment selector initializer
EntryPoint	52	4	Authenticated code entry point offset (bytes)
Reserved2	56	64	Reserved for future extensions
KeySize	120	4	Module public key size less the exponent (in multiples of four bytes) 64 - for version 0.0 96 – for version 3.0
ScratchSize	124	4	Scratch field size (in multiples of four bytes) 143 = (2 * KeySize + 15) – for version 0.0 208 = (2 * KeySize + 16) – for version 3.0
RSAPubKey	128	256 / 384	KeySize * 4 = Module public key
RSAPubExp	384 / absent	4/0	Module public key exponent Present – for version 0.0 Absent – for version 3.0
RSASig	388 / 512	256 / 384	KeySize * 4 = PKCS #1.5 RSA Signature
End of AC module header			
Scratch	644 / 896	572 / 832	ScratchSize * 4 = Internal scratch area used during initialization (needs to be all 0s)
User Area	1216 / 1728	N * 64	User code/data (modulo-64-byte increments)

ModuleType

Indicates the module type. The following module types are defined:

2 = Chipset authenticated code module.

Only ModuleType 2 is supported by GETSEC functions SENTER and ENTERACCS.

ModuleSubType

Indicates whether the module is capable of being executed at processor reset.

0 = ACM cannot be executed at processor reset

1 = ACM is capable of being executed at processor reset

ModuleSubType 1 is not supported for use by the GETSEC[SENTER] instruction.

HeaderLen

Length of the authenticated module header specified in 32-bit quantities. The header spans the beginning of the module to the end of the signature field. This is fixed to 161 for AC module version 0.0 and 320 for AC module version 3.0.

HeaderVersion

Specifies the AC module header version. Major and minor vendor field are specified, with bits 15:0 holding the minor value and bits 31:16 holding the major value. This should be initialized to zero for header version 0.0 and 0300H for header version 3.0. The processor will reject unsupported header versions, resulting in an abort during authentication.

ChipsetID

Module-specific chipset identifier.

Flags

Module-specific flags. The following bits are currently defined:

Table 9. AC module Flags Description

Bit position	Description
13:0	Reserved (must be 0)
14	Production (0) or pre-production (1)
15	Production (0) or debug (1) signed

ModuleVendor

Module creator vendor ID. Use the PCI SIG* assignment for vendor IDs to define this field. The following vendor ID is currently recognized:

00008086H = Intel

Date

Creation date of the module. Encode this entry in the BCD format as follows: year.month.day with two bytes for the year, one byte for the day, and one byte for the month. For example, a value of 20131231H indicates module creation on December 31, 2013.

Size

Total size of module specified in 32-bit quantities. This includes the header, scratch area, user code and data.

TXT SVN

TXT Security Version Number

SE SVN

Software Guard Extensions (a.k.a. Secure Enclaves) Security Version Number

CodeControl

Authenticated code control word. Defines specific actions or properties for the authenticated code module.

ErrorEntryPoint

If bit 0 of the *CodeControl* word is 1, the processor will vector to this location if a snoop hit to a modified line was detected during the load of an authenticated code module. If bit 0 is 0, then enabled error reporting via bit 1 of a HITM during ACEA load will result in an abort of the authentication process and signaling of an Intel® Trusted Execution Technology shutdown condition.

GDTLimit

Limit of the GDT in bytes, pointed to by *GDTBasePtr*. This is loaded into the limit field of the GDTR upon successful authentication of the code module.

GDTBasePtr

Pointer to the GDT base. This is an offset from the authenticated code module base address.

SegSel

Segment selector for initializing CS, DS, SS, and ES of the processor after successful authentication. CS is initialized to SegSel while DS, SS, and ES are initialized to SegSel + 8.

EntryPoint

Entry point into the authenticated code module. This is an offset from the module base address. The processor begins execution from this point after successful authentication.

Reserved2

Reserved. Should contain zeros.

KeySize

Defines the width the RSA public key in dwords applied for authentication, less the size of the exponent. The information in this field is intended to support external software parsing of an AC module independent of the module version. It is the responsibility of the developer to reflect an accurate KeySize. This field is not checked for consistency by the processor.

ScratchSize

Defines the width of the scratch field size specified in 32-bit quantities.
For version 0.0 of the AC module header, *ScratchSize* is defined by *KeySize* * 2 + 15.
For version 3.0 it is *KeySize* * 2 + 16.
The information in this field is intended to support external software parsing of an AC module independent of the module version. It is the responsibility of software to reflect an accurate *ScratchSize*. The processor does not check this field.

RSAPubKey

Contains a public key modulus to be used for decrypting the signature of the module. The size of this field is defined by the previously defined AC module field, *KeySize**4.

RSAPubExp

Header 0.0: Contains standard public key exponent.
Header 3.0: Not present and is assumed to be standard value 0x10001

RSASig

The RSASSA PKCS #1.5 RSA Signature of the module. The RSA Signature signs an area that includes the some of the module header and the USER AREA data field (which represents the body of the module). Parts of the module header not included are the RSA Signature, public key, and scratch field.

Scratch

Used for temporary scratch storage by the processor during authentication. This area can be used by the user code during execution for data storage needs.

User Area

User code and data represented in modulo-64-byte increments. In addition, the boundary between data and code should be on at least modulo-1024-byte intervals. The user code and data region are allocated from the first byte after the end of the Scratch field to the end of the AC module.

The chipset AC module information table is located at the start of the User Area and contains supplementary information that is specific to chipset AC modules. The chipset ID list is described in more detail in Section 2.2.3.1.

Table 10. Chipset AC Module Information Table

Field	Offset (Bytes)	Width (Bytes)	Description

UUID	0	16	UUID of the Chipset AC module information table defined as:
			ULONG UUID0; // 0x7FC03AAA
			ULONG UUID1; // 0x18DB46A7
			ULONG UUID2; // 0x8F69AC2E
			ULONG UUID3; // 0x5A7F418D
			This UUID is used to identify a file/memory image as being a chipset AC module.

ChipsetACMType	16	1	Module type (00h = BIOS; 01h = SINIT) Bit 3, if set, flags the ACM as a revocation module, hence “8” is a BIOS revocation AC, “9” is an SINIT revocation AC.
Version	17	1	Version of this table. Table versions are always backwards compatible. The highest version defined is currently 7. Version 5 included all changes added to suport TPM 2.0 family. Version 6 is added to cover addition of ACM Revision field. Version 7 is added to cover addition of a new ACM capability in capabilities field: Converged BtG and TXT (CBnT) support
Length	18	2	Length of this table in bytes.
ChipsetIDList	20	4	Location of the Chipset ID list used to identify chipsets supported by this AC Module. This field is an offset in bytes from the start of the AC Module. See Table 11 for details.

Field	Offset (Bytes)	Width (Bytes)	Description
OsSinitDataVer	24	4	Indicates the maximum version number of the OS to SINIT data structure that this module supports. It is assumed that the module is backward compatible with previous versions.
MinMleHeaderVer	28	4	Indicates the minimum version number of the MLE Header data structure that this module supports/requires. MLEs with more recent header versions are responsible for determining whether they can support this version of the ACM.
Capabilities	32	4	Bit vector of supported capabilities. The values match those of the Capabilities field in the MLE header. This can be used by an MLE to determine whether the ACM is compatible with it and to determine any optional capabilities it might support. See Table 4.
AcmVersion	36	1	Version of this AC Module. It is compared against the *SinitMinVersion* field in LCP_POLICY(2) to determine if the module is revoked.
ACM Revision	37	3	ACM Revision in the format: <Major>.<Minor>.<Build> = XX.YY.ZZ where X, Y and Z are hexadecimal digits. Range for each of the fields is therefore 0 – FF. It is assumed that this revision will be added by BIOS to the list of records and will be examined by tools such as TXTInfo and Brand Verification.
ProcessorIDList	40	4	Location of the Intel® TXT Processor ID list used to identify Intel® TXT processors supported by this AC Module. This field is an offset in bytes from the start of the AC Module. See Table 13 for details.
Version >= 5			
TPMInfoList	44	4	Location of table of TPM capabilities supported by ACM. This field is an offset in bytes from ACM start.

Table 11. Chipset ID List

Field	Offset (Bytes)	Width (Bytes)	Description
Count	0	4	Number of entries in the array *ChipsetID*
ChipsetIDs[]	4	Count * sizeof (TXT_ACM_CHIPSET_ID)	An array of count entries of the structure TXT_ACM_CHIPSET_ID (see Table 12).

Table 12. TXT_ACM_CHIPSET_ID Format

Field	Offset (Bytes)	Width (Bytes)	Description
Flags	0	4	Set of flags to further describe functions of the chipset ID structure. Bit Description: [0]: *RevisionIdMask* – if 0, the *RevisionId* field must exactly match the TXT.DIDVID.RID field. If 1, the *RevisionId* field is a bitwise mask that can be used to test for any bits set in the TXT.DIDVID.RID field. If any bits are set, the *RevisionId* is a match. [31:1]: Reserved for future use. Must be 0.
VendorID	4	2	Indicates the chipset vendor this AC Module is designed to support. This field is compared against the TXT.DIDVID.VID field.
DeviceID	6	2	Indicates the chipset vendor's device that this AC Module is designed to support. This field is compared against the TXT.DIDVID.DID field.
RevisionID	8	2	Indicates the revision of the chipset vendor's device that this AC module is designed to support. This field is used according to the *RevisionIdMask* bit in the Flags field.
Reserved	10	6	Reserved for future use.

Table 13. Processor ID List

Field	Offset (Bytes)	Width (Bytes)	Description
Count	0	4	Number of entries in the array *ProcessorID*
ProcessorIDs[]	4	Count * sizeof (TXT_ACM_PROCESSOR_ID)	An array of count entries of the structure TXT_ACM_PROCESSOR_ID (see next table).

Table 14. TXT_ACM_PROCESSOR_ID Format

Field	Offset (Bytes)	Width (Bytes)	Description
FMS	0	4	Indicates the Family/Model/Stepping of the processor this AC Module is designed to support. This field is compared against the corresponding value returned from the CPUID instruction.
FMSMask	4	4	Mask to apply to FMS

Field	Offset (Bytes)	Width (Bytes)	Description
PlatformID	8	8	Indicates the Platform ID of the processor this AC Module is designed to support. This field is compared against the value in the IA32_PLATFORM_ID MSR.
PlatformMask	16	8	Mask to apply to Platform ID

Table 15. TPM Info List

Field	Offset (Bytes)	Width (Bytes)	Description
TPM Capabilities	0	4	TPM supported capabilities (described below)
Count	4	2	Number of entries in AlgortihmID[]
AlgorithmID[]	6	Count * UINT16	An array of "Count" entries of algorithm IDs supported by SINIT ACM per the TPM2 specification enumeration in Part 2, Table 7: TPM_ALG_RSA == 0x0001 TPM_ALG_SHA1 == 0x0004 TPM_ALG_SHA256 == 0x000B TPM_ALG_SM3_256 == 0x0012 Etc.

Table 16. TPM Capabilities Field

Bit Position	Description
1:0	TPM2 *PCR Extend Policy* Support = 00b: illegal = 01b: Maximum Agility Policy. Measurements are done using TPM PCR event and sequence commands when a PCR algorithm is not included in the embedded set of algorithms. = 10b: Maximum performance Policy. Measurements are done using embedded fixed set of algorithms and TPM PCR extend commands = 11b: Both policies types are supported
5:2	TPM family support = 0000b: illegal = 0001b: discrete TPM 1.2 supported = 0010b: discrete TPM 2.0 supported = 1000b: firmware TPM 2.0 supported Above settings can be combined to indicate multiple support options

Bit Position	Description
6	TPM NV index set supported: = 0: Initial TPM 2.0 TPM NV index set out of 0x180_xxxx and 0x140_xxxx ranges = 1: TCG compliant TPM 2.0 TPM NV index set out of Intel Reserved range of indices 0x1C1_0100 – 0x1C1_00x13F This bit will always be forced to "1"
31:7	Reserved, must be zero

Notes on TPM2 PCR Extend Policy:

TPM2 PCR Extended Policy Support is a field describing ACM policy regarding integrity collection commands used:

- Maximum Agility PCR Extend Policy: ACM can support algorithm agile commands TPM2_PCR_Event; TPM2_HashSequenceStart; TPM2_HashUpdate; TPM2_EventSequenceComplete. When this policy is selected, ACM will use the commands above if not all PCR algorithms are covered by embedded set of algorithms and will extend all existing PCR banks. Side effect of this policy is possible performance loss.
- Maximum Performance PCR Extend Policy: ACM can support several hash algorithms via embedded SW. When this policy is selected, ACM will use embedded SW to compute hashes and then will use TPM2_PCR_Extend commands to extend them into PCRs. If PCRs utilizing hash algorithms not supported by SW are discovered, they will be capped with "1" value. This policy, when selected, will ensure maximum possible performance but has side effect of possible capping of some of the PCRs.

For an ACM that supports both extend policies, it will use the one indicated in the flags field in the *OsSinitData* structure (Table 22).

Notes on TPM family:

0001b - Discrete TPM 1.2 supported: this includes all FIFO interfaces that are used with this family – TIS 1.21, and TIS 1.3

0010b - Discrete TPM 2.0 supported: this includes all interfaces that are supported with this family – TIS1.3, and PTP2.0.

0011b – Both combinations above are supported

1000b – TPM2.0 family is supported over CRB interface.

1010b – TPM2.0 family is supported over FIFO and CRB interfaces, etc.

4.6.1 Memory Type Cacheability Restrictions

Prior to launching the authenticated execution environment using the GETSEC leaf functions ENTERACCS or SENTER, processor MTRRs (Memory Type Range Registers) must first be initialized to map out the authenticated RAM addresses as WB (write-back). Failure to do so may affect the ability for the processor to maintain isolation of the loaded authenticated code module. The processor will signal an Intel® TXT shutdown condition with the *#BadACMMType* error code during the loading of the authenticated code module if non-WB memory is detected.

Note that despite that CPU enforces only 4KB SINIT alignment, such allocation may require too many MTRRs to provide the WB cache-ability. It is suggested then, that BIOS allocate SINIT memory on the 128KB boundary. This combined with specially selected SINIT sizes will allow using minimal number of MTRRs (up to 3).

While physical addresses within the load module must be mapped as WB, the memory type for locations outside of the module boundaries must be mapped to one of the supported memory types as returned by GETSEC[PARAMETERS] (or UC as default). This is required to support inter-operability across SMX capable processor implementations.

4.6.2 Authentication and Execution of AC Module

Authentication is performed after loading the code module into the authenticated code execution area. Information from the authenticated code module header is used to support the authentication process. The *RSAPubKey* header version 0.0 field contains a public key plus a 32-bit exponent (skipped in the header 3.0 and is assumed to be a standard one) used for decrypting the signature of the authenticated code module. The signature is held in encrypted form in the *RSASig* header field and it represents the PKCS #1.5 of RSA Signature of the module. The RSA Signature signs an area that includes the sum of the module header and the entire USER AREA data field, which represents the body of the module. Those parts of the module header not included are the RSA Signature, the public key, and the scratch field. An inconsistent authenticated code module format, inconsistent comparison of the public key hash, or mismatch of the decrypted signature against the computed hash of the authenticated module or a corrupted signature padding value results in an abort of the authentication process and signaling of an Intel® TXT shutdown condition. As part of the authentication step, the processor stores the decrypted signature of the AC module in the first 20 or 32 bytes (depending on the CPU family) of the 'Scratch' field of the AC module header.

After authentication has completed successfully, the private configuration space of the Intel® TXT-capable chipset is unlocked. At this point, only the authenticated code module or system software executing in authenticated code execution mode is allowed to gain access to the restricted chipset state for the purpose of securing the platform.

The architectural state of the processor is partially initialized from contents held in the header of the authenticated code module. The processor GDTR, CS, and DS selectors

are initialized from fields within the authenticated code module. Since the authenticated code module must be relocatable, all address references must be relative to the authenticated code module base address in EBX. The processor GDTR base value is initialized to the AC module header field GDT *BasePtr* + module base address held in EBX and the GDTR limit is set to the value in the GDT Limit field. The CS selector is initialized to the AC module header *SegSel* field, while the DS selector is initialized to CS + 8. The segment descriptor fields are implicitly initialized to BASE=0, LIMIT=FFFFFh, G=1, D=1, P=1, S=1, read/write access for DS, and execute/read access for CS. The processor begins the authenticated code module execution with the EIP set to the AC module header *EntryPoint* field + module base address (EBX). The AC module-based fields used for initializing the processor state are checked for consistency and any failure results in a shutdown condition.

§

Appendix B SMX Interaction with Platform

B.1 Intel® Trusted Execution Technology Configuration Registers

Intel® TXT configuration registers are a subset of chipset registers. These registers are mapped into two regions of memory, representing the public and private configuration spaces. Registers in the private space can only be accessed after a measured environment has been established and before the TXT.CMD.CLOSE-PRIVATE command has been issued. The private space registers are mapped to the address range starting at FED20000H. The public space registers are mapped to the address range starting at FED30000H and are available before, during and after a measured environment launch. All registers are defined as 64 bits and return 0's for the unimplemented bits. The offsets in the table are from the start of either the public or private spaces (all registers are available within both spaces, though with different permissions).

After writing to one of the command registers (e.g., TXT.CMD.SECRETS), software should read the corresponding status flag for that command (e.g., TXT.E2STS [SECRETS.STS]) to ensure that the command has completed successfully.

B.1.1 TXT.STS – Status

Description	This is the general status register. AC modules and the MLE use this read-only register to get the status of various Intel® TXT features.
Offset	000H
Pub Attribs Priv Attribs	RO RO

Bits	Field Name	Field Description
0	SENTER.DONE.STS	The chipset sets this bit when it sees all threads have done an TXT.CYC.SENTER-ACK. When any of the threads does the TXT.CYC.SEXIT-ACK the TXT.THREADS.JOIN and TXT.THREADS.EXISTS registers will not be equal, so the chipset will clear this bit.
1	SEXIT.DONE.STS	This bit is set when all bits in the TXT.THREADS.JOIN register are clear. Thus, this bit will be set immediately after reset (since the bits are all 0). Once all threads have done a TXT.CYC.SEXIT-ACK, the TXT.THREAD.JOIN register will be 0, so the chipset will set this bit.
5:2	Reserved	Reserved

Bits	Field Name	Field Description
6	MEM-CONFIG-LOCK.STS	This bit will be set to 1 when the memory configuration has been locked. Cleared by TXT.CMD.UNLOCK.MEMCONFIG or by a system reset.
7	PRIVATE-OPEN.STS	This bit will be set to 1 when TXT.CMD.OPEN-PRIVATE is performed. Cleared by TXT.CMD.CLOSE-PRIVATE or by a system reset.
14:8	Reserved	Reserved
15	TXT.LOCALITY1.OPEN.STS	This bit is set when the TXT.CMD.OPEN.LOCALITY1 command is seen by the chipset. It is cleared on reset or when TXT.CMD.CLOSE.LOCALITY1 is seen.
16	TXT.LOCALITY2.OPEN.STS	This bit is set when either the TXT.CMD.OPEN.LOCALITY2 command or the TXT.CMD.OPEN.PRIVATE is seen by the chipset. It is cleared on reset, when either TXT.CMD.CLOSE.LOCALITY2 or TXT.CMD.CLOSE.PRIVATE is seen, and by the GETSEC[SEXIT] instruction.
63:17	Reserved	Reserved

B.1.2 TXT.ESTS – Error Status

Description	This is the error status register that contains status information associated with various error conditions. The contents of this register are preserved across soft resets.
Offset	008H
Pub Attribs Priv Attribs	RO RO

Bits	Field Name	Field Description
0	TXT_RESET.STS	This bit is set to '1' to indicate that an event occurred which may prevent the proper use of TXT (possibly including a TXT reset). To maintain TXT integrity, while this bit is set a TXT measured environment cannot be established; consequently, Safer Mode Extension (SMX) instructions GETSEC[ENTERACCS] and GETSEC[SENTER] will fail. This bit is sticky and will only be cleared on a power cycle.
7:1	Reserved	Reserved

B.1.3 TXT.ERRORCODE – Error Code

Description	This register holds the Intel® TXT shutdown error code. A soft reset does not clear the contents of this register; a hard reset/power cycle will clear the contents. This was formerly labeled the TXT.CRASH register.
Offset	030H
Pub Attribs Priv Attribs	RO RW

Bits	Field Name	Field Description
3:0	Type2 /Module Type	0 = BIOS ACM 1= SINIT
9:4	Type2 / Class Code	0-0x3f = Class code clusters several congeneric errors into a group
14:10	Type2 /Major Error Code	0-0x1f = Error Code within current Class Code.
15	Software Source	0 = Authenticated Code Module 1 = MLE
27:16	Type1 /Minor Error Code	This Field value depends on Class Code and / or Major Error Code.
29:28	Type1/Reserved	This is implementation and source specific. Provides details on the failure condition.
30	Processor/Software	0 = Error condition reported by processor (see Table 17) 1 = Error condition reported by software
31	Valid/Invalid	0 = Register content invalid, the rest of the register contents should be ignored. 1 = Valid error

Note: Upon successful execution, SINIT will put 0xC0000001 in the register.

Note: The format of the Type field for errors reported by SINIT is defined in an errors text file included with each SINIT AC module. This file also includes the definition of the error codes produced by that version of SINIT. Error definitions which are stable across SINIT AC Modules can be found in Appendix H.

Table 17. Type Field Encodings for Processor-Initiated Intel® TXT Shutdowns

Type	Error condition	Mnemonic
0	Legacy shutdown	#LegacyShutdown
1–4	Reserved	Reserved
5	Load memory type error in Authenticated Code Execution Area	#BadACMMType
6	Unrecognized AC module format	#UnsupportedACM

Type	Error condition	Mnemonic
7	Failure to authenticate	#AuthenticateFail
8	Invalid AC module format	#BadACMFormat
9	Unexpected snoop hit detected	#UnexpectedHITM
10	Invalid event	#InvalidEvent Note 2
11	Invalid MLE JOIN format	#BadJOINFormat
12	Unrecoverable machine check condition	#UnrecovMCError
13	VMX abort error occurred	#VMXAbort
14	Authenticated code execution area corruption	#ACMCorrupt
15	Invalid voltage/bus ratio	#InvalidVIDBRatio
16–65535	Reserved	Reserved

Note: The conditions under which most of these errors are generated can be found in the pseudo code of the SMX instructions in Chapter 6, “Safer Mode Extensions Reference”, of the Intel 64 and IA-32 Software Developer Manuals, Volume 2B.

Note: #InvalidEvent can be generated by the following:

- A CPU reset which is not caused by a TXT Reset
- A non-virtualized INIT event
- During RLP wakeup, bit 0 of the RLPs’ IA32_SMM_MONITOR_CTL MSR does not match that of the ILP
- An SENTER/SEXIT/WAKEUP event is received post-VMXON
- A thread wakes from the wait-for-SIPI state while another thread in the same CPU is executing an AC Module

B.1.4 TXT.CMD.RESET – System Reset Command

Description	A write to this register causes a system reset. The processor performs this as part of an Intel® TXT shutdown, after writing to the TXT.ERRORCODE register.
Offset	038H
Pub Attribs Priv Attribs	- WO

Bits	Field Name	Field Description
7:0		

B.1.5 TXT.CMD.CLOSE-PRIVATE – Close Private Space Command

Description	A write to this register causes the Intel® TXT-capable chipset private configuration space to be locked. Locality 2 will also be closed. Once locked, conventional memory read/write operations can no longer be used to access these registers. The private configuration space can only be opened for the MLE by successfully executing GETSEC[SENTER].
Offset	048H
Pub Attribs Priv Attribs	WO (a serializing operation, such as a read of the register, is required after the write to ensure that any future chipset operations see the write)

Bits	Field Name	Field Description
7:0		

B.1.6 TXT.SPAD – BOOTSTATUS

Description	This register is used by Startup ACM to convey to BIOS results of its operation. In general, it is of no interest to OS SW and is provided as reference. Bit definitions are not architectural and therefore are not guaranteed to be absolutely stable across all platform generations.
Offset	A0H
Pub Attribs Priv Attribs	RO RW

Bits	Field Name	Field Description
29:0	Reserved	
30	TXT Startup success	Indicates *TXT Startup Success*. Signifies successful TXT data preparation for BIOS, SINIT and MLE
46:31	Boot Status	General Startup ACM to BIOS status communication.
47	Memory power down executed	Indicates that memory content was cleared via power down
52:48	Boot Status details	Startup ACM to BIOS communication in MP platforms.
53	TXT Policy enable	Startup ACM indication of run-time enabled status of TXT. This can be result of either absence of FIT type 0xA record or that FIT type 0xA record exists and its policy setting enables TXT.
58:54	Boot Status details	Startup ACM to BIOS communication in MP platforms.
59	BIOS trusted	Indicates that BIOS is trusted.
60	TXT Policy disable	Indicates that TXT has been disabled by runtime FIT type 0xA record policy setting
61	Boot Status details	Startup ACM to BIOS communication in MP platforms. For details reader is referred to respective CBnT server BIOS specification.

62	CPU Error	Indicates ACM authentication error
63	S-ACM success	Indicates that S-ACM successfully enforced its logic for all provisioned technologies. Successful execution of TXT specifically is indicated by bit 30

B.1.7 TXT.DIDVID – TXT Device ID

Description	This register contains the vendor, device, and revision IDs for the memory controller or chipset.
Offset	110H
Pub Attribs Priv Attribs	RO RO

Bits	Field Name	Field Description
15:0	VID	Vendor ID: 8086 for Intel® components
31:16	DID	Device ID: specific to the chipset/platform
47:32	RID	Revision ID: specific to the chipset/platform
63:48	ID-EXT	Extended ID: specific to the chipset/platform

B.1.8 TXT.VER.EMIF – EMC Version Number Register

Description	This register identifies whether the memory controller or chipset is debug or release fused.
Offset	200H
Pub Attribs Priv Attribs	RO RO

Bits	Field Name	Field Description
30:0	Reserved	Reserved
31	DEBUG.FUSE	0 = Chipset is debug fused 1 = Chipset is production fused

B.1.9 TXT.CMD.UNLOCK-MEM-CONFIG – Unlock Memory Config Command

Description	When this command is invoked, the chipset unlocks all memory configuration registers.
Offset	218H
Pub Attribs Priv Attribs	- WO (a serializing operation, such as a read of the register, is required after the write to ensure that any future chipset operations.)

Bits	Field Name	Field Description
7:0		

B.1.10 TXT.SINIT.BASE – SINIT Base Address

Description	This register contains the physical base address of the memory region set aside by the BIOS for loading an SINIT AC module. If BIOS has provided an SINIT AC module, it will be located at this address. System software that provides an SINIT AC module must store it to this location.
Offset	270H
Pub Attribs Priv Attribs	RW RW

Bits	Field Name	Field Description
31:0		

B.1.11 TXT.SINIT.SIZE – SINIT Size

Description	This register contains the size (in bytes) of the memory region set aside by the BIOS for loading an SINIT AC module. This register is initialized by the BIOS.
Offset	278H
Pub Attribs Priv Attribs	RW RW

Bits	Field Name	Field Description
31:0		

B.1.12 TXT.MLE.JOIN – MLE Join Base Address

Description	Holds a physical address pointer to the base of the join data structure used to initialize RLPs in response to GETSEC[WAKEUP].
Offset	290H
Pub Attribs Priv Attribs	RW RW

Bits	Field Name	Field Description
31:0		

B.1.13 TXT.HEAP.BASE – TXT Heap Base Address

Description	This register contains the physical base address of the Intel® TXT Heap memory region. The BIOS initializes this register.
Offset	300H
Pub Attribs Priv Attribs	RW RW

Bits	Field Name	Field Description
31:0		

B.1.14 TXT.HEAP.SIZE – TXT Heap Size

Description	This register contains the size (in bytes) of the Intel® TXT Heap memory region. The BIOS initializes this register.
Offset	308H
Pub Attribs Priv Attribs	RW RW

Bits	Field Name	Field Description
31:0		

B.1.15 TXT.DPR – DMA Protected Range

Description	This register defines the DMA Protected Range of memory in which the TXT heap and SINIT region are located.
Offset	330H
Pub Attribs Priv Attribs	RW RW

Bits	Field Name	Field Description
0	Lock	Bits 19:0 are locked down in this register when this bit is set.
3:1	Reserved	Reserved
11:4	Size	This is the size of memory, in MB, that will be protected from DMA accesses. A value of 0x00 in this field means no additional memory is protected. The DPR range works independently of any other DMA protections, such as VT-d, and is done post any VT-d translation or TXT checks.
19:12	Reserved	Reserved
31:20	Top	Top address + 1 of DPR. This is the base of TSEG.

Note: Modern platforms utilizing SGX technology implement the DPR register as part of uncore, not as part of PCH and thus TXT.DPR register has been deprecated with no underlying HW functionality.
Actual DPR register is implemented as the PCIE register with a typical BDFR 0.0.0.0x5C. Platform manufacturers shall consult respective external data sheets to verify the above numeric value as it may change.

B.1.16 TXT.SCRATCHPAD – ACM_POLICY_STATUS

Description	This register is used by Startup ACM to convey to BIOS enforced policies. Only bits that might be of interest for OS SW are provided as a reference. Bit definitions are not architectural and therefore are not guaranteed to be absolutely stable across all platform generations.
Offset	378H
Pub Attribs Priv Attribs	RO RW

Bits	Field Name	Field Description
12:0	Reserved	
14:13	TPM type	TPM type detected by Startup ACM: = 0 – No TPM

		= 1 – dTPM 1.2 = 2 – dTPM 2.0 = 3 – PTT
31:15	Reserved	
34:32	S-CRTM Status	Startup ACM SCRTM establishment: = 0 – not established by Startup ACM > 0 – established by Startup ACM
35:33	Reserved	
36	TPM Startup locality	Indication of locality at which TPM2_Startup command was executed: = 0 – locality 3 = 1 – locality 0
63:37	Reserved	

B.1.17 TXT.CMD.OPEN.LOCALITY1 – Open Locality 1 Command

Description	Writing to this register "opens" the TPM locality 1 address range, enabling decoding by the chipset and thus access to the TPM. This locality is not automatically opened after GETSEC[SENTER] and must be opened explicitly. This locality will not be closed by the TXT.CMD.CLOSE-PRIVATE command.
Offset	380H
Pub Attribs Priv Attribs	- WO

Bits	Field Name	Field Description
7:0		

B.1.18 TXT.CMD.CLOSE.LOCALITY1 – Close Locality 1 Command

Description	Writing to this register "closes" the TPM locality 1 address range, disabling decoding by the chipset and thus access to the TPM. This locality will not be closed by the TXT.CMD.CLOSE-PRIVATE command.
Offset	388H
Pub Attribs Priv Attribs	- WO

Bits	Field Name	Field Description
7:0		

B.1.19 TXT.CMD.OPEN.LOCALITY2 – Open Locality 2 Command

Description	Writing to this register "opens" the TPM locality 2 address range, enabling decoding by the chipset and thus access to the TPM. This locality is automatically opened after GETSEC[SENTER]. This locality will be closed by the TXT.CMD.CLOSE-PRIVATE command.
Offset	390H
Pub Attribs Priv Attribs	- WO

Bits	Field Name	Field Description
7:0		

B.1.20 TXT.CMD.CLOSE.LOCALITY2 – Close Locality 2 Command

Description	Writing to this register "closes" the TPM locality 2 address range, disabling decoding by the chipset and thus access to the TPM. This locality will be closed by the TXT.CMD.CLOSE-PRIVATE command or by the GETSEC[SEXIT] instruction.
Offset	398H
Pub Attribs Priv Attribs	- WO

Bits	Field Name	Field Description
7:0		

B.1.21 TXT.PUBLIC.KEY – AC Module Public Key Hash

Description	This register contains the hash of the public key used for the verification of AC Modules. The size, hash algorithm, and value are specific to the memory controller or chipset.
Offset	400H
Pub Attribs Priv Attribs	RO RO

Bits	Field Name	Field Description
255:0		

Note: Modern platforms utilizing SGX technology implement the Public Key Hash register as part of the CPU core, not as part of PCH and thus TXT.PUBLIC.KEY register has been deprecated with no underlying HW functionality. Any value it may contain does not have meaning and must be disregarded.
Actual Public Key Hash is typically available in CPU MSRs 0x20::0x23.
Platform manufacturers shall consult respective external data sheets to verify the above numeric values as it may change.

B.1.22 TXT.CMD.SECRETS – Set Secrets Command

Description	Writing to this register indicates to the chipset that there are secrets in memory. The chipset tracks this fact with a sticky bit. If the platform reboots with this sticky bit set the BIOS AC module (or BIOS on multiprocessor TXT systems) will scrub memory. The chipset also uses this bit to detect invalid sleep state transitions. If software tries to transition to S3, S4, or S5 while secrets are in memory then the chipset will reset the system. The MLE issues the TXT.CMD.SECRETS command prior to placing secrets in memory for the first time.
Offset	8E0H
Pub Attribs Priv Attribs	- WO

Bits	Field Name	Field Description
7:0		

B.1.23 TXT.CMD.NO-SECRETS – Clear Secrets Command

Description	Writing to this register indicates there are no secrets in memory. The MLE will write to this register after removing all secrets from memory as part of the TXT teardown process.
Offset	8E8H
Pub Attribs Priv Attribs	- WO

Bits	Field Name	Field Description
7:0		

B.1.24 TXT.E2STS – Extended Error Status

Description	This register is used to read the status associated with various errors that might be detected. The contents of this register are preserved across soft resets.
Offset	8F0H
Pub Attribs Priv Attribs	RO RW

Bits	Field Name	Field Description
0	Reserved	Reserved
1	SECRETS.STS	0 = Chipset acknowledges that no secrets are in memory 1 = Chipset believes that secrets are in memory and will provide reset protection
63:2	Reserved	Reserved

B.1.25 TPM Platform Configuration Registers

The TPM contains Platform Configuration Registers (PCRs). The purpose of a PCR is to contain measurements. From a TPM standpoint, the TPM does not care what entity uses a PCR to store a measurement.

The TPM provides two types of PCRs: static and dynamic. Static PCRs only reset on system reset; dynamic PCRs reset upon request. Static PCRs are written by the static root of trust for measurement (SRTM). In the PC, the SRTM begins with the BIOS boot block. The dynamic PCRs are written by the dynamic root of trust for measurement (DRTM). In the PC, the DRTM is the process initiated by GETSEC[SENTER].

A PC TPM requires a minimum of 24 PCRs. The first 16 are designated the static Root of Trust and the next eight are designated the dynamic Root of Trust. Intel® TXT uses PCRs 17 and 18 within the dynamic Root of Trust to measure the MLE.

All PCRs for TPM 1.2 devices, static or dynamic, have the same size and same updating mechanism. The size is 160 bits. This size allows the PCRs to contain a SHA1 hash digest value. Storing a measurement value in the PCRs involves a TPM_Extend operation, which is itself a hash operation.

PCRs for TPM 2.0 devices will be sized appropriately for the largest digest resulting from algorithms they support, typically 256 bits or greater. This is elaborated in Section 1.8, 1.9, 1.10, and 3.6 above.

B.1.26 Intel® Trusted Execution Technology Device Space

There are several memory ranges within Intel® TXT address space provided to access Intel® TXT related devices. The first range is 0xFED4_xxxx that is divided up into 16

pages. Each page in the FED4 range has specific access attributes. A page in this region may be accessed by Intel® TXT cycles only, by Intel® TXT cycles and via private space, or by Intel® TXT cycles, private and public space.

Table 18. TPM Locality Address Mapping

Address Range	TPM Locality
FED4 0xxxH	Locality 0 (fully public)
FED4 1xxxH	Locality 1 (trusted OS)
FED4 2xxxH	Locality 2 (MLE access only)
FED4 3xxxH	Locality 3 (AC modules access only)
FED4 4xxxH	Locality 4 (Hardware or microcode access only)
All others	Reserved

The first five pages of the 0xFED4_xxxx region are used for TPM access. Each page represents a different locality to the TPM. Locality is an attribute used by the TPM to define how it treats certain transactions. The address range used for commands sent to the TPM defines locality. All Intel® TXT chipsets must support all localities. Locality 0 is considered public and accesses it is accepted by the chipset under all circumstances. Accesses to locality 0 are sent to the ICH even if Intel® TXT is disabled, there has been no SENTER, or private space is closed. Locality 4 is never open but may only be accessed with Intel® TXT cycles. There are Intel® TXT commands that will open localities 1 through 3. Localities 2-3 require that both TXT.CMD.OPEN.LOCALITY and TXT.CMD.OPEN-PRIVATE be done before allowing accesses in that range to be accepted. At reset, localities 1 through 3 are closed.

No status read check of the TPM is performed by the processor GETSEC [SENTER] instruction ahead of the TPM.HASH write sequence. If the TPM is not in acquiesced state at this time, then the PCRs 17-20 reset and hash registration to PCR 17 may not succeed. To ensure reliable system software functionality for TPM support, it is recommended that the GETSEC [SENTER] instruction only be executed once the TPM has acquiesced and ownership has been established in the context of the SENTER initiating process.

Upon successful execution of GETSEC [SENTER] and relinquishment of control by SINIT, the TPM's private space and locality 2 should be left open. No locality should be active.

§

Appendix C Intel® TXT Heap Memory

Intel® TXT Heap memory is a region of physically contiguous memory that is set aside by BIOS for the use of Intel® TXT hardware and software. The system software that launches the measured environment passes data to both the SINIT AC module and the MLE using Intel® TXT Heap memory. The system software is responsible for filling in the table contents prior to executing the SENTER instruction. An incorrect format or incorrect content of this table or tables described by this table will result in failure to launch the protected environment.

Table 19. Intel® Trusted Execution Technology Heap

Offset	Length (bytes)	Name	Description
0	8	BiosDataSize	Size in bytes of the Intel® TXT specific data passed from the BIOS to system software for the purposes of launching the MLE. This size includes the number of bytes for this field, so this field cannot be less than a value of 8[2].
8	BiosDataSize - 8	BiosData	BIOS specific data. The format of this data is described below in Table 20.
BiosDataSize	8	OsMleDataSize	Size in bytes of the data passed from the launching system software to the MLE. This size includes the number of bytes for this field, so this field cannot be less than a value of 8[1].
BiosDataSize + 8	OsMleDataSize – 8	OsMleData	System software -specific data. Format of data in this field is considered specific to the system software vendor.
BiosDataSize + OsMleDataSize	8	OsSinitDataSize	Size in bytes of the data passed from the launching system software to the SINIT AC module. This size includes the number of bytes for this field, so this field cannot be less than a value of 8[1].

[2] For proper data alignment on 64-bit processor architectures this field must be a multiple of 8 bytes.

Offset	Length (bytes)	Name	Description
BiosDataSize + OsMLEDataSize + 8	OsSinitDataSize – 8	OsSinitData	System software data passed to the SINIT AC module. The format of this data is described below in Table 22.
BiosDataSize + OsMleDataSize + OsSinitDataSize	8	SinitMleDataSize	Size in bytes of the data passed from the launched SINIT AC module to the MLE. This size includes the number of bytes for this field, so this field cannot be less than a value of 8[1].
BiosDataSize + OsMleDataSize + OsSinitDataSize + 8	SinitMleDataSize – 8	SinitMleData	SINIT data passed to the MLE. The format of this data is described below in Table 23.

Note: BiosDataSize + OsMleDataSize + OsSinitDataSize + SinitMleDataSize must be less than or equal to TXT.HEAP.SIZE.

C.1 Extended Data Elements

Extended data elements are self-describing data structures that will be used for all future extensions to TXT heap tables. The *ExtDataElements*[] field in each of the heap tables is an array/list of individual elements, terminated by a HEAP_END_ELEMENT:

```
ExtDataElements[] ::= <HEAP_EXT_DATA_ELEMENT>* |
<HEAP_END_ELEMENT>
```

Each element consists of the following data structure:

```
typedef struct {
    UINT32     Type;              // one of HEAP_EXTDATA_TYPE_*
    UINT32     Size;
    UINT8      Data[Size - 8];
} HEAP_EXT_DATA_ELEMENT;
```

The extended data element structures in the following sub-sections (named HEAP_*_ELEMENT) correspond to the contents of the *Data* field for the specific type of element.

While not required, it is recommended that *Size* be a 4-byte multiple.

Entities that use the *ExtDataElements*[] fields must ignore element types that they do not understand or care about. This allows forward and backward compatibility of these fields.

C.1.1 HEAP_END_ELEMENT

```
#define HEAP_EXTDATA_TYPE_END              0

typedef struct {
   UINT32       Type;           // = 0
   UINT32       Size;           // = 8
} HEAP_END_ELEMENT;
```

The HEAP_END_ELEMENT represents the terminating element of a given *ExtDataElements*[] list. It contains no *Data*[] field.

C.1.2 HEAP_CUSTOM_ELEMENT

```
#define HEAP_EXTDATA_TYPE_CUSTOM           4

typedef struct {
   UINT32 data1;
   UINT16 data2;
   UINT16 data3;
   UINT16 data4;
   UINT8 data5[6];
} UUID;

typedef struct {
   UUID         Uuid;
   UINT8        Data[];
} HEAP_CUSTOM_ELEMENT;
```

The HEAP_CUSTOM_ELEMENT allows for platform suppliers to communicate supplier-specific data through a standard location and mechanism. Software wishing to use this data must understand its format.

Uuid is a UUID value that uniquely identifies the format of the *Data* field. It is important to generate the UUID value using a process that will provide a statistically unique value.

The platform supplier defines the *Data* field's contents. The size of this data must be included within the size of the HEAP_EXTDATA_ELEMENT.*Size* field.

C.1.3 Benchmarking element

One of the usages of HEAP_CUSTOM_ELEMENT defined by CBnT is delivering of benchmarking information.

```
typedef struct {
   UUID  Uuid;       // 0x06A9C77B, 0x850B476B, 0xB3D0C9B4,
0x7BCB8A85
   UINT32   ModuleId;   // Creation data from the ACM header
   UINT32   Count;
   RECORD Record[Count];
}

typedef struct {
   UINT16   TaskNumber;
   UINT16 Tag;
   UINT64   Timestamp;
}
```

ModuleId is SINIT creation data from its header.

Count is count of benchmarking records.

Record is array of RECORD structures.

In a RECORD structure:

TaskNumber is internal sequential SINIT task number;

Timestamp is value of TSC counter at the time when record is created.

Tag is numeric tag associated with any point of interest of SINIT execution flow. For instance, duration of TPM command needs to be measured. For this purpose, two tags will be assigned – start tag and end tag. Sought duration will be different between two TSC timestamps (in TSC clocks);

Note: Benchmarking information is available in special build of NPW modules. Such modules are built by customer demand.

C.2 BIOS Data Format

The format of the data passed from the BIOS to the system software for the purposes of launching the measured environment is shown in Table 20.

Table 20. BIOS Data Table

Offset	Length (bytes)	Name	Description
0	4	Version	Version number of the *BiosData* table. The current value is 5 for TPM 1.2 family. TPM 2.0 requires version 6 (versions 4::6 are supported). This value is incremented for any change to the definition of this table. Future versions will always be backwards compatible with previous versions (new fields will be added at the end).

Offset	Length (bytes)	Name	Description
4	4	BiosSinitSize	This field indicates the size of the SINIT AC module provided by system BIOS. A value of 0 indicates the BIOS is not providing an SINIT AC module for system software use. A non-0 value indicates that the AC module will be at the location specified by the TXT.SINIT.BASE register and be of the specified size.
8	8	Reserved1	BIOS data table no longer contains *LcpPdBase* field since the field was associated with currently removed PS LCP *Policy Data File*.
16	8	Reserved2	BIOS data table no longer contains *LcpPdSize* field since this field was associated with currently removed PS LCP *Policy Data File*
24	4	NumLogProcs	This is the total number of logical processors in the system. The minimum value in this register must be at least 1.
Versions >= 3			
28	4	SinitFlags	BIOS-provided information for SINIT AC module consumption. Bit definition will be dependent on the chipset. Currently none are defined
Versions >= 5 with updates in version 6			
32	4	MleFlags	BIOS-provided information for system software and the MLE, about TXT capabilities of the BIOS. See Table 21.
Versions >= 4			
36	BiosDataSize - 36	ExtDataElements[]	Array/list of extended data element structures. See below for element definitions.

Table 21. MLE Flags Field Bit Definitions

Bit position	Description
Versions >= 5	
0	Support for TXT/VT-x/VT-d ACPI PPI specification Note 1
Versions >= 6	
2:1	00: legacy state / platform undefined 01: client platform, client SINIT is required 10: server platform, server SINIT is required 11: Reserved/illegal, must be ignored
All versions	
31:3	Reserved, must be zero

Note: "Intel® Trusted Execution Technology (Intel® TXT) – One-Touch Enabling, Intel TXT Provisioning Interface. Addendum to TCG Physical Presence Interface (PPI) Specification"

C.2.1 HEAP_BIOS_SPEC_VER_ELEMENT

```
#define HEAP_EXTDATA_TYPE_BIOS_SPEC_VER   1

typedef struct {
   UINT16       SpecVerMajor;
   UINT16       SpecVerMinor;
   UINT16       SpecVerRevision;
} HEAP_BIOS_SPEC_VER_ELEMENT;
```

The HEAP_BIOS_SPEC_VER_ELEMENT contains fields that indicate the version of the TXT BIOS specification to which this platforms BIOS corresponds. This element type is mainly useful for diagnostic tools.

C.2.2 HEAP_ACM_ELEMENT

```
#define HEAP_EXTDATA_TYPE_ACM             2

typedef struct {
   UINT32       NumAcms;
   UINT64       AcmAddrs[NumAcms];  // physical address of ACM
} HEAP_ACM_ELEMENT;
```

The HEAP_ACM_ELEMENT allows BIOS to indicate the ACMs that it contains and their locations in memory.

BIOS that support this element type should report all ACMs that they carry; both BIOS ACMs and SINIT ACMs.

Note: For SINIT ACM address in the AcmAddrs array shall point to the uncompressed module image in the TXT heap memory (Same as in TXT.SINIT.BASE register – see Appendix B.1.10).

Since the TXT architecture requires that BIOS provide at least one BIOS ACM, *NumAcms* must always be greater than 0.

AcmAddrs[] is an array of physical addresses of each of the ACMs.

C.2.3 HEAP_STM_ELEMENT

```
#define HEAP_EXTDATA_TYPE_STM             3

typedef struct {
   UINT8        Data[];
} HEAP_STM_ELEMENT;
```

The HEAP_STM_ELEMENT allows BIOS to indicate to the MLE STM related BIOS properties. Its format is specified in "SMI Transfer Monitor (STM) User Guide" https://firmware.intel.com/content/smi-transfer-monitor-stm.

The platform supplier defines the *Data* field's contents. The size of this data must be included within the size of the HEAP_EXTDATA_ELEMENT.*Size* field.

C.3 OS to MLE Data Format

Each system software vendor may have a different format for this data, and any MLE being launched by system software must understand the format of that software's handoff data.

C.4 OS to SINIT Data Format

Table 22 defines the format of the data passed from the launching system software to the SINIT AC module in the *OsSinitData* field.

Table 22. OS to SINIT Data Table

Offset	Length (bytes)	Name	Description
0	4	Version	Version number of the *OsSinitData* table. Current values are 6 for TPM 1.2 family and 7 for TPM 2.0 family. No other values are supported. This value is incremented for any change to the definition of this table. Future versions will always be backwards compatible with previous versions (new fields will be added at the end).
Versions <= 6			
4	4	Reserved	Reserved for future use
Version = 7			
4	4	Flags	Bit 0: PCR Extend Policy Control 0 – Maximum Agility Policy 1 – Maximum Performance Policy
Versions >=4			
8	8	MLE PageTableBase	Physical address of MLE page table (the MLE page directory pointer table address)
16	8	MLE Size	Size in bytes of the MLE image
24	8	MLE HeaderBase	Linear address of MLE header (linear address within the MLE page tables)

Offset	Length (bytes)	Name	Description
32	8	PMR Low Base	Physical base address of the PMR Low region (must be 2MB aligned). Can be set to zero if not desired to be enabled by SINIT. The MVMM must be loaded in one of the DPR, PRM low, or the PMR high regions.
40	8	PMR Low Size	Size of the PMR Low Region (must be 2MB granular). Set to zero if not desired to be enabled by SINIT.
48	8	PMR High Base	Physical base address of the PMR High region (must be 2MB aligned). Can be set to zero if not desired to be enabled by SINIT.
56	8	PMR High Size	Size of the PMR HIGH Region (must be 2MB granular). Set to zero if not desired to be enabled by SINIT.
64	8	LCP PO Base	Physical base address of the Platform Owner's Launch Control Policy, LCP_POLICY_DATA structure.
72	8	LCP PO Size	Size of the Launch Control Policy Platform Owner's Policy Data.
80	4	Capabilities	Bit vector of capabilities that SINIT is requested to use. This must be a subset of the ones SINIT supports - Table 4. Note that for CBnT bits 5:4 must be 11b, since D/A mapping is the only one supported.
Versions >= 6			
84	8	EFI RSDP Pointer	Physical address of RSDP when an EFI boot was performed. If not provided or invalid SINIT attempts to find the standard ACPI RSDP pointer.
92	OsSinitDataSize - 92	ExtDataElements[]	Array/list of extended data element structures. See below for element definitions. Note that for CBnT, there must be a single HEAP_EVENT_LOG_POINTER_ELEMENT(2_1) structure with a HEAP_END_ELEMENT terminating the list.

C.4.1 HEAP_TPM_EVENT_LOG_ELEMENT

This element specifies the pointer to the event log structure describing event log memory allocated by OS. Log will be created by SINIT. This element version is used to request TPM 1.2 style event log.

```
#define HEAP_EXTDATA_TYPE_TPM_EVENT_LOG_PTR  5

typedef struct {
   UINT64       EventLogPhysAddr;
} HEAP_TPM_EVENT_LOG_ELEMENT;
```

The HEAP_TPM_EVENT_LOG_ELEMENT is used for system software to inform SINIT of the location of the TPM PCR event log that the system software has allocated. See Appendix F for additional information about the log.

EventLogPhysAddr is the physical address of the event log structure. The event log structure must be completely below 4GB.

SINIT in TPM 1.2 mode will require HEAP_TPM_EVENT_LOG_ELEMENT to be present since event log is required for attestation.

C.4.2 HEAP_EVENT_LOG_POINTER_ELEMENT2_1

This element describes event log SINIT creates in TPM 2.0 mode which is compatible with format described in *"TCG PC Client Platform Firmware Profile"* specification.

Note that that HEAP_EXTDATA_TYPE_EVENT_LOG_POINTER2 element which was used in transition period when TCG event log format has not been yet finalized is no longer supported by SINIT.

```
#define HEAP_EXTDATA_TYPE_EVENT_LOG_POINTER2_1    8

   typedef struct {
      UINT64      PhysicalAddress;
      UINT32      AllocatedEventContainerSize;
      UINT32      FirstRecordOffset;
      UINT32      NextrecordOffset;
   } HEAP_EVENT_LOG_POINTER_ELEMENT2_1;
```

PhysicalAddress: Physical address of the event log base.

AllocatedEventContainerSize: Size of allocated event log memory.

FirstRecordOffset: Offset of the first record in event log.

NextRecordOffset: Offset of the free memory beyond the end of last entered record.

SINIT in TPM2.0 mode will require HEAP_EVENT_LOG_POINTER_ELEMENT2_1 to be present since event log is required for attestation and many of the *SinitMleData* table fields carrying similar data are removed in TPM2.0 mode – see Table 23

See F for further explanation.

C.5 SINIT to MLE Data Format

Table 23 below defines the format of the SINIT data presented to the MLE.

Table 23. SINIT to MLE Data Table

Offset	Length (bytes)	Name	Description
0	4	Version	Version number of the *SinitMleData* table. CBnT fixed the following version numbers based on TPM family: for TPM 1.2 generated table has version 8; for TPM 2.0 family generated table has value 9. This value is incremented for any change to the definition of this table. Future versions will always be backwards compatible with previous versions (new fields will be added at the end). See NOTE
Versions <= 8			
4	20	BiosAcmID	ID of the BIOS AC module in the system
24	4	EdxSenterFlags	Value of EDX SENTER control flags
28	8	MsegValid	MSEG MSR (Valid bit only)
36	20	SinitHash	SHA1 hash of the SINIT AC module: SHA1(SHA256(SINIT))
56	20	MleHash	SHA1 hash of the MLE
76	20	StmHash	SHA1 hash of STM. This is only valid if *MsegValid* = 1, else will contain zero
96	20	LcpPolicyHash	SHA1 Hash of the LCP policy that was enforced; if no hash is needed based on the LCP policy control field this will contain zero
116	4	PolicyControl	Taken from the LCP policy used
Versions >= 9			
4	20	Reserved	Must be zero
24	4	Reserved	Must be zero
28	8	Reserved	Must be zero
36	20	Reserved	Must be zero
56	20	Reserved	Must be zero
76	20	Reserved	Must be zero
96	20	Reserved	Must be zero
116	4	Reserved	Must be zero
Versions >= 7			
120	4	RlpWakeupAddr	MONITOR physical address used for waking up RLPs (write 32bit non-0 value)
124	4	Reserved	Reserved for future use. Must be zero
128	4	NumberOfSinitMdrs	Number of SINIT Memory Descriptor Records

Offset	Length (bytes)	Name	Description
132	4	SinitMdrTableOffset	Offset (in bytes, from start of this table) to the start of an array of SINIT Memory Descriptor Records as defined below. Each record describes a memory region as defined by the SINIT AC module (see Table 24).
136	4	SinitVtdDmarTableSize	Length of the Intel® Virtualization Technology (Intel® VT) for Directed I/O (Intel® VT-d) DMAR table pointed to by the *SinitVtdDmarTableOffset* field.
140	4	SinitVtdDmarTableOffset	Offset (in bytes, from start of this table) to the start of the SINIT provided DMAR table dump for the MLE.
Version = 8			
144	4	ProcessorSCRTMStatus	Bit 0 = 1 if PCR 0 measurement for this boot was rooted in processor hardware. This is possible only if all logical processors implement S-CRTM and the platform is designed to take advantage of that capability. Bit 0 = 0 if PCR0 measurement for this boot was rooted in BIOS. Bits 31:1 – Reserved for future use and must be zero.
Versions >= 9			
144	4	Reserved	Must be zero
Versions >= 8			
148	SinitMleDataSize - 148	ExtDataElements[]	Array/list of extended data element structures. See below for element definitions.

Note: Maximum version generated by SINIT in TPM 1.2 mode must be 8. In TPM 2.0 mode SINIT must generate version 9. Changes were caused by deprecating of number of fields having insufficient size for agile digests and replacing them by event log records.

Table 24. SINIT Memory Descriptor Record

Offset	Length (bytes)	Name	Description
0	8	Address	Physical address of the memory range described in this record.
8	8	Length	Length of the memory range.

Offset	Length (bytes)	Name	Description
16	1	Type	Memory range type. Valid values: 0 Usable, good memory 1 SMRAM– Overlaid – deprecated 2 SMRAM– Non-Overlaid – deprecated 3 PCIe - PCIe Extended Configuration Region 4 Persistent memory 5–255 Reserved
17	7	Reserved	Reserved for future use

The array of Memory Descriptor Records (MDRs) is not necessarily ordered and some MDRs may be of 0 length, in which case they should be ignored.

Memory of type 0 is usable for the MLE and any code or data that it may load. SINIT will verify that the MLE and its page table are located in memory of this type.

Memory types 1 and 2 are deprecated in future versions of SINIT, as SMRAM regions are not of use to the MLE.

Memory of type 3 is the PCI Express extended configuration region. The MLE may use this to verify that the PCIE configuration specified in the ACPI tables is using the appropriate address space.

Memory of type 4 is persistent, saving content in NV memory persisting over reset. If content is not encrypted, MLE shall not place secrets into this memory since it might be exposed upon reboot.

C.5.1 HEAP_MADT_ELEMENT

```
#define HEAP_EXTDATA_TYPE_MADT            6

typedef struct {
   UINT8        MadtData[];
} HEAP_MADT_ELEMENT;
```

The HEAP_MADT_ELEMENT contains a copy of the ACPI MADT table

MadtData contains a validated copy of the ACPI MADT table. Its size is specified in the MADT header as well as the *Size* field of the element. The format of the MADT table is described in the version of the "*Advanced Configuration and Power Interface Specification*" implemented by the platform.

C.5.2 HEAP_MCFG_ELEMENT

```
#define HEAP_EXTDATA_TYPE_MADT          9

typedef struct {
   UINT8        McfgData[];
} HEAP_MCFG_ELEMENT;
```

The *HEAP_MCFG_ELEMENT* contains a copy of the ACPI MCFG table

McfgData contains a validated copy of the ACPI MCFG table. Its size is specified in the MCFG header as well as the *Size* field of the element. The format of the MCFG table is described in the version of the "*Advanced Configuration and Power Interface Specification*" implemented by the platform.

C.5.3 Registry of Extended Heap Elements

Heap element types are global for all heap tables. Registry has been added for facilitating of reference.

Table 25. Extended Heap Elements Registry

Type	Label	Comment
0	HEAP_END_ELEMENT	This element is mandatory terminator of the list of extended elements after each of the heap tables.
1	HEAP_BIOS_SPEC_VER_ELEMENT	Element specifies revision of TXT BIOS WG specification supported by given BIOS
2	HEAP_ ACM_ELEMENT	Element provides information about quantity and position of BIOS embedded AC Modules, including SINIT if present.
3	HEAP_STM_ELEMENT	Element specifies STM related BIOS properties
4	HEAP_CUSTOM_ELEMENT	Element can be customized for non-standard use
5	HEAP_TPM_EVENT_LOG _ELEMENT	Element provides pointer to TPM 1.2 style Event Log.
6	HEAP_MADT_ELEMENT	Element encases MADT table copied to DPR
7	HEAP_EVENT_LOG_POINTER_ELEMENT2	Element provides pointers to Event Log(s) of original TXT TPM 2.0 style
8	HEAP_EVENT_LOG_POINTER_ELEMENT2_1	Element provides pointer to TCG Compliant TXT TPM 2.0 style Event Log

Type	Label	Comment
9	HEAP_MCFG_ELEMENT	Element encases MCFG table copied to DPR

§

Appendix D LCP Data Structures

SINIT supports LCP with both TPM 1.2 and TPM 2.0 families of devices. Despite that fundamentally all structures used by LCP in TPM 1.2 and TPM 2.0 modes of operations are designed for the same purpose, they are somewhat different between families.

To disambiguate usage of the structures in the following sections they are grouped by type with an explicit identification of the TPM family they are used with.

Common LCP definitions used with both supported TPM families are presented below.

- Identification of policy type and maximal number of lists in LCP data file:

```
#define LCP_POLTYPE_LIST            0
#define LCP_POLTYPE_ANY             1
#define LCP_MAX_LISTS               8
```

D.1 LCP Policy

Platform Owner policy structure has type LCP_POLICY (TPM 1.2) or LCP_POLICY2 (TPM2.0). Respective object is stored in the TPM NV PO index.

D.1.1 Policy Control

```
typedef struct {
    UINT32  reserved:1;
    UINT32  NPW_OK:1;          // NPW OK
    UINT32  Reserved2:1;
    UINT32  Pconf_Enforced:1; // Functionality see Appendix J.2.3
    UINT32     reserved3:28;
} PolicyControl
```

NPW_OK bit when set, tells Non-Production SINIT that it is allowed to run. If unset, SINIT will abort execution.

PCONF_Enforced bit when set, forces SINIT to seek second PCONF element match – see Appendix J.2.3. If unset, SINIT continues after first PCONF element match is found.

D.1.2 LCP_POLICY

TPM 1.2 structure only

The required fields for LCP_POLICY are as follows:

- Cryptographic definitions and structures

```
#define LCP_POLHALG_SHA1            0
```

Note: TPM 1.2 structures use non-TCG compliant crypto algorithm IDs.

```
typedef struct {
    UINT8           SHA1[20];
} LCP_HASH;

typedef struct {
    UINT16      Version;        // 0x0204
    UINT8       HashAlg;        // LCP_POLHALG_SHA1
    UINT8       PolicyType;     // one of LCP_POLTYPE_*
    UINT8       SINITMinVersion;
    UINT8       Reserved1;
    UINT16      DataRevocationCounters[LCP_MAX_LISTS];
    UINT32      PolicyControl;
    UINT8       MaxSinitMinVer;
    UINT8       Reserved1;
    UINT16      Reserved2;
    UINT32      Reserved3;
    LCP_HASH    PolicyHash;
 } LCP_POLICY;
```

Version must be 2.4

HashAlg identifies the hashing algorithm used for the *PolicyHash* field. For TPM 1.2 mode of operation it must be LCP_POLHALG_SHA1.

PolicyType indicates whether an additional LCP_POLICY_DATA structure is required. Supported values are LCP_POLTYPE_LIST and LCP_POLTYPE_ANY.

SinitMinVersion specifies the minimum version of SINIT that can be used.

DataRevocationCounters is an array of elements corresponding to an array of policy lists in LCP *Policy Data File*. This array provides a mechanism to revoke signed lists in that object.

The *PolicyControl* field contains policy bits with global LCP impact. Its content is deciphered in Appendix D.

The *MaxSinitMinVersion* field specifies the maximum value this policy will allow a revocation utility to set the AUX.AUXRevocation.SinitMinVer to.

PolicyHash field is described in Section 3.2.1.1

D.1.3 LCP_POLICY2

TPM 2.0 structure only

- Key algorithms

```
#define  TPM_ALG_RSA          0x0001 // Key algorithm
#define  TPM_ALG_ECC          0x0023
```

- Hashing algorithms

```
#define  TPM_ALG_SHA1         0x0004
#define  TPM_ALG_SHA256       0x000B
#define  TPM_ALG_SHA384       0x000C
#define  TPM_ALG_NULL         0x0010
#define  TPM_ALG_SM3_256      0x0012
```

- Signature algorithms

```
#define  TPM_ALG_RSASSA       0x0014
#define  TPM_ALG_RSAPSS       0x0016
#define  TPM_ALG_ECDSA        0x0018
#define  TPM_ALG_SM2          0x001B
```

- Digest sizes per hashing algorithm

```
#define  SHA1_DIGEST_SIZE     20
#define  SHA256_DIGEST_SIZE   32
#define  SHA384_DIGEST_SIZE   48
#define  SM3_256_DIGEST_SIZE  32

typedef union {
    UINT8 sha1[SHA1_DIGEST_SIZE];
    UINT8 sha256[SHA256_DIGEST_SIZE];
    UINT8 sha384[SHA384_DIGEST_SIZE];
    UINT8 sm3[SM3_256_DIGEST_SIZE];
} LCP_HASH2;
```

- Owner's selection of crypto algorithms

LcpHashAlgMask identifies the Platform Owner's selection of *HashAlgIDs* permitted during LCP policy evaluation. It uses the following *HashAlgID* mask definition:

```
typedef struct {
    UINT16 TPM_ALG_SHA1:1;       // BIT0
    UINT16 Reserved:2;           // BITS[2:1]
    UINT16 TPM_ALG_SHA256:1;     // BIT3
    UINT16 Reserved:1;           // BIT4
    UINT16 TPM_ALG_SM3_256:1;    // BIT5
    UINT16 TPM_ALG_SHA384:1;     // BIT6
    UINT16 Reserved:9;
} LCP_APPROVED_HASH_ALG;
```

LcpSignAlgMask identifies the Platform Owner's selection of signature algorithms permitted during LCP policy evaluation. A fully specified signature algorithm definition is comprised of the signature algorithm and its public key size, the internally used hash algorithm, and, for SM2, the curve used.

```
typedef struct {
    UINT32 Reserved:2                    // BITS[1:0]
    UINT32 TPM_ALG_RSA*/2048/SHA1:1      // BIT2: Legacy
    UINT32 TPM_ALG_RSA*/2048/SHA256:1    // BIT3: Legacy
    UINT32 Reserved:2                    // BITS[5:4]
    UINT32 TPM_ALG_RSA*/3072/SHA256:1    // BIT6: NOTE 1
    UINT32 TPM_ALG_RSA*/3072/SHA384:1    // BIT7: NOTE 1
    UINT32 Reserved2:4                   // BITS[11:8]
    UINT32 ECDSA/P256/SHA256:1           // BIT12: NOTE 2
    UINT32 ECDSA/P384/SHA384:1           // BIT13: NOTE 2
    UINT32 Reserved3:2                   // BITS[15:14]
    UINT32 TPM_ALG_SM2/SM2_CURVE/SM3:1 // BIT16: SMX
    UINT32 Reserved4:15
} LCP_APPROVED_SIGNATURE_ALG;
```

Note: Possible use of RSASSA or RSAPSS algorithm: based on alluded higher post-quantum resistance of RSA vs. ECDSA algorithms (https://eprint.iacr.org/2017/598) and to reduce LCP complexity modern SINITs are built without support of ECDSA signatures, however this support can be added per demand.

```
typedef struct {
    UINT16    Version;        // 0x0302 == Version 3.2
    UINT16    HashAlg;        // one of TPM_ALG_*
    UINT8     PolicyType;     // one of LCP_POLTYPE_*
    UINT8     SINITMinVersion;
    UINT16    DataRevocationCounters[LCP_MAX_LISTS];
    UINT32    PolicyControl;
    UINT8     MaxSinitMinVer;
    UINT8     Reserved;
    UINT16    LcpHashAlgMask  // LCP_APPROVED_HASH_ALG
    UINT32    LcpSignAlgMask  // LCP_APPROVED_SIGNATURE_ALG
    UINT32    Reserved2;
    LCP_HASH2 PolicyHash;
 } LCP_POLICY2;
```

All field definitions except of mask fields are identical to LCP_POLICY structure.

LacpHashAlgMask and *LcpSignAlgMask* fields allow Owner to select which algorithms SINIT can use when looking for a match.

D.2 LCP_POLICY_DATA

When LCP_POLICY*. PolicyType is set to LCP_POLTYPE_LIST, there must exist a *Policy Data File* which the SINIT policy engine will process.

Where this file resides on the platform is platform dependent, but it must be provisioned into the Intel® TXT heap data structures (see Appendix C.3) before executing the GETSEC[SENTER] instruction. While not required, it is recommended that software place the LCP_POLICY_DATA on a 4-byte aligned boundary to reduce access alignment penalties.

```
typedef struct {
    char             FileSignature[32];
    UINT8            Reserved[3];
    UINT8            NumLists;
    LCP_POLICY_LIST  PolicyLists[NumLists];
} LCP_POLICY_DATA, LCP_POLICY_DATA2
```

Note: When Intel ® TXT was upgraded to support TPM 2.0 family, LCP_POLICY_DATA structure was aliased to LCP_POLICY_DATA2 without changing of its content to maintain naming coherency.

FileSignature is the string “Intel(R) TXT LCP_POLICY_DATA\0\0\0\0”, where ‘\0’ is a single byte whose value is 0x00. This field is intended for use by software that needs to determine if a given file is an LCP_POLICY_DATA file.

The *Reserved* field must be set to all 0s.

The *NumLists* field must be less than or equal to LCP_MAX_LISTS.

Each list in *PolicyLists* may be either signed or unsigned.

D.3 LCP Policy List

D.3.1 List Signatures

SINIT supports two different signature formats: LCP_SIGNATURE2 designed for platforms using RSASSA signatures; and LCP_SIGNATURE2_1 designed for platforms utilizing RSAPSS signature algorithm.

Using both RSASSA and RSAPSS algorithms and therefore LCP_SIGNATURE2 and LCP_SIGNATURE2_1 signature format is not possible.

LCP_SIGNATURE2_1 is identical to the one used by Intel® Boot Guard technology.

D.3.1.1 LCP_SIGNATURE2 structure

The LCP_SIGNATURE2 structure includes as a subcomponent LCP_SIGNATURE structure historically used in TPM 1.2 mode but for clarity of its use in the new structure format aliased to LCP_RSA_SIGNATURE. Also, non-standard definitions of crypto algorithms used in LCP_SIGNATURE structures are deprecated.

```
typedef struct {
    UINT16     RevocationCounter;
    UINT16     PubkeySize;
    UINT8      PubkeyValue[PubkeySize];
    UINT8      SigBlock[PubkeySize];
} LCP_SIGNATURE, LCP_RSA_SIGNATURE;
```

The *RevocationCounter* field is a monotonically increasing value that can be used, in conjunction with the corresponding index of the *DataRevocationCounters* field in LCP_POLICY, to provide a method of revoking (or preventing rollback) of signed policies.

PubkeySize is public key size in bytes. Supported public key sizes are 2048 and 3072 bits and therefore field values are 256 or 384. Minimal required public key size is 2048 bits.

PubkeyValue is the modulus of the public key. The exponent is assumed to be fixed and must be 65537.

SigBlock is the actual signature with size 256 or 384 bytes.

Another subcomponent of LCP_SIGNATURE2 structure is LCP_ECC_SIGNATURE structure shown below.

```
typedef struct {
    UINT16  RevocationCounter;
    UINT16  PubkeySize;
    UINT32  Reserved;       // For future expansion
    UINT8   Qx[PubkeySize]  // x coordinate Public key
    UINT8   Qy[PubkeySize]  // y coordinate Public key
    UINT8   R[PubkeySize]   // R component of Signature
    UINT8   S[PubkeySize]   // S component of Signature
} LCP_ECC_SIGNATURE;
```

Here *RevocationCounter* and *PubkeySize* fields are identical to ones used in LCP_RSA_SIGNATURE structure except that *PubkeySize* must be 32 or 48 bytes corresponding to supported curves.

Qx and *Qy* are coordinates of the ECC/SM2 public key.

R and *S* are components of a signature.

As specified for all policy data, both the *PubkeyValue* and *SigBlock* must be in little-endian byte order. This may require tools that generate policies to reverse the byte order of keys and signatures produced by tools that use the ASN.1/big-endian format.

Taken together LCP_RSA_SIGNATURE and LCP_ECC_SIGNATURE form new LCP_SIGNATURE2 structure as follows.

```
typedef union {
    LCP_RSA_SIGNATURE RsaSignature;
    LCP_ECC_SIGNATURE EccSignature;
} LCP_SIGNATURE2;
```

D.3.1.2 LCP_SIGNATURE2_1 structure

This structure is complex and includes several substructures described below.

RSA_PUBLIC_KEY structure

```
typedef struct {
   UINT8    Version;             // 0x10
   UINT16   KeySize;
   UINT32   Exponent
   UINT8    Modulus[KeySize/8]
} RSA_PUBLIC_KEY;
```

Where *Version* is the version of the structure in *major:minor* format and must be equal to 0x10;

KeySize represents number of bits in the public key modulus. Supported values are 2048 and 3072 bits;

Exponent must be standard 0x10001;

Modulus is the modulus of a public key in little endian format – 256 or 384 bytes.

ECC_PUBLIC_KEY structure

```
typedef struct {
   UINT8    Version;             // 0x10
   UINT16   KeySize;
   UINT8    Qx[KeySize/8];
   UINT8    Qy[KeySize/8];
} ECC_PUBLIC_KEY;
```

Where *Version* is identical to RSA_PUBLIC_KEY above.

KeySize is number of bits of public key coordinate – must be 256 for SM2 algorithm and 256 or 384 for ECDSA algorithm.

Qx and *Qy* are two coordinates of public key in little endian format – 32 or 48 bytes each.

RSA_SIGNATURE structure

```
typedef struct {
   UINT8    Version;              // 0x10
   UINT16   KeySize;
   UINT16   HashAlg;
   UINT8    Signature[KeySize/8];
} RSA_SIGNATURE;
```

Where *Version* is the version of the structure in *major:minor* format and must be equal to 0x10;

KeySize represents the number of bits in the public key modulus. Supported values are 2048 and 3072 bits;

HashAlg is hash algorithm used by signing process. Supported are TPM_ALG_SHA1 (legacy only), TPM_ALG_SHA256, and TPM_ALG_SHA384;

Signature is RSASSA or RSAPSS signature in little endian format – 256 or 384 bytes.

ECC_SIGNATURE structure

```
typedef struct {
   UINT8    Version;              // 0x10
   UINT16   KeySize;
   UINT16   HashAlg;
   UINT8    R[KeySize/8];
   UINT8    S[KeySize/8];
} ECC_SIGNATURE;
```

Where *Version* is the identical to RSA_SIGNATURE definition;

KeySize is number of bits of signature coordinate – must be 256 for SM2 algorithm and 256 or 384 for ECDSA algorithm.

HashAlg is hash algorithm used by signing process. Supported are TPM_ALG_SM3_256 for SM2 signatures and TPM_ALG_SHA256, and TPM_ALG_SHA384 for ECDSA signatures, respectively.

R and *S* are SM2 or ECDSA signature coordinates in little endian format – 32 or 48 bytes each.

RSA_KEY_AND_SIGNATURE structure

```
typedef struct {
   UINT8    Version;              // 0x10
   UINT16   KeyAlg;               // TPM_ALG_RSA
   UINT8    Key[sizeof(RSA_PUBLIC_KEY)];
   UINT16   SigScheme;
   UINT8    Signature[sizeof(RSA_SIGNATURE)];
} RSA_KEY_AND_SIGNATURE;
```

Where *Version* is the version of the structure in *major:minor* format and must be equal to 0x10;

KeyAlg is key type algorithm per *"TCG Trusted Platform Module Library"* specification. Must be TPM_ALG_RSA;

Key is RSA_PUBLIC_KEY structure;

SigScheme is signature scheme. Must be TPM_ALG_RSASSA or TPM_ALG_RSAPSS;

Signature is RSA_SIGNATURE structure.

ECC_KEY_AND_SIGNATURE structure

```
typedef struct {
   UINT8    Version;              // 0x10
   UINT16   KeyAlg;               // TPM_ALG_ECC
   UINT8    Key[sizeof(ECC_PUBLIC_KEY)];
   UINT16   SigScheme;
   UINT8    Signature[sizeof(ECC_SIGNATURE)];
} ECC_KEY_AND_SIGNATURE;
```

Where *Version* is the version of the structure in major:minor format and must be equal to 0x10;

KeyAlg is key type algorithm per *"TCG Trusted Platform Module Library"* specification. Must be TPM_ALG_ECC;

Key is ECC_PUBLIC_KEY structure;

SigScheme is signature scheme. Must be TPM_ALG_ECDSA or TPM_ALG_SM2;

Signature is ECC_SIGNATURE structure.

KEY_AND_SUGNATURE structure

```
typedef union {
    RSA_KEY_AND_SIGNATURE RsaKeyAndSignature;
    ECC_KEY_AND_SIGNATURE EccKeyAndSignature;
} KEY_AND_SIGNATURE;
```

This structure is simply union of RSA and ECC key and signature structures

LCP_SIGNATURE2_1 structure

Finally, LCP_SIGNATURE2_1 structure is analogous to LCP_SIGNATURE2 structure where key and signature details are wrapped into *KeyAndSignature* field.

```
typedef struct {
    UINT16     RevocationCounter;
    KEY_AND_SIGNATURE   KeyAndSignature;
} LCP_SIGNATURE2_1;
```

Here *RevocationCounter* field is identical to one used in LCP_SIGNATURE2 structure.

- Tabular forms of key and signature structures

Since both RSA_KEY_AND_SIGNATURE and ECC_KEY_AND_SIGNATURE are quite complex and contain several nested substructures, the following tabular form of them shall clarify and visualize relationship of the components.

RSA_KEY_AND_SIGNATURE

<table>
<tr><th>Name</th><th>Size (bytes)</th><th>Sub-structure</th><th>Comment</th></tr>
<tr><td>Version</td><td>1</td><td></td><td>0x10</td></tr>
<tr><td>KeyAlg</td><td>2</td><td></td><td>Key algorithm, must be TPM_ALG_RSA = 0x1</td></tr>
<tr><td>Key</td><td>Varies</td><td>
<table>
<tr><th>Name</th><th>Size (bytes)</th><th>Comment</th></tr>
<tr><td>Version</td><td>1</td><td>0x10</td></tr>
<tr><td>KeySize</td><td>2</td><td>In bits = 2048, 3072</td></tr>
<tr><td>Exponent</td><td>4</td><td>0x10001</td></tr>
<tr><td>Modulus []</td><td>KeySize / 8</td><td>In LE format</td></tr>
</table>
</td><td>RSA_PUBLIC_KEY_STRUCT
Size varies: 263 bytes for 2048-bit key and 392 bytes for 3072-bit key</td></tr>
<tr><td>SigScheme</td><td>2</td><td></td><td>Signature scheme must be TPM_ALG_RSASSA = 0x14 or TPM_ALG_RSAPSS = 0x16.</td></tr>
<tr><td>Signature</td><td>Varies</td><td>
<table>
<tr><th>Name</th><th>Size (bytes)</th><th>Comment</th></tr>
<tr><td>Version</td><td>1</td><td>0x10</td></tr>
<tr><td>KeySize</td><td>2</td><td>In bits = 2048, 3072</td></tr>
<tr><td>HashAlg</td><td>2</td><td>Hash algorithm = SHA1, SHA256, SHA384, SHA512</td></tr>
<tr><td>Signature []</td><td>KeySize / 8</td><td>In LE format</td></tr>
</table>
</td><td>RSA_SIGNATURE_STRUCT
Size varies: 261 bytes for 2048-bit key and 390 bytes for 3072-bit key</td></tr>
</table>

ECC_KEY_AND_SIGNATURE

<table>
<tr><th>Name</th><th>Size (bytes)</th><th>Sub-structure</th><th>Comment</th></tr>
<tr><td>Version</td><td>1</td><td></td><td>0x10</td></tr>
<tr><td>KeyAlg</td><td>2</td><td></td><td>Key algorithm, must be TPM_ALG_ECC = 0x23</td></tr>
<tr><td>Key</td><td>Varies</td><td>
<table>
<tr><th>Name</th><th>Size (bytes)</th><th>Comment</th></tr>
<tr><td>Version</td><td>1</td><td>0x10</td></tr>
<tr><td>KeySize</td><td>2</td><td>In bits = 256, 384</td></tr>
<tr><td>Qx []</td><td>KeySize / 8</td><td>Qx coordinate in LE format</td></tr>
<tr><td>Qy []</td><td>KeySize / 8</td><td>Qy coordinate in LE format</td></tr>
</table>
</td><td>ECC_PUBLIC_KEY_STRUCT
Size varies: 67 bytes for 256-bit key and 99 bytes for 384-bit key</td></tr>
<tr><td>SigScheme</td><td>2</td><td></td><td>Signature scheme, must be TPM_ALG_ECDSA = 0x18 or TPM_ALG_SM2 = 0x1B.</td></tr>
<tr><td>Signature</td><td>Varies</td><td>
<table>
<tr><th>Name</th><th>Size (bytes)</th><th>Comment</th></tr>
<tr><td>Version</td><td>1</td><td>0x10</td></tr>
<tr><td>KeySize</td><td>2</td><td>In bits = 256, 384</td></tr>
<tr><td>HashAlg</td><td>2</td><td>Hash algorithm = SHA256, SHA384 for ECC and SM3 for SM2</td></tr>
<tr><td>Signature []</td><td>KeySize / 8</td><td>In LE format</td></tr>
</table>
</td><td>ECC_SIGNATURE_STRUCT
Size varies: 69 bytes for 256-bit key and 101 bytes for 384-bit key</td></tr>
</table>

D.3.2 List Structures

Historical LCP_POLICY_LIST structure used in only TPM 1.2 mode of operation is deprecated and replaced by LCP_POLICY_LIST2, and LCP_POLICY_LIST2_1 structure supporting algorithm agility.

Since the new list structures allow mixture of TPM 1.2 and TPM 2.0 style LCP elements, use of legacy LCP_POLICY_LIST structure is not required.

LCP_POLICY_LIST2 is designed for platforms using RSASSA signatures, and LCP_POLICY_LIST2_1 is designed for platforms utilizing RSAPSS signature algorithm.

The use of both RSASSA and RSAPSS algorithms and therefore LCP_POLICY_LIST2 and LCP_POLICY_LIST2_1 list format is not possible.

D.3.2.1 LCP_POLICY_LIST2 Structure

```
typedef struct {
    UINT16          Version;          // 0x0201
    UINT16          SigAlgorithm;     // one of TPM_ALG_* above
    UINT32          PolicyElementsSize;
    LCP_POLICY_ELEMENT  PolicyElements[];
#if (SigAlgorithm != TPM_ALG_NULL)
    LCP_SIGNATURE2    Signature;
#endif
} LCP_POLICY_LIST2;
```

Version must be 2.1.

SigAlgorithm field uses TPM2.0 compatible numeric values.

PolicyElementsSize specifies the size (in bytes) of all LCP_POLICY_ELEMENT structures in the object. It may be 0. *An* LCP_POLICY_LIST2 with no elements can be used as a "placeholder" signed list that can be updated at runtime with the actual signed data but without having to re-provision the LCP_POLICY in TPM NV.

If *SigAlgorithm* is not a TPM_ALG_NULL, then the *Signature* field must be present (else it must not). For a signed list, the signature will be calculated over the *entire* LCP_POLICY_LIST2 structure, including the *Signature* member, except for the *SigBlock* field (if *SigAlgorithm* is TPM_ALG_RSASSA) or R and S fields (if *SigAlgorithm* is TPM_ALG_ECDSA).

Mixture of TPM 1.2 and TPM2.0 style LCP_POLICY_ELEMENT structures is allowed.

Note: By allowing a mix of legacy and new LCP_POLICY_ELEMENT structures in the same list, each Authority can create one single list covering both TPM1.2 and TPM2.0 LCP data.

D.3.2.2 LCP_POLICY_LIST2_1 Structure

```
typedef struct {
    UINT16          Version;          // 0x0300
    UINT32          KeySignatureOffset;
    UINT32          PolicyElementsSize;
    LCP_POLICY_ELEMENT  PolicyElements[];
#if (KeySignatureOffset != 0)
    LCP_SIGNATURE2_1 KeySignature;
#endif
} LCP_POLICY_LIST2_1;
```

Version must be 3.0 to signify incompatible layout change.

KeySignatureOffset must be zero for unsigned lists. For signed lists it must be offset of *KeyAndSignature* field of KEYAND_SIGNATURE structure. Entire content of the list from

its base and up to *KeyAndSignature* field is covered by the signature (to disambiguate - *RevocationCounter* is also included into hashed area). *KeyAndSignature* itself is excluded from the hash.

PolicyElementsSize is identical to LCP_POLICY_LIST2 structure.

Mixture of TPM 1.2 and TPM2.0 style LCP_POLICY_ELEMENT structures is allowed.

D.3.2.3 LCP_LIST

```
typedef union {
    LCP_POLICY_LIST2    PolicyListOld;    // RSASSA
    LCP_POLICY_LIST2_1  PolicyListNew;    // RSAPSS
} LCP_LIST;
```

D.4 LCP_POLICY_ELEMENT

D.4.1 Policy Elements

The LCP structures used for TPM 1.2 and presented in the subsections below are not articulate enough to support the algorithmic agility possible with TPM 2.0 devices. New elements based on the existing elements have been defined to add such support. The changes have been designed to allow lists comprised of both TPM 1.2 and TPM 2.0 elements. This minimizes space requirements for NV RAM and simplifies processing logic. Where possible, the new element structures use constants as defined in the TCG 2.0 specification. In the subsections below each element definition is attributed as to what TPM family it is associated with.

D.4.2 Structure Endianness

Endianness deals with the sequencing order of stored bytes. There are two common sequencing orders: Little Endian (LE - format used by Intel) and Big Endian (BE – format used by TCG TPM specifications). All structures and data in the following sections are in Little Endian format, even LCP_POLICY, with a few exceptions of fields where some of the standard TPM structures are used:

- TPM_PCR_INFO_SHORT structure in PCONF TPM1.2 definition is a standard TPM 1.2 structure, it should be in Big Endian format, while the rest of PCONF structure is in Little Endian format.
- TPMS_QUOTE_INFO structure in PCONF TPM2.0 definition is a standard TPM 2.0 structure. Likewise, it should be in Big Endian format, while the rest of PCONF structure is in Little Endian format.

D.4.3 Generic Policy Element Structure

```
typedef struct {
    UINT32      Size;
    UINT32      Type;
    UINT32      PolEltControl;
    UINT8       Data[Size – 12];
} LCP_POLICY_ELEMENT;
```

The structures in the following sub-sections correspond to the contents of the *Data* field for the specific type of element.

While not required, it is recommended that *Size* be a 4-byte multiple.

PolEltControl field is split by half and each half has different scope of applicability.

- Lower bits 15:0 have LIST scope (i.e., impact all elements of a given LCP list).
- Higher bits 31:16 are intended for element type–specific uses.

Current PolEltControl field bit definitions are as the following:

```
typedef struct {
   UINT32       Reserved: 1        // Must be 0
   UINT32       STM_Required: 1
   UINT32       Pcr18_Extends: 1
   UINT32       Reserved2: 13      // Must be 0
   UINT32       Reserved3: 16      // Must be 0
} PolEltControl
```

STM_Required bit when set to “1” requires STM to be enabled. Defined in *LCP_MLE_ELEMENT* structure only and is reserved in all other element type structures.

Pcr18_Extends bit controls whether certain extends to PCR18 are carried or not. If set to “1” events EVTYPE_KM_INFO_HASH and EVTYPE_BPM_INFO_HASH are skipped – see Table 29. If set to “0” all events are created.

Bits of *Reserved2* (15:3) and *Reserved3* (31:16) fields are currently undefined and should be set to zero. Use of *Reserved* bit (0) is obsolete. It must be set to zero.

D.4.4 LCP_MLE_ELEMENT

TPM 1.2 mode element only.

```
#define LCP_POLELT_TYPE_MLE       0

typedef struct {
    UINT8        SINITMinVersion;
    UINT8        HashAlg;                    // one of LCP_POLHALG_*
    UINT16       NumHashes;
    LCP_HASH     Hashes[NumHashes];
} LCP_MLE_ELEMENT;
```

The LCP_MLE_ELEMENT represents a list of the acceptable MLEs, as measured by their hashes. SINIT will match the policy if MLE hash (as calculated when traversing its pages in the MLE page table) matches any hash within the list.

SINIT will use the largest of the *SinitMinVersion* fields (the one in LCP_POLICY and the one in the LCP_MLE_ELEMENT which contains the matching MLE hash) to determine the minimum allowable version of SINIT.

HashAlg specifies the hash algorithm to use when measuring the MLE and values in *Hashes[]*. This must be LCP_POLHALG_SHA1 as the only algorithm supported by TPM 1.2. If *NumHashes* is 0, then this element will evaluate to false for all MLEs.

D.4.5 LCP_PCONF_ELEMENT

TPM 1.2 mode element only.

```
#define LCP_POLELT_TYPE_PCONF      1

typedef struct {
    UINT16                  NumPCRInfos;
    TPM_PCR_INFO_SHORT      PCRInfos[NumPCRInfos];
} LCP_PCONF_ELEMENT;
```

The LCP_PCONF_ELEMENT represents a list of acceptable PCR values on the platform at launch. The platform will satisfy the policy if the PCR values at the time of launch match any of the *PCRInfos* within the list. When processing the platform configuration list, the LCP engine reads the appropriate PCR's as defined by the first TPM_PCR_INFO_SHORT value in the list, concatenates them and cryptographically hashes them together. The result is compared to the hash value in the TPM_PCR_INFO_SHORT. If there is no match, this process is repeated for each member of the list. As soon as a match is found, the LCP engine proceeds.

Additionally, it is recommended, although not necessary, that all TPM_PCR_INFO_SHORT structures in the platform configuration list test the same set of PCR values. If *NumPCRInfos* is 0, then this element will evaluate to false for all platform configurations.

Various TPM_* structures have been copied below to facilitate understanding of the list structure.

```
typedef struct tdTPM_PCR_INFO_SHORT{
    TPM_PCR_SELECTION          pcrSelection;
    TPM_LOCALITY_SELECTION     localityAtRelease;
    TPM_COMPOSITE_HASH         digestAtRelease;
} TPM_PCR_INFO_SHORT;

typedef struct tdTPM_PCR_SELECTION {
    UINT16                          sizeOfSelect;
    [size_is(sizeOfSelect)] BYTE      pcrSelect[];
} TPM_PCR_SELECTION;

#define TPM_LOCALITY_SELECTION BYTE // Each bit is the
//corresponding locality

typedef struct tdTPM_DIGEST{
    BYTE digest[digestSize];
} TPM_DIGEST;
typedef TPM_DIGEST  TPM_COMPOSITE_HASH;  // hash of
//TPM_PCR_COMPOSITE object
```

D.4.6 LCP_CUSTOM_ELEMENT

```
#define LCP_POLELT_TYPE_CUSTOM    3

typedef struct {
    UINT32    data1;
    UINT16    data2;
    UINT16    data3;
    UINT16    data4;
    UINT8     data5[6];
} UUID;

typedef struct {
    UUID      uuid;
    UINT8     Data[];
} LCP_CUSTOM_ELEMENT;
```

The LCP_CUSTOM_ELEMENT allows users, ISVs, IT, etc. to define policy-related data which can then be carried as part of a policy and interpreted by user/ISV/IT software. Because the data is contained within a policy, its integrity will be verified by SINIT as part of policy processing.

Uuid is a UUID value that uniquely identifies the format of the *Data* field. This field will be used by all custom software that may have its own policy data. It is thus important to generate the UUID value using a process that will provide a statistically unique value.

The *Data* field's contents are defined by the entity that "owns" the UUID of the element. The size of this data must be included within the size of the LCP_POLICY_ELEMENT.*Size* field.

D.4.7 LCP_MLE_ELEMENT2

TPM 2.0 mode element only.

```
#define LCP_POLELT_TYPE_MLE2 0x10
typedef struct {
    UINT8      SINITMinVersion;
    UINT8      Reserved
    UINT16     HashAlg;    // one of TPM_ALG_*
    UINT16     NumHashes;
    LCP_HASH2  Hashes[NumHashes];
} LCP_MLE_ELEMENT2;
```

Definitions of all fields are identical to LCP_MLE_ELEMENT.

D.4.8 LCP_PCONF_ELEMENT2

TPM 2.0 mode element only.

The PCONF Element structure is designed such that it can be created using the TPM2_Quote command and evaluated using the TPM2_PolicyPCR command.

Following below is number of structures copied from TCG TPM 2.0 library specification to facilitate understanding of PCONF creation and evaluation process.

The TPM2_Quote command returns the following structure:

```
typedef struct {
    UINT16        size;
    TPMS_ATTEST   attestationData;
} TPM2B_ATTEST;
```

Where TPMS_ATTEST has the following form (only the last field is shown):

```
typedef struct {
   (...) ;
TPMU_ATTEST attested; // == TPMS_QUOTE_INFO
} TPMS_ATTEST;
```

TPMU_ATTEST is a union, where the relevant member within is the TPMS_QUOTE_INFO structure:

```
typedef struct {
    TPML_PCR_SELECTION  pcrSelect;
    TPM2B_DIGEST        pcrDigest
} TPMS_QUOTE_INFO;
```

pcrSelect is a TPML_PCR_SELECTION structure denoting the PCRs being quoted, and the *pcrDigest* is a TPM2B_DIGEST structure wherein the digest is a composite digest of the PCRs being quoted.

```
typedef struct {
    UINT16  hash;    // One of TPM_ALG_* algorithm IDs
    UINT8   sizeofSelect;
    UINT8   pcrSelect[sizeofSelect];
} TPMS_PCR_SELECTION;

typedef struct {
    UINT32        count;   // must be 1 for use in PCONF
    TPMS_PCR_SELECTION  pcrSelections;
} TPML_PCR_SELECTION;

typedef struct {
    UINT16  size;
    UINT8   buffer[size];
} TPM2B_DIGEST;
```

The new PCONF_ELEMENT structure is then defined as:

```
#define LCP_POLELT_TYPE_PCONF2 0x11
typedef struct {
    UINT16          HashAlg;    // one of TPM_ALG_*
    UINT16          NumPCRInfos;
    TPMS_QUOTE_INFO  PCRInfos[NumPCRInfos];
} LCP_PCONF_ELEMENT2;
```

Since TPMS_QUOTE_INFO is a standard TPM 2.0 structure its fields, where applicable, are inserted in big endian format. Rest of the LCP_PCONF_ELEMENT2 structure is in little endian format.

TPML_PCR_SELECTION structure, which is a sub-component of the new LCP_PCONF_ELEMENT2 definition, contains the “count” field which allows making multiple PCR selections using different *HashAlgIDs*.

The “*count*” field is kept in PCONF element definition to use as much of unmodified TPM2.0 structures as possible but for the purpose of TXT *count* will be enforced to be “1”. SINIT code will validate “count == 1” condition and will abort if it is not met.

D.4.9 LCP_STM_ELEMENT2

TPM 2.0 mode element only.

```
#define LCP_POLELT_TYPE_STM2 0x14
typedef struct {
    UINT16     HashAlg;    // one of TPM_ALG_*
    UINT16     NumHashes;
    LCP_HASH2  Hashes[NumHashes];
} LCP_STM_ELEMENT2;
```

The LCP_STM_ELEMENT represents a list of the acceptable MSEG objects, which can be either an STM, or a PPAM module used by Intel® Nifty Rock technology, as measured by

their hashes. SINIT will match the policy if MSEG object hash (as calculated when evaluating its header data) matches any hash within the list.

HashAlg specifies the hash algorithm to use when measuring the MSEG object and the values in *Hashes[]*.

If *NumHashes* is 0 then this element will evaluate to false for all MSEG objects.

D.5 NV AUX Index Data Structure

The AUX Index data structure requires some modification to accommodate algorithmically agile digests. The modified structure definition follows:

```
typedef struct {
   UINT8 MinVer;
   UINT8 Flags;
} ACM_AUX_REVOCATION;


typedef struct {
   ACM_AUX_REVOCATION SinitRevocation;
   ACM_AUX_REVOCATION Reserved;
} Revocation Area;
```

Table 26. AUX Data Structure

Revocation area	BIOS AC registration data	AUX Data Version	Other data	Internal TXT digest area

The table above diagrammatically illustrates contents of the AUX index.

Revocation area – storage containing minimal revisions of modules allowed to run.

BIOS AC registration data - contains value uniquely identifying BIOS ACM installed in the system. This value is extended into PCR 17 by SINIT module since BIOS ACM is in TXT TCB

AUX Data Version – identifies version of the structure. This version allows SINIT module to farther qualify AUX data and prevent forward compatibility issues.

Other data – BIOSAC / SINIT intercommunication area.

Internal TXT digest area - BIOSAC / SINIT intercommunication area containing variable size data values in TPMT_HA format.

§

Appendix E Platform State upon SINIT Exit and Return to MLE

The following table describes the state of the ILP after returning to the MLE from GETSEC[SENTER] and the RLPs after waking from SENTER. This is the state seen by the MLE.

Table 27. Platform State upon SINIT exit and return to MLE

Resource	ILP on MLE re-entry point	RLP on MLE re-entry point
CPU		
CR0	PG←0, AM←0,WP←0; others unchanged	PG←0, CD←0, NW←0, AM←0, WP←0; PE←1, NE←1
XCR0	AVX State←1 [Note 1], SSE State←1; others unchanged	Unchanged
CR3	Undefined	Undefined
CR4	0x00004000	0x00004000
EFLAGS	0x000000XX (XX = Undefined)	0x000000XX (XX = Undefined)
EIP	[MLEHeader.EntryPoint]	[TXT.MLE.JOIN + 12]
ESP	Undefined	Undefined
EBP	Undefined	Undefined
ECX	Pointer to MLE page table [Note 2]	Undefined
EBX	[MLEHeader.EntryPoint]	Undefined
EAX, EDI, ESI	Undefined	Undefined
CS	Sel=[SINIT.SegSel], base=0, limit=0xFFFFF, G=1, D=1, AR=0x9B	Sel=[TXT.MLE.JOIN + 8], base=0, limit=0xFFFFF, G=1, D=1, AR=0x9B
DS, ES, SS	Undefined	Sel=[TXT.MLE.JOIN + 8] + 8, base=0, limit=FFFFFH, G=1, D=1, AR=0x93
GDTR	Base=[SINIT.GDTBase], Limit=[SINIT.GDTLimit] [Note 3]	Base=[TXT.MLE.JOIN + 4], Limit=[TXT.MLE.JOIN]
DR7	0x00000400	0x00000400
MMX/XMM registers	Values at the SENTER unchanged.	Values at the SENTER unchanged.
YMM / ZMM	Undefined	Undefined
IA32_DEBUGCTL MSR	0	0
IA32_EFER MSR	0	0

IA32_MISC_ENABLE MSR	IA32_MISC_ENABLE & 0xFFF37CEA Notes 4, 5	IA32_MISC_ENABLE & 0xFEE324A8 Note 5
Performance counters and counter control registers	0	0
IA32_APIC_BASE MSR	35:12 cleared to 0xFEE00 NOTE 6	35:12 cleared to 0xFEE00, bit 8 (BSP) cleared to 0 NOTE 6
LTCS		
Private Space	Open	
TXT.ERRORCODE	0xC0000001	
TPM		
Locality	No locality active. Locality 2 open,	
PCI		
PCI Index/Data ports 0xCF8-0xCFF	Undefined	
VT-D		
Global Command and Global Status registers	Queued Invalidation (QI) and Interrupt Remapping (IR) is forced to be disabled in all enabled VT-d units.	

Note:

- AVX State is set only if CPU supports AVX.
- If bit 2 of the Capabilities field in SINIT's Chipset AC Module Information Table is set, then ECX will contain the pointer to the MLE's page table. If clear, the content of ECX is undefined.
- GDTR on entry to MLE retains values established by SINIT and is therefore incorrect and unusable for MLE. MLE developers should establish their own GDT immediately.
- Bit 3 (thermal monitor enabled) will be set to 1 if it was previously clear.
- Bit 18 (MONITOR/MWAIT enable) will be set to 1 if it was previously clear and bit 1 of *OsSinitData.Capabilities* (use of MONITOR for RLP wakeup) is set.
- Before exit, SINIT module restores ILP IA32_APIC_BASE MSR APIC base address to the value of ACPI MADT table. Restoring the RLP ILP and IA32_APIC_BASE MSR values is responsibility of the MLE.
- The TPM will not have any locality active following SENTER.

§

Appendix F TPM Event Log

The TPM Event Log is a data structure that describes the data whose hashes are extended to the TPM PCR indices. This allows remote attestation verifiers to reconstruct the PCR values in order to make trust decisions about their components. This event log is equivalent to the one generated by BIOS (see "*TCG PC Client Specific Implementation Specification for Conventional BIOS*") but is for the use of system software and SINIT.

System software allocates the memory for the log and provides the physical address of the container (see Table 28) to SINIT in a HEAP_TPM_EVENT_LOG_ELEMENT in the *OsSinitData.ExtDataElements*[] field – see Appendix C.4.1 or in a HEAP_TPM_EVENT_LOG_ELEMENT2_1 element – see Appendix C.4.2.

A CBnT compatible SINIT ACM supports only details/authorities PCR mappings (see – Section 1.10.2 and Table 4) and requires this element to be present. SINIT ACM will abort execution if system software does not provide this element.

There are no requirements for event log to be in DMA protected memory – SINIT will not enforce it.

F.1 TPM 1.2 Event Log

Event Log Container

Logically Event log Container comprises of an Event Log header followed by an array of records. Header provides full description of a log while records are structures describing actual extend events.

Table 28. Event Log Container Format

Field	Offset	Size (bytes)	Description
Signature	0	20	"TXT Event Container\0"
Reserved	20	12	Must be 0
ContainerVerMajor	32	1	Major version number of this structure. The current value is 1. Different major versions indicate incompatible structure format and/or behaviors.
ContainerVerMinor	33	1	Minor version number of this structure. The current value is 0. Different minor versions indicate compatible structure format (i.e., new fields added at the end) and/or behaviors.

Field	Offset	Size (bytes)	Description
PCREventVerMajor	34	1	Major version number of the PCREvent structure. The current value is 1. Different major versions indicate incompatible structure format and/or behaviors.
PCREventVerMinor	35	1	Minor version number of the PCREvent structure. The current value is 0. Different minor versions indicate compatible structure format (i.e., new fields added at the end) and/or behaviors.
ContainerSize	36	4	Allocated size of container, including PCREvents[]
PCREventsOffset	40	4	Offset (in bytes, from start of this table) to the start of PCREvents[] array.
NextEventOffset	44	4	Offset (in bytes, from start of this table) of the next byte after the last event in PCREvents[]. I.e. the offset of the next available event slot.
PCREvents[]	PCREventsOffset	Container Size - PCREventsOffset	Array of event records - TPM12_PCREvent structures (see below)

Record Structure

```
typedef struct {
    UINT32    PCRIndex;
    UINT32    Type;
    UINT8     Digest[20];
    UINT32    Size;
    UINT8     Data[Size];
} TPM12_PCREvent;
```

TPM12_PCREvent structure describes a single event record where the *Data* field contains actual data whose SHA1 hash was extended into the specified *PCRIndex*. Depending on the event *Type*, the *Data*[] may be the entire hashed object or just a description (e.g., version). All PCR events in the *PCREvents* [] array are in the order in which the PCR extends were performed. Because *Data*[] is not a fixed size for each event, the events must be traversed in order.

PCRIndex is the index of the TPM PCR into which the hash of the data was extended.

Type indicates the event type, as described in Table 29.

Digest is the SHA1 hash that was extended in this event.

Size is the size of the *Data*.

Data is either the actual data that digested to *Digest*, or a description of same, determined by *Type*.

F.2 TPM 2.0 Event Log

"*TCG PC Client Platform. EFI Protocol Specification*" defined new event logging structure suitable for multi-algorithm event logging. To maintain TCG compliance, SINIT modules will support this event log format in TPM 2.0 mode of execution.

TCG Compliant Event Logging Structures

Information below is identical to one presented in "*TCG PC Client Platform. EFI Protocol Specification*" and is included for easy reference. In case of any discrepancies, information included in the TCG specification shall take precedence.

```
typedef struct {
  UINT32                PCRIndex;
  UINT32                EventType;
  TPML_DIGEST_VALUES    Digest;
  UINT32                EventSize;
  BYTE                  Event[EventSize];
} TCG_PCR_EVENT2;
```

Where TPML_DIGEST_VALUES structure layout is presented below:

```
typedef struct {
  UINT32    count;          // Count of TPMT_HA structures
  struct {
    UINT16  AlgorithmID;    // Hash algorithm of array element
X
                            // TPMI_ALG_HASH
    UINT8   digest[AlgorithmID_DIGEST_SIZE] // Digest value in
                                            // array element X
  } TPMT_HA digests[count] // Array of TPMT_HA structures
} TPML_DEGEST_VALUES;
```

Note that although the type names from *TPM 2.0 Library Specification* are used, the encoding of the *count* member and the *AlgorithmID* are little-endian as is the rest of the log format.

First record of the log must follow TCG_PCR_EVENT structure in TPM 1.2 format and declare revision of the supported EFI Platform specification

```
typedef struct {
  UINT32 PCRIndex;          // 0
  UINT32 EventType;         // 3 == EV_NO_ACTION
  UINT8  Digest[20];        // ZeroSha1Digest[20] = {00, 00…00}
  UINT32 EventDataSize;     // sizeof(EventData[])
  UINT8  EventData[];       // TCG_EfiSpecIDEventStruct
} TCG_PCR_EVENT;
```

```
typedef struct {
  UINT8   signature[16];      // 'Spec ID Event03', 00
  UINT32  platformClass;      // 00 – PC Client Platform
Class;
                              // 01 – Server Platform Class
  UINT8   specVersionMinor;   // 00 – minor of 2.00
  UINT8   specVersionMajor;   // 02 – major of 2.00
  UINT8  specErrata;       // 00 – errata of 2.00
  UINT8  uintnSize;        // 01 for UINT32; 02 for UINT64
  UINT32 numberOfAlgorithms; // Number of hash algorithms used
in this event log
  TCG_EfiSpecIdEventAlgorithmSize digestSizes[numberOfAlgorithms]
                           // array/ of value pairs
  UINT8  vendorInfoSize;   // FF is maximum
  UINT8  vendorInfo[VendorInfoSize]; // Vendor specific
} TCG_EfiSpecIDEventStruct;

typedef struct {
  UINT16      algorithmId;
  UINT16      digestSize;
} TCG_EfiSpecIdEventAlgorithmSize;
```

The digest of this record MUST NOT be extended into any PCR.

Bit9 in the capabilities field of ACM Info Table *ACMInfoTable.Capabilities*[9] added to declare format of supported event log – see Table 4 with this revision of specification is always forced to "1".

F.3 Events Types

The following event types are defined for SINIT event logging. Events for legacy PCR mapping (LG) are removed from the table.

For brevity of description the following equation – Digest = $Hash_{HashAlgId}$ (*EventData*) shall be read as:

- TPM1.2 – SHA1 (*EventData*);
- TPM2.0 – TPML_DIGEST_VALUES structure filled with respective number of hashes corresponding to HashAlgIds of all PCR banks.

Table 29. Event Types

Event Type	Value	Digest and Event	PCR	Comments
EVTYPE_BASE (EB)	0×400	Base of TXT event types	N/A	
EVTYPE_PCR_MAPPING	EB + 1	*Digest*= ZeroDigest$_{HashAlgIDSize}$ NOTE 1 *EventData*=Selected PCR Mapping DWORD *EventData*=0 if using legacy PCR Mapping; *EventData*=1 if using D/A mapping. *EventDataSize* = 4	0xFF	Not extended – informative only. Since LG mapping is not supported this event is reserved.
EVTYPE_HASH_START	EB + 2	*Digest*=Hash$_{HashAlgID}$ (*EventData*) *EventData*=SinitHash + EDX *EventDataSize* = 36.	17	*EventData* is SINIT Hash and EDX value stored during SINIT authentication process in ACM header scratch area.
EVTYPE_MLE_HASH	EB + 4	*Digest*= Hash$_{HashAlgID}$ (MLE) *EventData*=EmptyBuffer NOTE 2 *EventDataSize=0*	17	
EVTYPE_BIOSAC_REG_DATA	EB + 10	*Digest* = BIOS AC registration data *EventData*=EmptyBuffer *EventDataSize* = 0	17	TPM 1.2: BIOS AC registration data are 20 bytes of data stored in AUX index, uniquely identifying BIOS ACM.
		Digest= Hash$_{HashAlgID}$ (*EventData*) *EventData*=BIOS AC Registration Data *EventDataSize*=32	17	TPM 2.0: *EventData* are 32 bytes of data stored in AUX index, uniquely identifying BIOS ACM.
EVTYPE_CPU_SCRTM_STAT	EB + 11	*Digest*= Hash$_{HashAlgID}$ (*EventData*) *EventData*=CPU SCRTM status DWORD *EventData*=0 if SCRTM was established by BIOS; *EventData*=1 if SCRTM was established by CPU *EventDataSize* = 4	18	TPM 1.2
			17 & 18	TPM 2.0
EVTYPE_LCP_CONTROL_HASH	EB + 12	*Digest* = Hash$_{HashAlgID}$ (*EventData*) *EventData* = LCP *PolControl* DWORD *EventDataSize=4*	17 &18	*EventData* is effective *PolControl* derived from TPM PO Index. If index does not exist *PolControl*=0

Event Type	Value	Digest and Event	PCR	Comments
EVTYPE_ELEMENTS_HASH	EB + 13	*Digest* = SHA1(*EventData*) *EventData* = array of matching LCP elements data as filled by LCP. *EventDataSize*=3 * (20 + 4)=72	17	TPM 1.2
		N/A	N/A	Reserved in TPM2.0 mode
EVTYPE_STM_HASH	EB + 14	If STM is present *Digest*=SHA1(STM) If STM is not present *Digest*=ZeroDigest$_{20}$ NOTE 1 *EventData*=EmptyBuffer NOTE 2 *EventDataSize=0*	17	TPM 1.2: If STM is not present ZeroDigest$_{20}$ is extended instead.
		If STM is present *Digest*=Hash$_{HashAlgID}$ (STM) If STM is not present *Digest*= Hash$_{HashAlgID}$ (0x00) *EventData*=Empty buffer EventDataSize=0		TPM 2.0: If STM is not present, hash of single byte with zero value will be measured instead
EVTYPE_OSSINITDATA_CAP_HASH	EB + 15	*Digest*=Hash$_{HashAlgID}$ (*EventData*) *EventData=OsSinitData Capabilities* DWORD *EventDataSize=4*	18	*EventData* is capabilities DWORD passed to SINIT by MLE Preamble TPM 1.2: Extended to PCR18 only
			17 & 18	*Ditto* *TPM 2.0: Extended to PCR 17 & 18*
EVTYPE_SINIT_PUBKEY_ HASH	EB + 16	*Digest*=Hash$_{HashAlgID}$ (PUBKEY_HASH) *EventData*=EmptyBuffer *EventDataSize*=0	18	PUBKEY_HASH is digest of ACM Signer key. It can be retrieved from MSRs 0x20::0x23 NOTE 3
EVTYPE_LCP_HASH	EB + 17	Digest = SHA1(*EventData*) *EventData* = array of digests of lists (authorities) containing matching LCP elements data as filled by LCP. *EventDataSize*= [1::3] * 20=20::60	18	TPM 1.2: Size is variable because digest of each contributing authority is inserted only once regardless of how may matching elements it may contain.

Event Type	Value	Digest and Event	PCR	Comments
		N/A	N/A	Reserved in TPM2.0 mode
EVTYPE_LCP_DETAILS _HASH	EB + 18	N/A	N/A	Reserved in TPM 1.2 mode
		Digest= $Hash_{HashAlgID}$ (*EventData*) *EventData*= concatenation of digests and policy controls of matching policy elements. *EventDataSize*= sizeof(*EventData*) If policy evaluates to ANY, Then *Digest*= $Hash_{HashAlgID}$ (0x00) *EventData*=0x00 *EventDataSize*=1	17	TPM 2.0: *EventData* is concatenation of policy controls and digests of all matching elements contributed to effective policy. Analogous to event type 13 but uses different event format.
EVTYPE_LCP _AUTHORITIES_HASH	EB + 19	N/A	N/A	Reserved in TPM 1.2 mode
		Digest= $Hash_{HashAlgID}$ (*EventData*) *EventData*= concatenation of list descriptors containing matching policy elements. *EventDataSize*= sizeof(*EventData*) If policy evaluates to ANY, Then *Digest*= $Hash_{HashAlgID}$ (0x00) *EventData*=0x00 *EventDataSize*=1	18	TPM 2.0: *EventData* is concatenation of list description structures of all lists containing matching elements contributed to effective policy. Analogous to event type 17 but uses different event format.
EVTYPE_NV_INFO _HASH	EB + 20	N/A	N/A	Reserved in TPM 1.2 mode
		Digest= $Hash_{HashAlgID}$ (*EventData*) *EventData*=array of TPMS_NV_PUBLIC structures of defined indices. *EventDataSize*= sizeof(EventData)	17 & 18	TPM 2.0: *EventData* is concatenation of TPMS_NV_PUBLIC structures of all defined TPM NV indices (AUX & PO).

Event Type	Value	Digest and Event	PCR	Comments
EVTYPE_COLD_BOOT _BIOS_HASH	EB + 21	*Digest*= $Hash_{HashAlgID}$ (S-BIOS) *EventData*="TXT cold boot BIOS" *EventDataSize*= sizeof(*EventData*)	17	TPM 1.2: S-BIOS is Early TXT BIOS verified & measured by S-ACM. SHA1 digest is taken either from BPM or AUX index. *EventData* is string "TXT cold boot BIOS"
				TPM2.0: Ditto. TPML_DIGEST _VALUES structure extended to PCR is populated by digests taken either from BPM or AUX index. Ditto.
EVTYPE_KM_HASH	EB + 22	N/A	N/A	Reserved in TPM 1.2 mode
		Digest= $Hash_{HashAlgID}$ (Key Manifest Signature) *EventData*="Key Manifest signature". EventDataSize= sizeof(*EventData*)	17	*Digest* is analogous to the above entry, but hashed object is signature of key manifest. *EventData* is string "Key Manifest signature"
EVTYPE_BPM_HASH	EB + 23	N/A	N/A	Reserved in TPM 1.2 mode
		Digest= $Hash_{HashAlgID}$ (Boot Policy Manifest signature). *EventData*="Boot Policy Manifest signature". EventDataSize= sizeof(*EventData*)	17	*Digest* is analogous to the above entry, but hashed object is signature of boot policy manifest. *EventData* is string "Boot Policy Manifest signature"
EVTYPE_KM_INFO _HASH	EB + 24	N/A	N/A	Reserved in TPM 1.2 mode
		Digest= $Hash_{HashAlgID}$ (*EventData*) *EventData*=Key Manifest descriptor. EventDataSize= sizeof(*EventData*)	18	*Digest* is analogous to the above entry, but hashed object is key manifest descriptor NOTE 4 *EventData* key manifest descriptor
	EB + 25	N/A	N/A	Reserved in TPM 1.2 mode

Event Type	Value	Digest and Event	PCR	Comments
		EVTYPE_BPM_INFO _HASH	18	*Digest* is analogous to the above entry, but hashed object is boot policy manifest descriptor NOTE 4 *EventData* boot policy manifest descriptor
EVTYPE_BOOT_POL _HASH	EB + 26	N/A	N/A	Reserved in TPM 1.2 mode
		Digest= $Hash_{HashAlgID}$ (*EventData*) *EventData*= ACM_POLICY_STATUS register. EventDataSize= sizeof(*EventData*)	17	*Digest* is analogous to the above entry, but hashed object is ACM_POLICY _STATUS register. *EventData* is QWORD register LT 0x378
EVTYPE_RANDOM _VALUE	EB + 254	N/A	N/A	Reserved in TPM 1.2 mode
		Digest= $Hash_{HashAlgId}$ *(EventData)* *EventData* = YYMMDDWW NOTE 5 *EventDataSize* = 4	17 & 18	Caps PCR value. Executed as part of NPW module only.
EVTYPE_CAP_VALUE	EB + 255	N/A	N/A	Reserved in TPM 1.2 mode
		Digest= OneDigest NOTE 6	17 & 18	Caps PCR value if digest cannot be computed due to limited embedded SW capabilities. No real event log entry will be created for this action.

Note:

1. $ZeroDigest_{HashAlgIDSize}$ is the array of 0x00 bytes with a size corresponding to a hash algorithm. $ZeroDigest_{20}$ is array of 20 0x00 bytes
2. EmptyBuffer is zero length array of bytes.
3. MSRs numbers delivering digest of signer key are not architectural and are not guaranteed to be stable across all future platforms.
4. EVTYPE_KM_INFO_HASH and EVTYPE_BPM_INFO_HASH events creation and respective PCR 18 extends are controlled by a value of *PCR18 Extend* bit (bit 2) of a *PolEltControl* of an MLE LCP element.
5. YYMMDDWW are concatenated RTC bytes representing year, month, day of the month and day of the week, respectively.

6. OneDigest is sequence of bytes: 01,00,00...00 with a total size equal to HashAlgId_DIGEST_SIZE of capped TPM bank

§

Appendix G ACM Hash Algorithm Support

G.1 Supported Hash Algorithms

ACM support will be not restricted to fixed set of hash algorithms. Instead it will support a variable list of algorithms which may include any algorithms supported by TPM 2.0 specification limited only by PRD, space and supporting libraries, and defined on by-project bases.

List of currently supported algorithms is presented in 3.2.5

G.2 Hash Algorithm Lists

Based on the list of TPM 2.0 supported algorithms the ACM creates the following lists of hash algorithms for different usages:

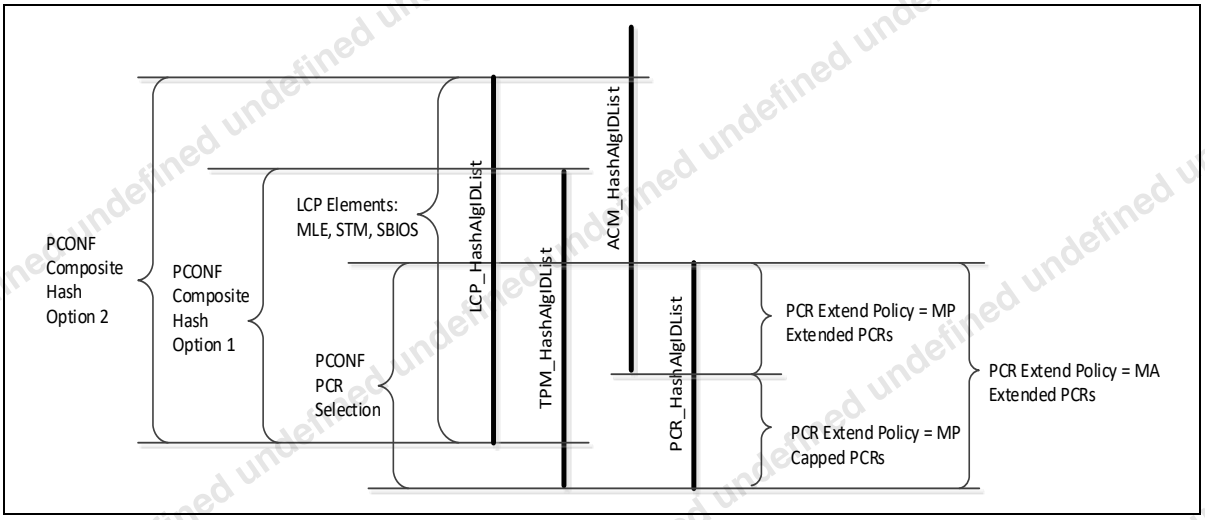
- *TPM_HashAlgIDList* - list of TPM supported *HashAlgIDs*. This is the list of all *HashAlgIDs* supported by given entity of the TPM device. Determined dynamically at run time;
- *PCR_HashAlgIDList* - list of *HashAlgIDs* for which given specimen of TPM device implements dynamic PCRs 17 & 18. Determined dynamically at run time;
- *ACM_HashAlgIDList* - list of ACM Supported *HashAlgIDs*. This is the list of *HashAlgIDs* supported by ACM via embedded SW. It is established at ACM build time and reported via ACM Info Table – see Table 15
- *Lcp_HashAlgIDList* - List of LCP prescribed *HashAlgIDs*. This is the list stored in PO TPM NV index by Platform Owner. This list is described below.

G.3 Hash Algorithm List Subsets

ACM will detect content of all four *HashAlgID* Lists:

1. *TPM_HashAlgIDList* will be detected querying of TPM capabilities.
2. *PCR_HashAlgIDList* will be created as a subset of *TPM_HashAlgIDList*. *PCR_HashAlgIDList* may not coincide with *TPM_HashAlgIDList* because "*TPM Library specification for TPM family 2.0*" allows empty PCR banks.
3. *ACM_HashAlgIDList* is defined by ACM build options.
4. *Lcp_HashAlgIDList* – will be read from PO index. This list will represent Platform Policy.

Figure 11. Hash Algorithm List Selection



Based on the above lists, ACM will create a few derivative lists to handle permissible hashing while performing LCP enforcement:

- EFF_HashAlgIDList = (ACM_HashAlgIDList U TPM_HashAlgIDList) ∩ LCP_HashAlgIDList
  This list contains all hashes supplied by all crypto-engines available for ACM – embedded SW and TPM intersected with effective Platform Policy mask. This list will be used to enforce LCP policy represented by MLE, and STM elements.

Note: Current implementation of ACM does not use TPM based hashing commands and therefore *TPM_HashAlgId* list for practical purposes can be considered empty.

- LCP policy enforcement represented by PCONF elements will be as follows:
  - PCR Selection used by PCONF element will be limited only by PCR banks supported by given TPM (i.e., by PCR_HashAlgIDList);
  - Hash algorithm used to compute composite digest will be handled by one of two rules:
    - If composite hash will be handled by TPM2_PolicyPCR command, permissible algorithm will be one of supported by TPM and Platform Policy. That is:
      LCP_TPM_HashAlgIDList = TPM_HashAlgIDList ∩ LCP_HashAlgIDList.
      This option #1 is currently supported by ACM and LCP tools.
    - If composite hash will be computed by ACM SW, permissible algorithm will be selected by EFF_HashAlgIDList.
      This option #2 is a stretch goal.
- If PCR Extend Policy is selected to be MP, then ACM_PCR_HashAlgIDList = ACM_HashAlgIDList ∩ PCR_HashAlgIDList list will represent all extended PCR banks. All PCRs not included in this list will be capped.

- If PCR Extend Policy is selected to be MA, then all TPM PCRs will be extended and none will be capped.

§

Appendix H ACM Error Codes

After a successful measured launch, the TXT.ERRORCODE register will contain 0xC0000001 value. Failed launches leave other values in this register, which survive warm resets and may be useful for diagnosis and remediation of the failure's cause.

The tables below describe the format of this register as written during CPU-initiated and ACM-initiated shutdowns.

Table of TXT.ERRORCODE values stable across ACM releases, available in prior revisions of this document has been removed as proven to have little value. Instead, Intel will provide an "Error Code" tool accompanied by an Excel spreadsheet and CSV file as part of SINIT release kit. Both will contain error values and relevant descriptions matching released SINIT ACM.

Note: Excel spreadsheet is saved in CSV format to speed-up browsing.

Release kits for released SINITs can be found at http://software.intel.com/en-us/articles/intel-trusted-execution-technology.

Table 30. General TXT.ERRORCODE Register Format

Bit	Name	Description
31	Valid	Valid error when set to 1. The rest of the register contents should be ignored if '0'.
30	External	0 – induced by processor, 1 – induced by external software.
29:16	Type1	Implementation and source specific
15	SW source	0 – ACM; 1 – MLE
14:0	Type2	Implementation and source specific. Provides details about cause of shutdown.

Table 31. TXT.ERRORCODE Register Format for CPU-initiated TXT-shutdown

Bit	Name	Value / Error
31	Valid	1
30	External	0
29:24	Reserved	0
23:16	Extended value	All errors except #0x10 == 0 Error #0x10 = ACM SVN
15	Reserved	0
14:0	Type2	0 = Legacy shutdown (non-TXT-specific).
		1 – 4 = Reserved

Bit	Name	Value / Error
		5 = Authenticated RAM load memory type error
		6 = Unrecognized AC module format
		7 = Failure to authenticate
		8 = Invalid AC module format
		9 = Unexpected snoop hit detected
		0xA = Illegal event or *IllegalProcessorState* – collection of various illegal causes. 1. A CPU reset occurs during AC-mode or post-SENTER and LT.E2STS[RESET.STS] == 0 (i.e., reset was not caused by an LT-shutdown). 2. A non-virtualized INIT event occurs while post-SENTER. 3. LTSX only: During RLP WAKEUP, the RPL thread's value of MSR bit IA32_SMM_MONITOR_CTL[0] (aka MSEG valid) does not match the ILP thread's value. 4. An SENTER, SEXIT, or WAKEUP doorbell is received while post-VMXON. 5. A thread wakes from wait-for-SIPI while some other thread in the same package is in AC-mode. #1 is by far the most common observed in post-Si debug.
		0xB = Invalid JOIN format
		0xC = Unrecoverable machine check condition
		0xD = VMX abort
		0xE = AC memory corruption
		0xF = Illegal voltage/bus ratio
		0x10 = Low SGX security level post 1st SGX instruction:
		0x11-0xFFFF = Reserved

Table 32. TXT.ERRORCODE Register Format for ACM-initiated TXT-shutdown

Bit	Name	Description
31	Valid	Valid error when set to 1. The rest of the register contents should be ignored if '0'.
30	External	= 1– induced by external software.
29:28	Type1 / Reserved	Free for specific implementation
27:16	Type1 / Minor Error Code	Field value depends on Class Code and / or Major Error Code. Several examples are: 1. If Class Code = "TPM Access" and Major Error Code = "TPM retuned an error" – Field value = TPM returned error code TPM returned error code is posted in different format for TPM 1.2 and TPM 2.0 modes.

Bit	Name	Description
		TPM 1.2: If error code is fatal, it occupies bits [23:16] and bit 24 remains clean. For non-fatal error codes lower byte is placed into bits [23:16] and bit 24 is asserted. For instance, error code 0x803 will be translated into 0x103. This is legacy error code representation. TPM 2.0: TPM returned error code is posted as is. 2. If Class Code = "Launch Control Policy and Major Error Code = "Policy Integrity Fail" – Field value = (LIST_INDEX << 6) + Specific Minor Error Code 3. If Class Code = "Range Check Error" – Field value = Index of first range in conflict with another range according to Project Range Table.
15	SW source	0 = ACM; 1 = MLE
14:10	Type2 / Major Error Code	0 – 0x1F = Error code within current class code
9:4	Type2 / Class Code	0 – 0x3F = Class code clusters several congeneric errors into a group.
3:0	Type2 / Module Type	0 = BIOS ACM 1 = SINIT

§

Appendix I TPM NV

Table 33. TPM Family 1.2 NV Storage Matrix

Name	Label	Index Value	Description	Public Size in bytes	Attributes	Read Auth	Write Auth	Locality Write	Locality Read	PCR Write or Read
LCP Platform Owner	PO	0x40000001	LCP Structure for Platform Owner	Platform Specific Size: 54	OWNERWRITE	None	Owner	Any	Any	Any
Launch Auxiliary	AUX	IVB and before platforms: 0x50000002 IVB after PRT and beyond platforms: 0x50000003	Inter – module mailbox BIOSAC registration data	Platform Specific LT-CX Size: 64 LT-SX Size: 96	None	None	None	3 or 4	Any	Any
SGX SVN	SGX	0x50000004	BIOS – MLE mailbox. Value of SINIT SVN and flags	Platform Specific Size: 8	None	None	None	3-0	3-0	Any

Table 34. TPM Family 2.0 NV Storage Matrix

Name	Label	Index Value	Description	Public Size in bytes	Attributes TPMA_NV_	Auth Value	Auth Policy	Locality Write	Locality Read	Locality Delete
LCP Platform Owner	PO	0x1C1_0106 NOTE3	LCP Structure for Platform Owner	Platform Specific Size: >= 38 + HASHALGID_DIGEST_SIZE Note 1	OWNERWRITE POLICYWRITE AUTHREAD NO_DA	Empty Buffer	Policy Owner	Any, controlled by Owner choice	Any	Any, controlled by Owner choice
Launch Auxiliary	AUX	0x1C1_0102 NOTE3	Inter – module mailbox BIOSAC registration data	Platform Specific Size: >= 40 + 2 * HASHALGID_DIGEST_SIZE Note 1	POLICYWRITE POLICY_DELETE WRITE_STCLEAR AUTHREAD NO_DA PLATFORMCREATE	Empty Buffer	Fixed policy Note 2	3 or 4	Any	3 or 4
SGX SVN	SGX	0x1C1_0104 NOTE3	BIOS – MLE mailbox. Value of SINIT SVN and flags	Platform Specific Size: 8	AUTHWRITE POLICY_DELETE AUTHREAD NO_DA PLATFORMCREA	Empty Buffer	OEM policy	Any	Any	Any, controlled by OEM policy

Note: HASH$_{ALGID}$_DIGEST_SIZE is the size of the digest of respective hash algorithm used to store data in the index. Respective sizes in bytes for all used algorithms are:

- SHA1_DIGEST_SIZE = 20
- SHA256_DIGEST_SIZE = 32
- SM3_256_DIGEST_SIZE = 32
- SHA384_DIGEST_SIZE = 48

Note: Policy digest must match the following:

- Let policy A to be:
  A = TPM2_PolicyLocality (Locality 3 & Locality 4)
- Let policy B to be:
  B = TPM2_PolicyCommandCode (TPM_CC_NV_UndefineSpecial)
- *AuthPolicy* can be computed as:
  authPolicy = {A} OR {{A} AND {B}}

Note: Initially, TXT used index handles selected from 0x180_xxxx and 0x140_xxxx ranges based on early version of TCG *"Registry of reserved TPM 2.0 handles and localities"* and TXT legacy. Later revision of TCG registry repurposed the above ranges creating mismatch between TXT practice and normative TCG document. Moreover, selection of TXT related NV index handles from the ranges assigned to OEMs and Users does not guarantee its conflict free usage. In order to address both issues Intel requested TCG to assign "Intel Reserved" range of indices 0x1C1_0100 - 0x1C1_013F for TXT and other purposes out of range 0x1C1_xxxx reserved for component vendors.

Note: TXT ACMs will support only 0x1C1_xxxx index set. Indication of supported set is provided via TPM capabilities – see Table 16 but will always be forced to "1".

§

Appendix J Detailed LCP Checklist

J.1 Policy Validation Checklist

The following checks must be performed by the LCP *Policy Engine* to validate LCP data integrity:

J.1.1 TPM NV AUX Index

1. Index must be provisioned.
2. Index attributes must be per Table 33 for TPM 1.2 index and per Table 34 for TPM 2.0 index.
3. *nameAlg* hashing algorithm must have at least 256-bit digest (e.g., be either SHA256 / SM3 or stronger).
4. Size must be per Table 33 and Table 34 for TPM 1.2 and TPM 2.0 AUX indices, respectively.
5. Read AUX index data and analyze content.
6. Additional verification steps can be added since AUX index content is platform specific.

J.1.2 TPM NV PO Index

1. Index may be provisioned.
2. Index attributes must be per Table 33 for TPM 1.2 index and per Table 34 for TPM 2.0 index.
3. *nameAlg* hashing algorithm must have at least 256-bit digest (e.g., be either SHA256 / SM3 or stronger)
4. Read PO index data and analyze content.
5. *PO.Version* must be per 3.2.1.
6. *PO.HashAlg* must be SHA1 for TPM 1.2 index and one of the supported algorithms per 3.2.5 for TPM 2.0 index.
7. TPM 2.0 mode: *PO.HashAlg* must be permitted by *PO.LcpHashAlgMask*.
8. Size must be per Table 33 for TPM 1.2 index and per Table 34 for TPM 2.0 index. Size of *PO.PolicyHash* field must be equal digest size of *PO.HashAlg* field.
9. *PO.PolicyType* must be either LCP_POLTYPE_LIST or LCP_POLTYPE_ANY – see 3.2.1.
10. *PO.SinitMinVersion* must be not greater than value of *AcmVersion* field in Table 8.
11. TPM 2.0 mode: algorithms corresponding to set bits in *PO.LcpHashAlgMask and PO.LcpSignAlgMask* must be supported by ACM per 3.2.5.

J.1.3 NPW Mode

Detect whether system in NPW mode and is allowed to run:

1. System is in NPW mode if:
   a. NPW bit is asserted in *Flags* field of SINIT header or
   b. Startup ACM ACM_SVN value is < 2 or
   c. If AUX index Startup ACM self-registration area is in initial TPM NV state ("All ones for TPM 1.2 or "All zeros" for TPM 2.0).
2. If system is in NPW mode *PolicyControl.NPW_OK* bit must be asserted – see 3.2.1.
3. If PO index is undefiled, and platform is in NPW mode, SINIT must abort execution.

J.1.4 Policy Data File

1. If *PO.PolicyType* is LCP_POLTYPE_LIST, then respective PO *Policy Data File* must exist and be accessible.

J.1.5 PO Policy Data File if exists

1. Loop through all lists as indicated by *File.NumberOfLists* value
2. Validate signature of each of the lists:
   a. If list is unsigned – skip to step 3 computation of list digest.
   b. If list is signed but signature is not supported by *Policy Engine* – abort since this prevents validation of data integrity.
   c. If list is signed and signature is supported – validate signature. Supported signature algorithms are listed in Appendix D.3.1.
3. Compute the digest of the list:
   a. For unsigned list compute digest as *ListDigest[ListNumber] = PO.HashAlg* (*ListData*), where *PO.HashAlg* is hashing algorithm in PO index per Appendix D.1 and hashing is performed over entire unsigned list data.
   b. For signed list compute digest as *ListDigest[ListNumber] = PO.HashAlg* (*ListPublicKey*), where *PO.HashAlg* is hashing algorithm in PO index per 0 and *ListPublicKey* is either *PublicKey* field of RSASSA signature or concatenation of *Qx|Qy* public key components fields of SM2/ECDSA signature – signature formats are in Appendix D.3.1.
4. Compute digest of entire *PO Policy Data File* by concatenation of all digests of individual lists and hashing it once more using the same *PO.HashAlg* (e.g., *PO.PolicyDataFileDigest = PO.HashAlg(List0Digest[0]|...|ListDigest[N])*
5. Computed *PO.PolicyDataFileDigest* file digest must match value of *PO.PolicyHash* field of PO index.

J.1.6 PO Policy Data File List Integrity

1. *List.Version* must be per Appendix D.3.2.
2. Loop through all elements of the list and sum-up their sizes. Obtained value must match *File.PolicyElementsSize* field value. Empty list with a *File.PolicyElementsSize == 0* is a valid placeholder. It must contain no elements.
3. Validate hash algorithms used by elements. Scan through all elements in list while checking their *Elt.HashAlgorithm* values. Abort if any is not supported by ACM per 0. In TPM 1.2 mode scan only TPM 1.2 style elements while in TPM 2.0 mode scan only TPM 2.0 style elements.
4. In TPM 2.0 mode only, verify that *Count* field of TPMS_QUOTE_INFO structure which is part of PCONF element definition equals 1.
5. If list is signed check the *List.RevocationCounter* value. If it is lesser than *PO.DataRevocationCounter* [*ListNumber*] abort with error. *ListNumber* here is the orderly list number in the *Policy Data File*.

J.2 Policy Enforcement Checklist

J.2.1 Policy type handling by SINIT

1. For SINIT module: if *PO.PolicyType* equals to LCP_POLTYPE_ANY, allow any platform configuration and any MLE and STM to run. Exit policy enforcement logic.

J.2.2 MLE Element Enforcement

1. Start of MLE element scan. Process lists and elements in the lists in the order of appearance. Look for MLE elements. Skip all other element types.
   a. TPM 1.2: Scan all lists looking for TPM 1.2 style MLE elements.
   b. TPM 2.0: Scan all lists only looking for TPM 2.0 style MLE elements.
      i. If list is signed and *List.SignatureAlgorithm* is not permitted by *PO.LcpSignAlgMask*, skip the list.
2. Start of matching loop.
   a. For every MLE type element found do the following:
      i. TPM 2.0: Check *HashAlgorithm* field against *PO.LcpHashAlgMask* value. If not permitted by a mask skip element.
      ii. Indicate MLE element present: record MLE_MATCH_REQUIRED flag;
      iii. Try to match element:
         1. Scan through all digests in element digest array comparing digest with computed MLE digest. It is assumed that MLE digest has been calculated before LCP execution flow.
         2. If match is found

a. Check the value of *Elt.SinitMinVersion* field. It must be not greater than value of *AcmVersion* field in Table 8. If greater abort with error – SINIT is revoked.

b. Check *PolEltControl.STM_is_required* bit of matching element. If asserted but previous execution has not found STM present, exit with error.

c. Save value of *PolEltControl.PCR_Extends* bit – it will be consumed outsize of LCP content by a PCR 18 extend code.

d. MLE policy is satisfied - continue to the next element type.

3. If match is not found yet – continue to the "Start of matching loop".

3. End of matching loop.
4. If this point is reached – no match was found.
   a. If MLE_MATCH_REQUIRED_FLAG is asserted – exit with error.
   b. MLE_MATCH_REQUIRED_FLAG is de-asserted, assumed MLE policy is satisfied – continue to the next element type.

J.2.3 PCONF Element Enforcement

1. Start of PCONF element scan. Process lists and elements in the lists in the order of appearance. Look for PCONF elements. Skip all other element types.
   a. TPM 1.2: Scan all lists looking for TPM 1.2 style PCONF elements.
   b. TPM 2.0: Scan all lists only looking for TPM 2.0 style PCONF elements.
      i. If list is signed and *List.SignatureAlgorithm* is not permitted by *PO.LcpSignAlgMask*, skip the list.
2. Initialize variable. Set PCONF_MATCHES_FOUND = 0
3. Start of matching loop PASS 1.
   a. For every PCONF type element found do the following:
      i. TPM 2.0: Check *HashAlgorithm* field against *PO.LcpHashAlgMask* value. If not permitted by a mask skip element.
      ii. Indicate PCONF element present: record PCONF_MATCH_REQUIRED_FLAG_PASS_1
      iii. Try to match element.
         1. Scan through all *PCRInfos* structures in element array, query PCRs according to PCR Selection structure and compute relevant *Composite Digest* of selected PCR values.
         2. Then compare this digest to value stored in element's *PCRInfo* structure.
         3. If match is found – increment PCONF_MATCHES_FOUND

a. If *PO.PolicyControl.Pconf_Enforced* bit is de-asserted, assumed PCONF policy is satisfied – continue to the next element type
b. If *PO.PolicyControl.Pconf_Enforced* bit is asserted
    i. Skip rest of elements in current list and go to the next list.
    ii. Then go to the "Start of PO matching loop PASS 2"
4. If match is not found yet – continue to the "Start of PO matching loop PASS 1"

4. End of matching loop PASS 1
5. Start of matching loop PASS 2.
    a. Repeat all steps of "Matching loop PASS 1" above with the following changes:
        i. If PCONF element is found, record it in PCONF_MATCH_REQUIRED_FLAG_PASS_2
        ii. After match is found and PCONF_MATCHES_FOUND is incremented
        1. If PCONF_MATCHES_FOUND == 2, PCONF policy is satisfied – continue to the next element type
        iii. If match is not found yet – continue to the "Start of matching loop PASS 2"
6. End of matching loop PASS 2
7. At this point either singe or two matching passes were executed
    a. If *PO.PolicyControl.Pconf_Enforced* bit is de-asserted
        i. If PCONF_MATCH_REQUIRED_FLAG_PASS_1 is de-asserted, assumed PCONF policy is satisfied – continue to the next element type.
        ii. Else PCONF_MATCH_REQUIRED_FLAG_PASS_1 is asserted - exit with error.
    b. If *PO.PolicyControl.Pconf_Enforced* bit is asserted. Only one special configuration entails an error – all other are successes.
        i. If PCONF_MATCH_REQUIRED_FLAG_PASS_1 == 1 and PCONF_MATCH_REQUIRED_FLAG_PASS_2 == 1 and PCONF_MATCHES_FOUND == 1 – exit with error.
        ii. Else assumed PCONF policy is satisfied – continue to the next element type.

J.2.4 STM Element Enforcement

1. If prior execution has not found STM in the system – (based on MSEG enable status – see Table 23) scan of STM elements is not performed. Only if matching MLE element requires STM this will generate an error exit – see Appendix J.2.2.
2. Start of STM element scan. Process lists and elements in the lists in the order of appearance. Look for STM elements. Skip all other element types.

   a. TPM 1.2: Scan all lists looking for TPM 1.2 style STM elements.
   b. TPM 2.0: Scan all lists only looking for TPM 2.0 style STM elements.
      i. If list is signed and *List.SignatureAlgorithm* is not permitted by *PO.LcpSignAlgMask*, skip the list.
3. Start of matching loop
   a. For every STM type element found do the following:
      i. TPM 2.0: Check *HashAlgorithm* field against *PO.LcpHashAlgMask* value. If not permitted by a mask skip element
      ii. Indicate STM element present: record STM_MATCH_REQUIRED flag
      iii. Try to match element:
         1. Scan through all digests in element digest array comparing digest with computed STM digest. It is assumed that STM digest has been calculated before LCP execution flow.
         2. If match is found, STM policy is satisfied – continue to the next task. This is the last element type
         3. If match is not found yet – continue to the "Start of matching loop"
4. End of matching loop
5. If this point is reached – no match was found.
   a. If STM_MATCH_REQUIRED_FLAG is asserted - exit with error since no match was found
6. STM_MATCH_REQUIRED_FLAG is de-asserted, assumed STM policy is satisfied – continue to the next task. This is the last element type

§