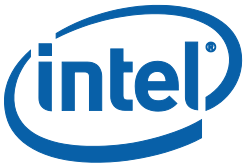


# Intel® Trusted Execution Technology (Intel® TXT)

## Software Development Guide

***Measured Launched Environment Developer's Guide***

***May 2017***

***Revision 014***

Document: 315168-014

# Contents

## Figures



## Tables

# Revision History

| Revision Number | Description | Revision Date |
|---|---|---|
| -001 | • Initial release. | May 2006 |
| -002 | • Established public document number<br>• Edited throughout for clarity. | August 2006 |
| -003 | • Added launched environment consideration<br>• Renamed LT to Intel® TXT | October 2006 |
| -004 | • Updated for production platforms<br>• Use MLE terminology | August 2007 |
| -005 | • Updated for latest structure versions and new RLP wakeup mechanism<br>• Added Launch Control Policy information<br>• Removed TEP Appendix<br>• Many miscellaneous changes and additions | June 2008 |
| -006 | • Miscellaneous errata<br>• Added definition of LCP v2<br>• Multiple processor support | December 2009 |
| -007 | • Miscellaneous errata<br>• Documented ProcessorIDList support<br>• Described CPU Hotplug handling<br>• Updated TXT configuration registers<br>• Documented new TXT Heap structures<br>• Added LCP_SBIOS_ELEMENT<br>• Documented processor and system state after SENTER/RLP wakeup | March 2011 |
| -008 | • Format updates | June 2011 |
| -009 | • Numerous updates from prior author<br>• Added text for LCP details/authorities<br>• Corrected osinitdata offset 84 for versions 6+ | April 2013 |
| -010 | • Corrections to data structures, algorithm detail versus prior versions<br>• Inclusion of TPM 2.0 changes and additions | March 2014 |
| -011 | • Update TPM_PCR_INFO_SHORT structure and TPMS_QUOTE_INFO structure Endianness | May 2014 |
| -012 | • Added SGX requirement for TXT platform<br>• Updated LCP changes of TPM2.0 transitions<br>• Added TCG compliant TXT event log formats<br>• Documented TPM NV definitions<br>• Inclusion of detailed LCP checklists | July 2015 |
| -013 | • Added each MTRR base must be multiple of that MTTR size Prior to GETSEC[SENTER] execution. | August 2016 |

| Revision Number | Description | Revision Date |
|---|---|---|
| -014 | • Added changes related to Converged BtG / TXT technologies and CNSS Advisory 02-15 | May 2017 |

# Foreword

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

To simplify document all obsolete features no longer supported due to convergence are removed from it starting from revision 014. If needed to review removed information Reader is recommended to consult revision 013.

# 1 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 such that the result (a) is unique and (b) indicates 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. A cryptographic hash algorithm 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:

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.

Then, the OS loads and launches a different MLE.

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 through the use of 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 – BIOS AC module used for performing of subordinate tasks such as TXT Opt-in preparation, clearing of 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 Chapter 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

Similar to 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 TPMM 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 family of devices 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 of an object into $PCR_n$" implies hashing of that object using each of the three hashing algorithms and extending obtained digests into $PCR_n$ of all of the 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 of the 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 a MLE decision, and will be communicated to the launched ACM via the “flags” field in Table 19.

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 of the algorithms supported by the device.

In TPM1.2 mode certain values are extended into PCRs without hashing. Some of them are extended this way historically; other are using extends of zero digests or constant values as an indication of various platform states or events. Here zero digest is array of 20 zeros: {0,0,…,0}

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

In TPM2.0 this practice is discouraged and simply cannot be supported when maximum agility (MA) extend policy is enforced – see Table 19. Therefore in all above cases we will not extend constant values as is but 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/authorities 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 section C.5.

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 SINIT module itself by means of _HASH_START/_HASH_DATA/_HASH_END HW event sequence (will be denoted as _HASH_*). This event sequence is supported by all of the TPM families and is described both in *TPM Main* and *TPM Library specifications*

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 G.

### 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 2). 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 2).

**Note:** in the following sections DIGEST value is obtained as a result of hashing of respective data. Used hash algorithm is SHA1 for TPM 1.2 or determined by respective PCR bank for TPM 2.0.

#### 1.10.2.1 PCR 17 (Details)

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

PCR 17 is initialized using the _TPM_HASH_* sequence. 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 also in *SinitMleData.EdxSenterFlags*). As part of this sequence, PCRs 17-23 are reset to 0 before _TPM_HASH_START. The digest of SINIT is also stored in the *SinitMleData.SinitHash* field of the SINIT to MLE data table (*SinitMleData* – see section C.5).

Result of _HASH_* differs from one CPU family to another. Originally Intel ® CPUs were computing SINIT digest using SHA1 producing 20 bytes output. Newer Intel ® CPUs were computing SINIT digest using SHA256 producing 32 bytes output. As a result *SinitMleData.SinitHash* field defined as 20 bytes long could not comprise computed digest.

To cope with this situation, *SinitMleData* has been redefined. When SINIT is run by earlier processor it creates *SinitMleData* table revision 6 in which *SinitMleData.SinitHash* field contains actual digest of SINIT. When SINIT is run by newer processor it creates *SinitMleData* table revision 7 or higher in which

*SinitMleData.SinitHash* field contains value of PCR 17 after the initial extend operation (see below for more details).

After initial SINIT measurement PCR 17 is extended with the DIGEST values of the following items concatenated in this order:

The following hashes are extended to PCR17 in the order given:

- BIOS AC registration info retrieved from AUX Index. In TPM 1.2 mode, 20 bytes of that data, in TPM 2.0 mode, DIGEST of 32 bytes of that data.
- DIGEST of Processor S-CRTM status coded as DWORD.
- DIGEST of *PO.PolicyControl* field of used policy retrieved from PO index coded as DWORD
- DIGEST of all matching elements used by the policy. If there is no policy used, for 1.2 family, this digest is zero; for 2,0 family, this is DIGEST(0x0)
- DIGEST of STM. If STM is not enabled, for 1.2 family, this digest is zero; for 2.0 family, this is DIGEST(0x0)
- DIGEST of *Capability* field of *OsSinitData* table, coded as DWORD
- DIGEST of MLE.

#### 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 following hashes are extended to PCR18 in the order given:

- DIGEST of public key modulus used to verify SINIT signature.
- DIGEST of Processor S-CRTM status coded as DWORD – same value as extended to PCR17.
- DIGEST of *Capability* field of *OsSinitData* table, coded as DWORD – same value as extended to PCR17.
- DIGEST of *PO.PolicyControl* field of platform owner (PO) policy coded as DWORD – same value as extended to PCR17.
- DIGEST of LCP – DIGEST of concatenation of hashes of lists containing matching elements. If no policy, for 1.2 family, this digest is zero; for 2.0 family, it is DIGEST(0x0)

## 1.11 DMA Protection

This section briefly describes the two chipset mechanisms that can be used to protect regions of memory from DMA access by bus master 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 as long as it does not conflict with either the 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) in order 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 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 1 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 1. 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 2. MLE/SINIT Capabilities Field Bit Definitions**

| Bit position | Description |
|---|---|
| 0 | Support for GETSEC[WAKEUP] for RLP wakeup<br>All MLEs should support this.<br>1 = supported/requested<br>0 = not supported |
| 1 | Support for RLP wakeup using MONITOR address (*SinitMleData.RlpWakeupAddr*)<br>All MLEs should support this.<br>1 = supported/requested<br>0 = not supported |

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

**Note:** Legacy TBOOT computes MTRR masks assuming 36 bits width of 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 3-1: 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 a number of 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 of the 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 **Error! Reference source not found.Error! Reference source not found.**

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 from disk into memory. 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 4.3.1.

The 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 KBytes of physically contiguous memory is allocated for the purpose of loading the SINIT AC module; this has increased to 320 Kbytes for latest generation processors. System software must use this region for any SINIT AC module that it loads.

The SINIT AC module must be located on a 4 KBytes 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 KBytes of memory beyond the end of the SINIT AC image. See the GETSEC[ENTERACCS] instruction and the Authenticated Code Module Format, section 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 below 4 GB. 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 a failure of the AC module to complete normal execution and an Intel® TXT Shutdown.

### Listing 3. AC Module Matching Pseudocode

```
TXT_ACM_HEADER               *AcmHdr;        // see Table 5.
TXT_CHIPSET_ACM_INFO_TABLE   *InfoTable;     // see Table 7

//
// 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 (TXT.VER.FSBIF != 0xFFFFFFFF) {
25.     IF (AcmHdr->Flags[15] == TXT.VER.FSBIF[31]) {
26.        Fail: production flags mismatch;
27.     }
```

```
28.  }
29.  ELSE IF (AcmHdr->Flags[15] == TXT.VER.EMIF[31]) {
30.        Fail: production flags mismatch;
31.  }

//
// Match AC module to system chipset
//
TXT_ACM_CHIPSET_ID_LIST   *ChipsetIdList;    // see Table 8
TXT_ACM_CHIPSET_ID        *ChipsetId;        // see Table 9

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 10
TXT_ACM_PROCESSOR_ID        *ProcessorId;       // see Table 10

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 1) that corresponds to the *Capabilities* field in the SINIT AC module Information Table (see Table 7).

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 GBytes.

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 KByte 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 will use this page table after launch then 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 Appendix C)
- Initialize contents of the *OsMleData* (see C.3) and *OsMleDataSize* (with the size of the *OsMleData* field + 8H) fields.
- Initialize contents of the *OsSinitData* (see section 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 all of the 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).

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

**Error! Reference source not found.** 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 has been reserved for this purpose (see section **Error! Reference source not found.**), 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 state 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. Make sure each variable MTRR base must be 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 11K bytes in size, an 8K and a 4K or three 4K MTRRs should be used to map it, not a single 32K MTRR. Any unused variable range MTRRs should have their valid bit cleared. If the 8$^{th}$ 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

**Error! Reference source not found.** 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 would 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.  32.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 the 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 then 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 of the instructions and data that are used before paging is 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 a 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. In 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 uses 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 to 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 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 F 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 4K byte 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 section 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 through the use of 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, and affects on, the system should be fully understood before allowing an untrusted VM to access them.

### 2.4.6 Protecting Secrets

If there will be 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 a 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:

1. New processors are released from reset. They execute measured BIOS code.
2. BIOS configures the new processors. At the end of configuration, BIOS clears the BSP flag in the APICBASE register of new processors and leaves them in a CLI/HLT loop.
3. The MLE is notified of this event via the standard ACPI mechanism.
4. 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.
5. 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.
6. The MLE issues the INIT–SIPI-SIPI sequence to each newly added logical processor to take control of these processors.
7. 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 not all processors reach the rendezvous in the guest, the ILP may timeout and VMCALL to the MLE teardown routine. If not all processors 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 actually 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 then it must be torn down before VMX is disabled. See section 25.15.7 of the *Intel 64 and IA-32 Software Developer Manual, Volume 3B 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 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 a specified criteria. Policies may be defined by the Platform Owner, and also leveraged from Boot Guard (BtG) verifications.

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. The discussion below is separated into coverage of TPM1.2 (referred to as LCP V2) of TPM2.0 (LCP V3) in the respective subsections.

## 3.1 Overview

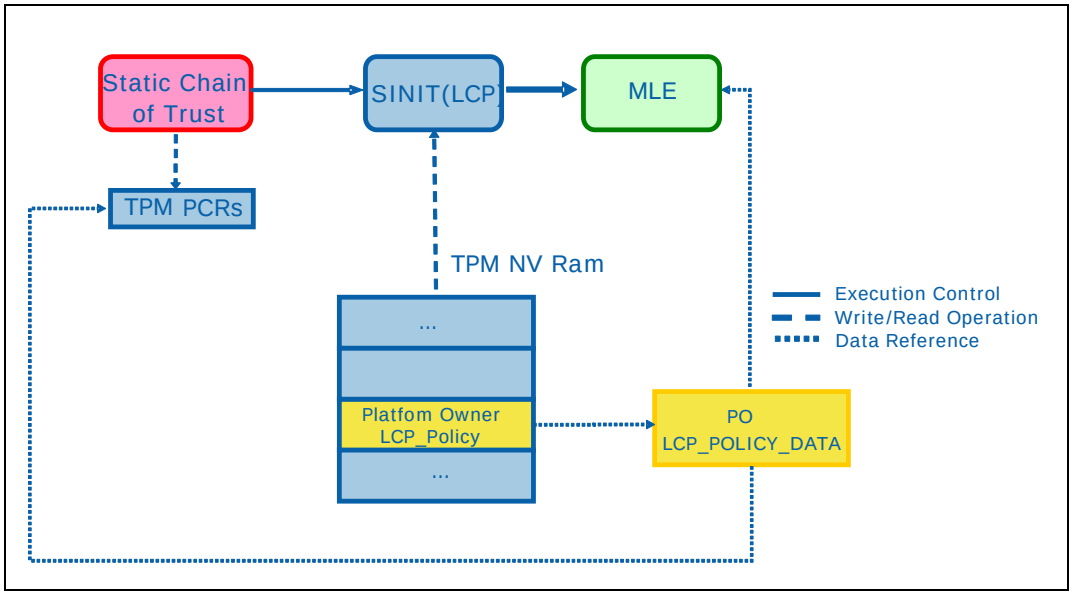
The Launch Control Policy architecture consists of the following components:

- 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 collections of lists, and valid policy elements, such as measurements of MLEs or platform configurations.

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 two sections 3.2 and 3.3 provide detailed description of LCP components and data structures they operate. There are two versions of LCP structures: version 2 (V2) used in TPM 1.2 mode and version 3 (V3) applicable to TPM 2.0 mode of operation.

Section 3.2 contains general description of LCP architecture as applied to both modes of operation but predominantly refers to V2 structures.

Section 3.3 is dedicated to TPM 2.0 mode of operation. It describes architectural deltas of LCP in this mode and operates with V3 structures.

Early version 1 of LCP components is no longer supported by TXT architecture.

Despite that in general both versions of LCP structures are disjoint, there is one cross-section area where LCP engine operating in TPM 1.2 mode must be aware of V3 structures. As will be shown in section 3.3.4.1 V3 policy lists are allowed to contain mixture of V2 and V3 policy elements. Therefore policy engine operating in TPM 1.2 mode must be able to (a) recognize and browse V3 policy lists, (b) validate all types of V3 style signatures, and (c) be able to compute hashes of such lists for the purpose of verifying the hash value of stored LCP policy.

## 3.2 LCP Components. General provisions, V2

The policy that the SINIT AC module implements is named as Platform Owner (PO) policy. PO policy is stored in the non-volatile store 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: Auxiliary TXT index (AUX) and Software Guard Extension index (SGX).

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 3.5
- Digests of various components passed from BIOS AC to SINIT

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 section **Error! Reference source not found.Error! Reference source not found.Error! Reference source not found.**) 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 Structure**

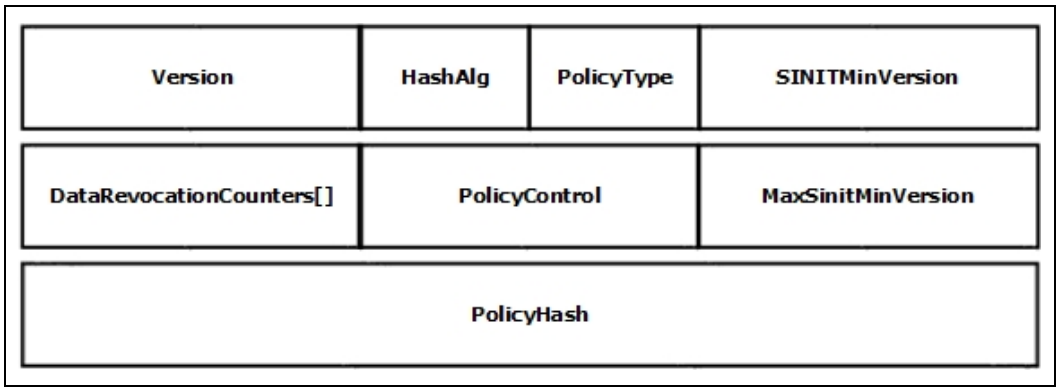



Figure 2 diagrammatically illustrates main fields of the *LCP_POLICY* structure (fields not to scale):

*Version* specifies the version of the *LCP_POLICY* structure 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 version are not guaranteed to be backwards compatible with each other and so SINIT will fail to launch if it finds a major version that it is not compatible with. The version of the *LCP_POLICY* structure defined here is 2.4 (204H) – see section 3.4.2.

*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 7). 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 a *LCP_MLE_ELEMENT* element in a policy then the *SINITMinVersion* in that element will be combined with the value in the *LCP_POLICY*, per the description in section D.4.1. This is done to allow MLE supplier to request 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 LCP *Policy Data File*. This array provides a mechanism to revoke signed lists in that object.

The *DataRevocationCounters* field specifies, for each *LCP_POLICY_LIST*, the minimum counter value, from the *RevocationCounter* field of that list, 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.
For LCP_POLICY_LISTs that are not signed, the corresponding *DataRevocationCounters* index will be ignored.
For each LCP_POLICY_LIST #I that is signed, it must set the *DataRevocationCounters[#I]* element at the index corresponding to its own index in *PolicyLists[#I]*. E.g. if *PolicyLists[0]* is signed, *PolicyLists[1]* unsigned, and *PolicyLists[2]* signed, then the revocation counter for *PolicyLists[0]* will be *DataRevocationCounters[0]*, the counter for *PolicyLists[2]* will be *DataRevocationCounters[2]*, and *DataRevocationCounters[1]* will be ignored. Values in indices greater than the number of lists will be ignored.

The *PolicyControl* field contains policy bits with global LCP impact. Its content is deciphered in section D.1. Field is essentially simplified – only two bits are left:

- 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 section K.2.3
- 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].

**Note**: 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 this policy will allow a revocation utility to set the AUX.AUXRevocation.SinitMinVer to.
"0" in this field means the SINIT minimum version may not be changed. "0xff" allows unconditional changes to the SINIT minimum version.

*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.

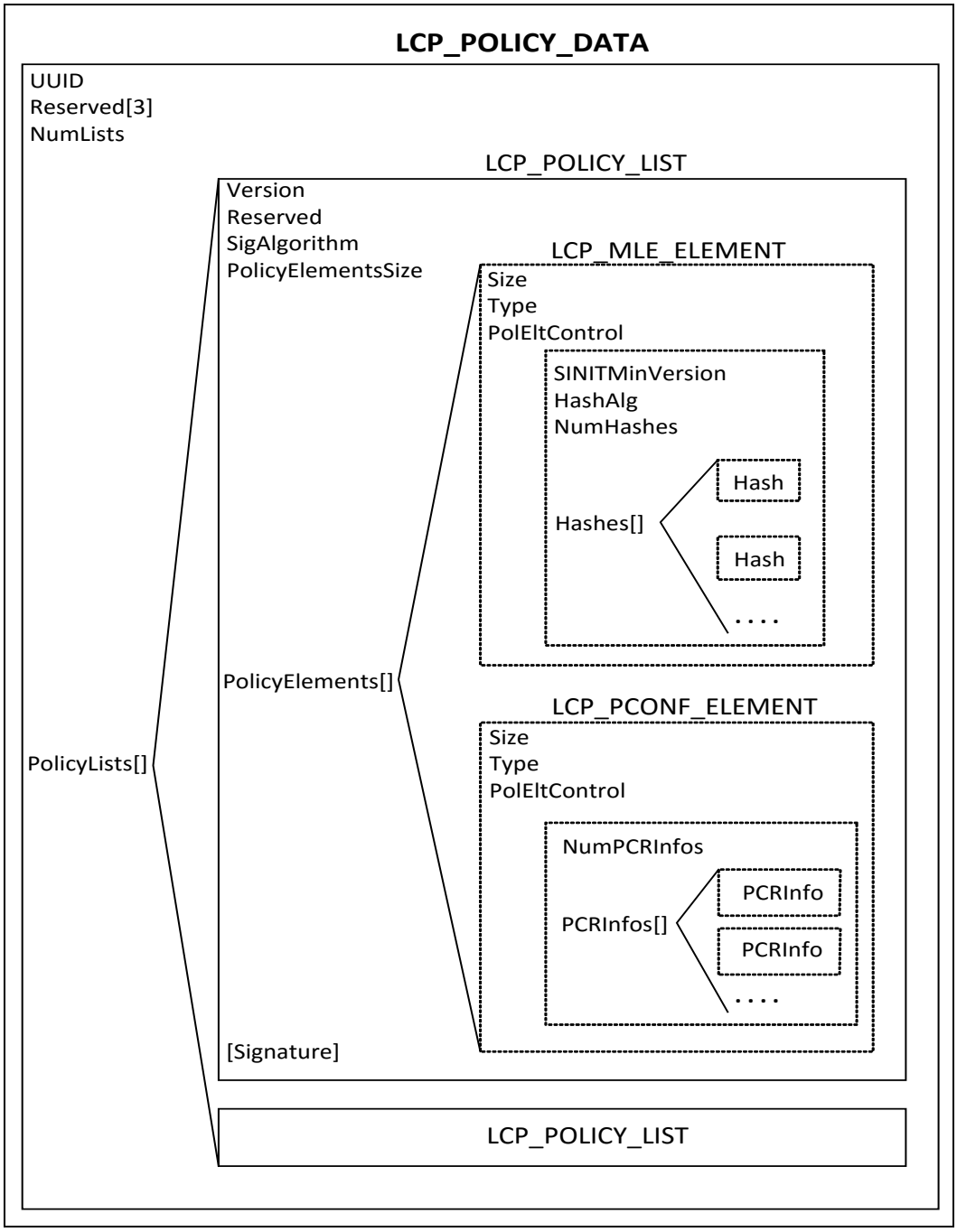
The measurement of a policy list depends on whether the list is signed. For a list signed using any algorithm supported by the ACM, the measurement is the hash (as specified by the *HashAlg* field of the corresponding *LCP_POLICY*) of the public (verification) key in the *PubkeyValue* member of the *Signature* field for RSA, or the Qx and Qy public coordinates for elliptic curve signatures. For unsigned lists (*LCP_POLSALG_NONE*), the measurement is the hash (also as specified by the *HashAlg* field of the corresponding *LCP_POLICY*) of the entire list (*LCP_POLICY_LIST*).

The value of the *PolicyHash* field will be the hash of all of the 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 SHA1(SHA1(list)).

### 3.2.2 LCP Policy Data

The purpose of the *LCP_POLICY_DATA* structure is to provide the 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* for version V2 can be found in Appendix D.

**Figure 3. LCP_POLICY_DATA Structure**




The *PolicyLists[]* field allows the object to contain a number of lists. A 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.).

### 3.2.3 LCP Policy Element

Policy elements are the self-describing entities that contain the actual policy conditions. Since they are self-describing, policy engine can ignore the elements that it doesn't understand or support. This allows for adding new element types without breaking backwards compatibility.

**Figure 4. LCP_POLICY_ELEMENT structure**

| Size | Type | PolEltControl | Data[Size - 12] |
|---|---|---|---|

Figure 4 diagrammatically 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. A policy engine can use this to skip over an element that it does not understand or support.

The *PolEltControl* provides a number of 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 and is defined as:

- Bits31:16 Reserved for element type –specific uses and should be set to zero
- Bits15:2 Reserved and should be set to zero
- Bit 1 if set to 1 requires STM to be enabled. Defined in *LCP_MLE_ELEMENT* structure only and is reserved in all other element type structures.
- Bit 0 Reserved and must be set to 0

The contents of the *Data[]* field are dependent of the type of the element and are described for each type in the subsections of section D.4.

### 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 that the TPM NV must be updated to increment the revocation counter). This allows updates to be 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 needs to be kept secure and under the control of the key owner at all times.
- The private signature key needs to be strong enough for the full lifetime of the policy [We have estimated up to seven years].

### 3.2.5 Supported Cryptographic Algorithms

The following algorithms are defined for version V2 of the Launch Control Policy:

Hashing – SHA1

Signature – RSA PKCS V1.5

As version V2 and V3 elements may both be present in a V3 list being processed in TPM 1.2 mode, all V3 style list signatures supported by the ACM will be validated.

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, the policy will fail and, depending on the ACM evaluating the policy, the environment will not be permitted to launch or the platform will not boot.

## 3.2.6 Policy Engine Logic

### 3.2.6.1 Policies

Before evaluating a policy, the policy engine must first verify the policy's integrity. For policy of type LCP_POLTYPE_LIST, the engine must verify each LCP_POLICY_LIST in the LCP_POLICY_DATA. In the signed list case, the signature must be verified. 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 that was read from the TPM NVRAM.

The policy engine must scan the policy for each policy element that 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 which depends on the *PO.PolicyControl.Pconf_Enforced* enforcement flag per description in K.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 in order 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 and the second containing only a LCP_PCONF_ELEMENT, then the MLE being launched must appear in the first list's *LCP_MLE_ELEMENT* and the current platform configuration must satisfy the second list's LCP_PCONF_ELEMENT; otherwise the launch will fail.
>
> If SINIT is processing a policy that contains two lists, each containing only an LCP_MLE_ELEMENT element, then the MLE being launched must appear in at least one list's LCP_MLE_ELEMENT. Since no other element types are present, any other platform condition or state is acceptable (e.g. any PCR values).

### 3.2.6.2 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 that 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 were not done, such an MLE could get access to data that were sealed against some known policy by another MLE.

As a matter of integrity, the *LCP_POLICY.PolicyControl* field will always be extended into PCR 17. If there are no policies, 32 bits of 0s will be extended.

### 3.2.6.3 No Policy Data

When no policy is executed (includes all ACMs per above; there may be Platform Owner policy but none of its policy elements are understood by the ACMs' policy engines or it contains no policy elements), 20 bytes of 0s will be extended to PCR 17.

### 3.2.6.4 Allow Any Policy

If an Owner policy exists and is of type *LCP_POLTYPE_ANY*, then 20 bytes of 0s will extended to PCR 17.

### 3.2.6.5 Policy with LCP_POLICY_DATA

Because the measurement may contain the measurements of more than one policy list, it is important that the 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, LCP_POLTYPE_STM. If additional policy element types are supported in the future, their evaluation order will be specified.

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 and this allows substitute attack when lists are swapped.

The value that will be extended to PCR 17 will be the hash of the concatenation of all policy list measurements used.

If an Owner policy exists and is of type LCP_POLTYPE_LIST, then the value extended to PCR 17 is calculated as described above.

### 3.2.7 Platform Owner Index

The Platform Owner policy index is intended to represent the policy defined by the owner of the platform, and as such should be provisioned with access control permissions that enforce that control over the policy remains with the platform owner.

The major required index properties are:

Read Locality: Any

Read authorization: None

Write authorization: Owner policy

Complete index provisioning data can be found in Table 32

The following discussion is TPM 1.2 specific.

The simplest access control setting that maintains platform owner control is to set the Write Auth to Owner. However, this also means that updating the policy requires the owner authorization. As the owner authorization can be used for almost all TPM management operations, it may be desirable to limit its use.

Another possibility would be to use a separate authorization just for this index. That would eliminate having to provide the owner authorization just to update the policy.

If trusted software, running in the context of the MLE or with access to TPM locality 2, needs to be able to update the policy, then it would be possible to have no Write Auth but set the Write Locality to 2. This would permit whatever software was able to successfully launch to update the policy.

Note that it is not advisable to use PCR Write controls, since it would mean that the specified PCR could not change over time (e.g. if the software measured into it was upgraded). This is because the index attributes cannot be changed once the index is created.

## 3.3 LCP Components. V3 Deltas

The changes required from version V2 to effect launch control in TPM 2.0 mode are defined and described in this section. Complete set of definitions can be found in Appendix E

### 3.3.1 TPM NV RAM

Launch Control Policy in TPM 2.0 mode will continue to use the legacy TPM NV RAM index names of AUX and PO. Purposes of these indices remain the same but attributes, sizes and layout change. Properties and owning authorities change as well, as described below. Complete index provisioning data can be found in Table 33

#### 3.3.1.1 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 from among those supported by the TPM (*TPM_HashAlgIDList* from H.3) that meet cryptographic strength constraints for 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 an index vulnerable.

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

#### 3.3.1.2 Platform Owner Index

Platform Owner (PO) index purpose and usage remains the same as in V2 LCP – see section 3.2.7, but its definition for TPM 2.0 must be changed. New PO index attributes are specified in Table 33

PO index size depends on the hash algorithm specified in its *HashAlg* field (see Appendix D.1 and Appendix **Error! Reference source not found.**).

### 3.3.2 LCP Policy 2

LCP_POLICY structure defined in 3.2.1 (renamed as LCP_POLICY2) has been modified from the V2 version as described here. For a full listing, see Appendix E.

*Version* is now 3.2 (302H).

*HashAlg* grows to 16 bits and now uses TPM2.0 numeric values.

Two new fields *LcpHashAlgMask* and *LcpSignAlgMask* are defined. Then specify hash and signature algorithm sets Platform Owner tells SINIT to grant.
If LCP engine encounters list with unauthorized signature algorithm – it will abort.
If it encounter elements using unauthorized hash algorithms - it will skip them not allowing to match but it will record that given element type exists and will require a match for this element type to be satisfied upon the scan end.

*PolicyControl* has the same basic content as specified in section **Error! Reference source not found.**

### 3.3.3 LCP Policy Data

This structure matches that for V2, but the *LCP_POLICY_LIST* element has been replaced by *LCP_LIST* – see E.2.1, a union of *LCP_POLICY_LIST* and *LCP_POLICY_LIST2*. This allows V3 *Policy Data File* to contain a mixture of V2 and V3 policy lists.

### 3.3.4 LCP Policy Elements

Additional policy element types have been defined for TPM2.0 mode. These new element types are enhancements to the TPM1.2 elements with augmentations for algorithm agility. These new elements are defined in Appendix E.

For *PolEltControl*, please see 3.2.3 for more information, it is the same as for V2 structure.

#### 3.3.4.1 LCP Element Structures

The LCP structures used for TPM 1.2 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, allows to maintain the same number of authorities (8) since no authority will have to supply separate lists for V2 and V3 elements, and simplifies processing logic.

Where possible, the new element structures use constants as defined in the TCG 2.0 specification. See Appendix E.

### 3.3.5 List Signatures

*HashAlgIDs* accepted in RSA and SM2 signatures will be limited to those included in the *LcpHashAlgMask* defined in Appendix E.1. That means that if for instance RSA signature uses SHA256 to prepare encoded message, SHA256 must be permitted by *LcpHashAlgMask.*

### 3.3.6 PCR Extend Policy

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

- *Extend* command allows to extend already computed digest(s) into one or several selected PCR banks;
- *Event* and *Event Sequence* commands allow sending of 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 when selected 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 C.4

#### 3.3.6.1 SINIT Policy Selection

ISVs supplying MLE solutions can retrieve information of SINIT supported embedded SW algorithms from TPM Info List Table (see Table 11) and also 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 of the TPM PCR banks are supported by SINIT embedded SW, MP *Extend Policy* can be chosen. Otherwise MA Extend Policy has to be a choice. Chosen policy may be delivered to MLE via configuration setting.

### 3.3.7 V3 Policy Engine Logic.

V3 policy engine logic is successor of principles defined for V2 engine and described in section 3.2.6. Obvious differences are related to TPM algorithm agility peculiar to TPM 2.0 family of devices, new format of LCP structures, and new data combining possibilities. The following sections will describe deltas in policy engines behavior.

#### 3.3.7.1 General V3 LCP Evaluation Principles

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 H. Using of these lists for evaluation of different element types is described in the following sections.

##### 3.3.7.1.1 Evaluation of MLE and STM policy elements

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

Use of list is not *Extend Policy* dependent.

#### 3.3.7.1.2 Evaluation of PCONF LCP Elements

The evaluation of PCONF elements depends on TPM capabilities, and will be limited only by PCR banks supported by given TPM i.e. by *PCR_HashAlgIDList* as described in **Error! Reference source not found.**.

Hash algorithm used to compute composite digest will be also determined by *LCP_TPM_HashAlgIDList* as described in **Error! Reference source not found.**, 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.

### 3.3.7.2 Policies

V3 policy engine supports the same polices as its V2 counterpart – see section 3.2.6.1

## 3.3.8 Measuring the Enforced Policy

Effective LCP hashes for V2 structures are described in section 3.2.6.5

The updated V3 LCP structures described in Appendix E require changes in how effective LCP hashes are computed for extension into PCR 17 and 18.

### 3.3.8.1 Effective LCP Policy Details

The Effective LCP Policy Details Hash represents all of the elements contributing to the currently established policy. As in V2, this is a hash of the concatenation of descriptors per element type, extended into PCR 17. However, the size of the digests encountered may vary, and it is no longer possible to allocate a single fixed size for all of the descriptors.

The following data structures address this issue.

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

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

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

For each of the elements involved, this last structure will be initialized to {1, *EffPolicyControl*, *EffEltDigest*} if matched, or {0} if not. The concatenation to be hashed will then be:

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

And the value extended into PCR 17 will be:

$$LcpEffPolicyDetails = HASH_{HashAlgID}(Concat)$$

***Note:*** Concat expression above contains $E\_E\_D_{PCONF1}$ and $E\_E\_D_{PCONF2}$ entries. They are placeholders for matching PCONF elements if *PO.PolicyControl.Pconf_Enforced* bit is set per K.2.3. If it is not set, $E\_E\_D_{PCONF2}$ will shrink to “not present” element indicator.

If LCP policy was “ANY”, a single byte of “0” will be measured into PCR 17 instead.

### 3.3.8.2 Effective LCP Policy Authorities

The Effective LCP Authorities Hash provides aggregated information about all of the 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. If the LCP Policy evaluates to “ANY”, a single byte of “0” will be measured into PCR 18 instead.

#### 3.3.8.2.1 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_RSADSA, TPM_ALG_ECDSA or SM3.

*HashAlg* is the algorithm paired with the signature algorithm used to compute the 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 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 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;
} LIST_DSCR;
```

#### 3.3.8.2.2 LcpEffPolicyAuthoritiesData byte stream

The method for constructing the data byte stream is straightforward. For each authority, if it contains a matching element, its representation will be derived from *LIST_DSCR*; if not, an empty buffer will represent it.

For signed lists, *EffListData* is constructed by concatenating:

- The signature algorithm ID
- The hash algorithm ID associated with the signature
  - For RSASSA this is the hash algorithm used to create the encoded message—it is extracted from the encoded message during signature verification in DER-encoded form, and then converted to its TPM2.0-compliant 16-bit ID.
  - For SM2 this is the hash algorithm used to create the message digest, which is SM3.

***Note:*** SINIT does not support SM2 signatures with any hashing algorithm but S3.

- The *PubkeySize* taken from the LCP_SIGNATURE2 structure
- *EffAuthDigest* as computed according to 3.3.8.2.1

For unsigned lists, *EffListData* is constructed by concatenating the signature algorithm ID TPM_ALG_NULL and the *EffAuthDigest* data as computed according to 3.3.8.2.1.

These representations (*EffListData*) are then concatenated:

$$LcpEffPolicyAuthoritiesData = EffListData_{MLE} \mid EffListData_{PCONF1} \mid EffListData_{PCONF2} \mid EffListData_{STM}$$

***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 K.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.

## 3.3.9 Effective TPM NV info Hash

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:

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

The data extended into PCR 17 and 18 will be:

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

## 3.4 Combined Policy Engine Processing Logic

The new structures defined for LCP V3 require changes in the processing engine logic for both TPM1.2 and TPM2.0 mode. They are enumerated in this section.

In the following description legacy structures represent version 2 of LCP while new structures represent version 3 of LCP (are denoted as V2 and V3 style structures respectively.

### 3.4.1 Overall Topological Changes

1. V3 style LCP *Policy Data Files* can contain combination of V2 and V3 policy lists
2. V3 style policy lists can contain combination of V2 and V3 policy elements. Purpose of such combining is to allow *Authorities* to maintain single *Policy Data Files* covering TPM1.2 and TPM2.0 modes of execution.
3. V2 style lists can contain only V2 style policy elements.

### 3.4.2 Processing of Policy Data Files

1. De-jure we recognize V2 and V3 *Policy Data Files* but de-facto structures of these files are identical. The only what makes *Policy Data File* to be V3 is presence of V3 style lists. Policy Engine logic will not differentiate V2 and V3 style files.
2. Policy Engine in TPM1.2 mode will process V2 and V3 policy lists looking for V2 style elements only. All unrecognized elements including all V3 elements are ignored in this mode;
3. Policy Engine in TPM2.0 mode will process only V3 policy elements. All unrecognized elements including all V2 elements are ignored in this mode.
4. Policy Engine processes data file twice – first time to perform integrity verification (validation), second time to enforce (evaluate) policy.
5. During integrity verification phase Policy Engine will process all of the lists in file in the order of appearance.
6. During policy enforcement phase Policy Engine will process file lists and elements in the following order:
    a. Policy lists in are processed in the order of appearance.
    b. Elements are sought for match in the following order: MLE; PCONF; STM
    c. Elements of each type will be evaluated in the order of appearance in *Policy Data File*, irrespective to hash algorithms they are using.
7. Any integrity error causes platform reset.

8 There are three possible results of policy enforcement:

a. Successful policy evaluation a.k.a. all required matches found. Control is passed to further module tasks;

b. 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 module tasks;

c. No match error. Platform is reset

## 3.4.3 TPM 1.2 mode

### 3.4.3.1 Supported Cryptographic Algorithms

There are no changes in supported algorithms.

### 3.4.3.2 Integrity Verification

In TPM1.2 mode logic of engine will remain the same as today but one not customer visible change is required.

Since both legacy and new lists will be present in updated policy file, and since version of V3 style lists is promoted it will be not recognized by TPM1.2 compatible engine. It will stop evaluation and will abort with error when such list is met.

To prevent this TPM1.2 evaluation logic must be changed to recognize new V3 style lists and evaluate them seeking legacy V2 style elements only. This also implies that engine in TPM1.2 mode must be able to validate all forms of signatures supported in TPM2.0 mode.

Policy Engine must validate all kinds of signatures including those supported in V3 style lists. Policy engine must exit with error if signature cannot be validated by any reason.

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

### 3.4.3.3 Policy Enforcement

There are no differences in legacy policy enforcement.

## 3.4.4 TPM 2.0 Mode

### 3.4.4.1 Supported Cryptographic Algorithms

1. There are two sources of supported crypto-graphical algorithms for signature verification and hashing – embedded SW and TPM. List of currently supported algorithms is presented in E.1

2. For the sake of performance TPM based services must be used only if embedded SW lacks required support.

### 3.4.4.2 Integrity Verification

#### 3.4.4.2.1 TPM NV Indices

Policy engine will perform the same verification steps as in TPM 1.2 mode and some additional checks outlined below.

1 *nameAlg* strength check.

   a. When each of the indices is verified, Policy Engine will compare *nameAlg* of the index to the minimal required and abort with error if it has lesser than required strength.

   b. Minimal required *nameAlg* is SHA256 or SM3 if they are supported by current TPM. If the only TPM supported algorithm is SHA1, it will be accepted by Policy Engine and will not generate an error.

2 LCP Policy Engine must ensure that *PS.PolicyControl.AUXDeletionControl* = 0. If not – it must abort with error.

3 Policy Engine must ensure as a matter of integrity:

   a. *PO.LcpHashAlgMask, PO.LcpSignAlgMask* are not empty;

      i. Each bit set in a *PO.LcpHashAlgMask* mask means that relevant hash algorithm is permitted.

      ii. Each bit set in *PO.LcpSignAlgMask* mask means that relevant combination of signature algorithm, key size, hash algorithm and curve is permitted.

   b. All hash algorithms permitted by *PO.LcpHashAlgMask* and *PO.LcpSignAlgMask* are supported by ACM – see E.1 and E.3.1;

   c. *PO.HashAlg* is permitted by *PO.LcpHashAlgMask*.

#### 3.4.4.2.2 Policy Lists

1. When verifying RSA signatures Policy Engine will decrypt signature to obtain encoded message. It will then use DER encoding to identify hash algorithm (*HashAlgID*) used to hash plain text message.

2. Policy Engine will validate signatures of all signed lists – both V2 and V3 style in all of available *Policy Data Files* and will reset platform if verification fails for any reason.

   a. Possible causes of errors include: signature of hash algorithm not supported by ACM; not supported key size, not supported curve; signature verification failure.

   **Note 1**. Abort in case when signature algorithm is not supported by ACM is needed to thwart attack when adversary intentionally changes signature algorithm ID of the list to appear it to be not supported with intention to cause fallback to ANY policy.

**Note 2**. Despite that V3 style elements cannot be present in V2 style lists signature of V2 style lists still must be verified. This is needed to thwart attack when adversary changes revision of V3 list to make it appear as V2 list.

3. Policy Engine will check revocation counters associated with lists and will abort with error if any of the lists is revoked.
4. Policy Engine will check hash algorithms of all elements in all lists and abort with error if finds one not supported by ACM.
    a. "Not supported" in this context must be understood as not supported by ACM only if support provided by embedded SW or not supported by TPM if ACM chooses to use TPM assistance.
    b. This requirement is necessary because otherwise special form of ACM substitution attack is possible when ACM is substituted by analogous ACM but lacking support of some of the crypto-algorithms.

#### 3.4.4.3 Policy Enforcement

1. When evaluating list signatures if combination of signature algorithm, key size, hash algorithm, and curve is not permitted by *Eff.LcpSignAlgMask* the list will be skipped from enforcement and its entire content ignored.
2. For each found element engine will consult *Eff.LcpHashAlgMask*. It will process element only if its *HashAlgID* is permitted by mask and will skip element otherwise.
3. Engine must follow current implementation logic:
    a. It must record each new element type as "required" when it is first discovered if its hash algorithm is permitted by effective *Eff.LcpHashAlgMask*. Element types filtered out by *Eff.LcpHashAlgMask* will not trigger "required" assertion.
    b. If at the end of evaluation match for "required" element type is not found, error must be generated.

If any of the element types was not discovered at all – it must be assumed to be evaluated successfully.

## 3.5 Revocation

### 3.5.1 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 text on 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 7. The *SINITMinVersion* field of the PS Index defines the minimum version of an 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 OEM (platform supplier) to restrict execution of SINIT versions prior to one they specify.

An SINIT version satisfying a platform *SINITMinVersion* may be found to have a security flaw resulting in platform vulnerabilities *after* the OEM has provisioned the PS Index. 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 ACM functions. Such a revocation or BIOS ACM would increase the AUX Index SVN requirement to a new target value, precluding execution by SINIT modules with lower SVN values, effectively revoking them. Misuse or misapplication of such a revocation / BIOS ACM could impact platform functionality adversely. For example, the AUX Index SVN bound to could be elevated to a new value for which no ACMs having a sufficiently high SVN exist for the affected platform.

The *MaxSinitMinVersion* field in the PO Index allows the platform owner or user to control such revocation. Unless this field is set to 0xff, the value given is the maximum value a revocation ACM may set the AUX index required SVN to.

§ §

# 4 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 left 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 and so that software should be able to detect the error in the TXT.ERRORCODE register and take appropriate action.

The particular remediation action needed will depend 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 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 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 reference 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 of the 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 of the 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 at 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 *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 will remain as default 0xFF value

and this will 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 V2 and V3 structures can be found in Table 32 and Table 33 respectively.

The following must be noted for TPM 2.0 index definition:

1. *authValue* of index by convention must remain at default value “empty buffer”. This will enable unrestricted access by any agent from any locality.
2. Deletion of index can be performed under control of OEM policy only – analogously to other TXT indices.

**Table 3. SGX Index Content**

| | |
|---|---|
| Bytes 1 - 7. Reserved, must be zero. | Byte 0. SGX SINIT_SVN.<br>Saved by MLE of specially defined tool. Must be initialized to default value “1” |

Table 3. SGX Index Contentdiagrammatically illustrates the SGX index structure

Table 4 shows content of *IA32_SE_SVN_STATUS* MSR used by software components:

**Table 4. IA32_SE_SVN_STATUS MSR (0x500)**

| Bits | Name | Comment |
|---|---|---|
| 0 | Lock | 0 – BIOS_SE_SVN MSR can be floated down. Launching a<br>properly signed ACM will not lead to LT shutdown irrespective of its SE SVN<br>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.<br>On CPUs that enumerate CPUID.feature_flags.SMX as 0, these bits are reserved (0). |
| 63:24 | reserved | |

This is read only MSR.

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 SINIT module, decompressing it and placing in heap SINIT memory. It also has to program *BiosSinitSize* field in BIOS Data table – see Table 17.

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 of the necessary information to program

*BIOS_SE_SVN MSR. SGX* index may not exist and is ignored if exists and 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 located 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 has to be updated and system reset to let BIOS use 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 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 Startup ACM to be executed at platform reset. Intel ® CPUs can support only single such ACM and therefore combining of 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 alignment of 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 for Client CNL and Server ICX platforms"*.

1. BtG, Startup and BIOS ACMs are combined into single binary referred to as Startup ACM (S-ACM). The Startup function of a S-ACM performs all steps required by BtG and TXT
2. Client and Server TXT flavors are unified. Client reset attack mitigation changed to be carried by BIOS and not by SCLEAN function.
3. Information about OEM established policies consumed by Startup function is delivered using BtG Key and Boot Policy manifests (KM and BPM). Delivering of information via FIT table records is 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. In result PS index, associated Policy Data File and SBIOS element are eliminated.
6. Supported cryptography has been reduces 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 of the 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 in platform 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 is removed since now S-ACM revocation is handled using BtG style;
   *PolicyControl* flags are 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 section K.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 defined in 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 2

6. BIOS data table no longer contains *LcpPdBase* and *LcpPdSize* fields since these fields were associated with currently removed PS LCP Data file – see section C.2



7. *ProcessorSCRTMStatus* field of *SinitMleData* heap table will be set to “1” since with BtG/TXT PCR0 measurement is always rooted in CPU HW – see C.5
8. MLE will see new events logged into TXT event log. Full list of new events can be found in **Error! Reference source not found.**
9. Three new registers have been defined in TXT device space: new ACM_POLICY_REGISTER at offset 0x780 has been defined to carry combination of BtG and TXT polies. MLE may benefit from it if needed; ACM_STATUS_REGISTER at offset 0x328 carries pre-boot error code information instead or ERRORCODE register. 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 *“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.

§ §

# Appendix A Intel® TXT Execution Technology Authenticated Code 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 5. This definition represents Revision 0.0 and 3.0 of the AC module header version (defined in the *HeaderVersion* field). ACM with version 3.0 support converge of Boot Guard and TXT.

**Table 5. 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<br>0 – TXT ACM<br>1 – S-ACM |
| HeaderLen | 4 | 4 | Header length (in multiples of four bytes)<br><br>161 - for version 0.0<br>224 – for version 3.0 |
| HeaderVersion | 8 | 4 | Module format version<br>0.0 – for SINIT ACM before 2017<br>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 |

| Field | Offset<br>version 0.0 / 3.0 | Size (bytes)<br>version 0.0 / 3.0 | Description |
|---|---|---|---|
| Date | 20 | 4 | Creation date (BCD format: year.month.day) |
| 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 Anclaves) 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)<br><br>64 - for version 0.0<br>96 – for version 3.0 |
| ScratchSize | 124 | 4 | Scratch field size (in multiples of four bytes)<br><br>143 = (2 * KeySize + 15) – for version 0.0<br>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<br>Present – for version 0.0<br>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) |

| Field | Offset version 0.0 / 3.0 | Size (bytes) version 0.0 / 3.0 | Description |
|---|---|---|---|
| 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 6. 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. 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***

Contains standard public key exponent.

***RSASig***

The 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 is 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 7. Chipset AC Module Information Table**

| Field | Offset (Bytes) | Width (Bytes) | Description |
|---|---|---|---|
| UUID | 0 | 16 | UUID of the Chipset AC module information table defined as:<br>ULONG UUID0; // 0x7FC03AAA<br>ULONG UUID1; // 0x18DB46A7<br>ULONG UUID2; // 0x8F69AC2E<br>ULONG UUID3; // 0x5A7F418D<br>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)<br>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.<br>Version 6 is added to cover addition of ACM Revision field.<br>Version 7 is added to cover addition of new ACM capability in capabilities field: Boot Point Technology 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 8 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 2. |
| AcmVersion | 36 | 1 | Version of this AC Module. It is compared against the *SINITMinVersion* field in LCP 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 10 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 8. 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 9). |

**Table 9. TXT_ACM_CHIPSET_ID Format**

| Field | Offset | Width (Bytes) | Description |
|---|---|---|---|
| Flags | 0 | 4 | Set of flags to further describe functions of the chipset ID structure.<br>Bit Description:<br>[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.<br>[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 10. Processor ID List**

| Field | Offset | 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 11. TXT_ACM_PROCESSOR_ID Format**

| Field | Offset | 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 |
| 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 12. TPM Info List**

| Field | Offset | 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:<br>TPM_ALG_RSA == 0x0001<br>TPM_ALG_SHA1 == 0x0004<br>TPM_ALG_SHA256 == 0x000B<br>TPM_ALG_SM3_256 == 0x0012<br>Etc. |

**Table 13. TPM Capabilities Field**

| Bit Position | Description |
|---|---|
| 1:0 | TPM2 *PCR Extend Policy* Support<br>00: illegal<br>01: 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.<br>10: Maximum performance Policy. Measurements are done using embedded fixed set of algorithms and TPM PCR extend commands<br>11: Both policies types are supported |
| 5:2 | TPM family support<br>0000: illegal<br>0001: discrete TPM 1.2 supported<br>0010: discrete TPM 2.0 supported<br>1000: firmware TPM 2.0 supported<br>Above settings can be combined to indicate multiple support options |
| 6 | TPM NV index set supported:<br>0 : Initial TPM 2.0 TPM NV index set out of 0x180_xxxx and 0x140_xxxx ranges<br>1: TCG compliant TPM 2.0 TPM NV index set out of Intel Reserved range of indices 0x1C1_0100 – 0x1C1_00x13F<br>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 in regards to 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

aforementioned commands 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 a number of 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 19).

**Notes on TPM family:**

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

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

0011 – Both combinations above are supported

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

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

### A.1.1 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 error code *#BadACMMType* during the loading of the authenticated code module if non-WB memory is detected.

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

### A.1.2 Authentication and Execution of AC Module

Authentication is performed after loading of 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 field contains a public key plus a 32-bit exponent 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 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 SINIT AC Module) 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**<br>**Priv Attribs** | RO<br>RO |

| Bits | Field Name | Field Description |
|---|---|---|
| 0 | SENTER.DONE.STS | The chipset sets this bit when it sees all of the threads have done an TXT.CYC.SENTER-ACK.<br>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 of the bits in the TXT.THREADS.JOIN register are clear. Thus, this bit will be set immediately after reset (since the bits are all 0).<br>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 |
| 6 | MEM-CONFIG-LOCK.STS | This bit will be set to 1 when the memory configuration has been locked. |

| Bits | Field Name | Field Description |
|---|---|---|
| | | 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.<br>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**<br>**Priv Attribs** | RO<br>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.<br>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**<br>**Priv Attribs** | RO<br>RW |

| Bits | Field Name | Field Description |
|---|---|---|
| 3:0 | Type2 /Module Type | 0 = BIOS ACM<br>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<br>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 14)<br>1 = Error condition reported by software |
| 31 | Valid/Invalid | 0 = Register content invalid, the rest of the register contents should be ignored.<br>1 = Valid error |

**NOTES:**

1. Upon successful execution, SINIT will put 0xC0000001 in the register.
2. 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 Table 31.

### Table 14. 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 |
| 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 |

| Type | Error condition | Mnemonic |
|---|---|---|
| 15 | Invalid voltage/bus ratio | #InvalidVIDBRatio |
| 16 – 65535 | Reserved | Reserved |

**NOTES:**

1. 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.***
2. #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**<br>**Priv Attribs** | -<br>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**<br>**Priv Attribs** | <br>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.VER.FSBIF – Frontside Bus Interface

| | |
|---|---|
| **Description** | This register identifies whether the chipset is debug or release fused.<br>On certain chipsets, a 4-byte read to this address will return either 0xFFFF_FFFF or 0x0000_0000. In these cases, the MLE should read an alternate offset (TXT.VER.EMIF, 200H) to capture this information. |
| **Offset** | 100H |
| **Pub Attribs**<br>**Priv Attribs** | RO<br>RO |

| Bits | Field Name | Field Description |
|---|---|---|
| 30:0 | Reserved | Reserved |
| 31 | DEBUG.FUSE | 0 = Chipset is debug fused<br>1 = Chipset is production fused |

### 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**<br>**Priv Attribs** | RO<br>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.QPIIF – Intel® QuickPath Interconnect Interface

| | |
|---|---|
| **Description** | This register identifies whether the memory controller or chipset is debug or release fused.<br>On certain chipsets, a 4-byte read to TXT.VER.FSBIF will return 0xFFFF_FFFF or 0x0000_0000. In these cases, the MLE should read this register to determine if the chipset is debug or release fused. |
| **Offset** | 200H |
| **Pub Attribs**<br>**Priv Attribs** | RO<br>RO |

| Bits | Field Name | Field Description |
|---|---|---|
| 30:0 | Reserved | Reserved |
| 31 | DEBUG.FUSE | 0 = Chipset is debug fused<br>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**<br>**Priv Attribs** | -<br>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.1.10 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**<br>**Priv Attribs** | RW<br>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**<br>**Priv Attribs** | RW<br>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**<br>**Priv Attribs** | RW<br>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**<br>**Priv Attribs** | RW<br>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**<br>**Priv Attribs** | RW<br>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**<br>**Priv Attribs** | RW<br>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.<br>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. |

## B.1.16 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.<br>This locality is not automatically opened after GETSEC[SENTER] and must be opened explicitly.<br>This locality will not be closed by the TXT.CMD.CLOSE-PRIVATE command. |
| **Offset** | 380H |
| **Pub Attribs**<br>**Priv Attribs** | -<br>WO |

| Bits | Field Name | Field Description |
|---|---|---|
| 7:0 | | |

## B.1.17 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.<br>This locality will not be closed by the TXT.CMD.CLOSE-PRIVATE command. |
| **Offset** | 388H |
| **Pub Attribs**<br>**Priv Attribs** | -<br>WO |

| Bits | Field Name | Field Description |
|---|---|---|
| 7:0 | | |

## B.1.18 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.<br>This locality is automatically opened after GETSEC[SENTER].<br>This locality will be closed by the TXT.CMD.CLOSE-PRIVATE command. |
| **Offset** | 390H |
| **Pub Attribs**<br>**Priv Attribs** | -<br>WO |

| Bits | Field Name | Field Description |
|---|---|---|
| 7:0 | | |

## B.1.19 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.<br>This locality will be closed by the TXT.CMD.CLOSE-PRIVATE command or by the GETSEC[SEXIT] instruction. |
| **Offset** | 398H |
| **Pub Attribs**<br>**Priv Attribs** | -<br>WO |

| Bits | Field Name | Field Description |
|---|---|---|
| 7:0 | | |

## B.1.20 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**<br>**Priv Attribs** | RO<br>RO |

| Bits | Field Name | Field Description |
|---|---|---|
| 255:0 | | |

## B.1.21 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.<br>The MLE issues the TXT.CMD.SECRETS command prior to placing secrets in memory for the first time. |
| **Offset** | 8E0H |
| **Pub Attribs**<br>**Priv Attribs** | -<br>WO |

| Bits | Field Name | Field Description |
|---|---|---|
| 7:0 | | |

## B.1.22 TXT.CMD.NO-SECRETS – Clear Secrets Command

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

| Bits | Field Name | Field Description |
|---|---|---|
| 7:0 | | |

## B.1.23 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**<br>**Priv Attribs** | RO<br>RW |

| Bits | Field Name | Field Description |
|---|---|---|
| 0 | Reserved | Reserved |
| 1 | SECRETS.STS | 0 = Chipset acknowledges that no secrets are in memory<br>1 = Chipset believes that secrets are in memory and will provide reset protection |

| | | |
|---|---|---|
| 63:2 | Reserved | Reserved |

## B.2 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 1.8, 1.9, and 1.10 above.

## B.3 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 15. 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 LocalityX.OPEN 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 insure 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 16. 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. Note 1. |
| 8 | BiosDataSize - 8 | BiosData | BIOS specific data. The format of this data is described below in Table 17. |
| 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. Note 2. |
| 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. Note 1. |
| 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 19. |

| Offset | Length (bytes) | Name | Description |
|---|---|---|---|
| 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. [Note]. |
| BiosDataSize + OsMleDataSize + OsSinitDataSize + 8 | SinitMleDataSize – 8 | SinitMleData | SINIT data passed to the MLE. The format of this data is described below in Table 20. |

***Note:***

1. For proper data alignment on 64bit processor architectures this field must be a multiple of 8 bytes.

2. *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.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 17.

**Table 17. 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 < 2 are not 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 && < 5** | | | |
| 28 | 4 | SinitFlags | BIOS-provided information for SINIT AC module consumption. Bit definition will be dependent on the chipset. |
| **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 18. |
| **Versions >= 4** | | | |
| 36 | BiosDataSize - 36 | ExtDataElements[] | Array/list of extended data element structures. See below for element definitions. |

### Table 18. 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<br>01: client platform, client SINIT is required<br>10: server platform, server SINIT is required<br>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];  // phys addr of ACM
} HEAP_ACM_ELEMENT;
```

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

BIOSes 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 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 STM specification.

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

**Table 19. OS to SINIT Data Table**

| Offset | Length (bytes) | Name | Description |
|---|---|---|---|
| 0 | 4 | Version | Version number of the *OsSinitData* table. Current values are 4 through 6 (versions < 4 are not supported) for TPM 1.2 family. TPM 2.0 family version must be 7. 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). |
| **Version <= 6** | | | |
| 4 | 4 | Reserved | Reserved for future use |
| **Version = 7** | | | |
| 4 | 4 | Flags | Bit 0: PCR Extend Policy Control<br>0 – Maximum Agility Policy<br>1 – Maximum Performance Policy |
| 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) |
| 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. |

| Offset | Length (bytes) | Name | Description |
|---|---|---|---|
| 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. Note that for TPM 2.0 family, bits 5:4 must be zero, since D/A mapping is required. |
| **Version = 5** | | | |
| 84 | 8 | EFI RSDT Pointer | Physical address of RSDT table when an EFI boot was performed. This will be ignored if SINIT finds the standard ACPI RSDT table. |
| **Versions >= 6** | | | |
| 84 | 8 | EFI RSDP Pointer | Physical address of RSDP when an EFI boot was performed. This will be ignored if SINIT finds the standard ACPI RSDP pointer. |
| 92 | OsSinitDataSize - 92 | ExtDataElements[] | Array/list of extended data element structures. See below for element definitions. Note that for TPM2, there must be at least one HEAP_EVENT_LOG_POINTER_ELEMENT2 structure with a HEAP_END_ELEMENT terminating the list. For TPM1.2, the list can be empty. |

## C.4.1 HEAP_TPM_EVENT_LOG_ELEMENT

This element specifies event log created by SINIT in TPM 1.2 mode

```
#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 G 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.

### 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;
```

*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 20

See Appendix G for further explanation.

## C.5 SINIT to MLE Data Format

Table 20 below defines the format of the SINIT data presented to the MLE.

**Table 20. SINIT to MLE Data Table**

| Offset | Length (bytes) | Name | Description |
|---|---|---|---|
| 0 | 4 | Version | Version number of the *SinitMleData* table. Current values are 6 through 9 (versions < 6 are not supported). Supported value for TPM 2.0 is 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 |

| Offset | Length (bytes) | Name | Description |
|---|---|---|---|
| **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 |
| 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 | Reserved |
| 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 |
| 128 | 4 | NumberOfSinitMdrs | Number of SINIT Memory Descriptor Records |
| 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**Note:** Maximum version generated by SINIT in TPM 1.2 mode must be 8. In TPM 2.0 mode SINIT must generate version 9.<br><br>Table 21Table ). |

| Offset | Length (bytes) | Name | Description |
|---|---|---|---|
| 136 | 4 | SinitVtdDmarTableSize | Length of the Intel® Virtualization Technology (Intel® VT) for Directed I/O (Intel® VT-d) DMAR table pointed to by the *SinitVtdDmarTable* 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. |
| **Versions >= 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.<br>Bit 0 = 0 if PCR0 measurement for this boot was rooted in BIOS.<br>It will be set to '1" since with CBnT PCR0 measurement is always rooted in CPU HW<br>Bits 31:1 – Reserved for future use |
| **Versions >= 9** | | | |
| 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.

### Table 21Table C-6. 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. |
| 16 | 1 | Type | Memory range type. Valid values:<br>0 Usable, good memory<br>1 SMRAM– Overlayed – deprecated<br>2 SMRAM– Non-Overlayed – deprecated<br>3 PCIe - PCIe Extended Configuration Region<br>4 Persistent memory<br>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 will be not scrubbed by BIOS ACM.

## 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.

§ §

# Appendix D LCP v2 Data Structures

## D.1 LCP_POLICY

Platform Owner policy structure is of the type LCP_POLICY. Respective object is stored in the TPM NV. The required fields for LCP_POLICY are as follows:

```
#define LCP_POLHALG_SHA1            0

#define LCP_POLTYPE_LIST            0
#define LCP_POLTYPE_ANY             1

typedef struct {
    UINT8            Sha1[20];
} LCP_HASH;

#define LCP_MAX_LISTS         8

typedef struct {
    UINT16      Version;          // 2.4
    UINT8       HashAlg;          // one of LCP_POLHALG_*
    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;

typedef struct {
    UINT32     reserved:1;
    UINT32     NPW_OK:1;          // NPW OK
    UINT32     Reserved2:1;
    UINT32     Pconf_Enforced:1; // Functionality see K.2.3
    UINT32     reserved3:28;
} PolicyControl
```

## D.2 LCP_POLICY_DATA

For each policy of type 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 **Error! Reference source not found.**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
```

*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

```
#define LCP_POLSALG_NONE            0
#define LCP_POLSALG_RSA_PKCS_15     1

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.

Supported public key sizes are 2048 and 3072 bits. It is required that a public key size of at least 2048 bits be used. Larger sizes may take longer to verify.

The exponent is fixed and must be 65537.

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.

***Note:*** Legacy policy engine recognizes V3 style list signatures presented in section E.3.1

## D.3.2 LCP_POLICY_LIST Structure

```
typedef struct {
    UINT16                  Version;
    UINT8                   Reserved;
    UINT8                   SigAlgorithm;     // one of SIGALG_*
    UINT32                  PolicyElementsSize;
    LCP_POLICY_ELEMENT      PolicyElements[];
    optionally LCP_SIGNATURE Signature;
} LCP_POLICY_LIST;
```

The *Reserved* field must be set to 0.

*PolicyElementsSize* specifies the size (in bytes) of all of the LCP_POLICY_ELEMENTS structures in the object. It may be 0. *A* LCP_POLICY_LIST 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 SIGALG_RSA_PKCS_15 then the *Signature* field must be present (else it must not). For a signed list, the RSA signature will be calculated over the *entire* LCP_POLICY_LIST structure, including the *Signature* member, except for the *SigBlock* field.

The version of the LCP_POLICY_LIST structure defined here is 1.0 (100H).

## D.4 LCP_POLICY_ELEMENT

```
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.

### D.4.1 LCP_MLE_ELEMENT

```
#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. An MLE will match the policy if its hash (as calculated when traversing its

pages in the MLE page table; not the value of PCR 18 after it has been extended) 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 also of the values in *Hashes[]*.

If *NumHashes* is 0 then this element will evaluate to false for all MLEs.

## D.4.2 LCP_PCONF_ELEMENT

```
#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 the time of 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 and 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 and every 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.

The 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.4.3 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 for 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.5 Structure Endianness

Endianness deals with the sequencing order of stored bytes. There are two common sequencing orders: Little Endian (format used by Intel) and Big Endian. All structures and data are in Little Endian format, even LCP_POLICY.

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.

§ §

# Appendix E LCP Data Structures, v3

## E.1 NV Index LCP Policy

*HashAlg* is now 16 bits, and uses the encoding defined by the "*TPM2.0 Library specification"* as repeated here for reference:

```
#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
```

Additionally, *LcpSignAlgMask* uses the following encodings (also defined by the *TPM2.0 specification*) for signing algorithms:

```
#define  TPM_ALG_RSA        0x0001
#define  TPM_ALG_SM2        0x001B
```

As described in 3.3.2, new 16-bit field *LcpHashAlgMask* and new 32-bit field *LcpSignAlgMask* are added to the LCP_POLICY structure for Platform Owner policy.

*LcpHashAlgMask* identifies the Platform Owner's selection of *HashAlgIDs* permitted during LCP policy evaluation.

*LcpHashAlgMask* bit field will use the following *HashAlgID* mask definition:

```
typedef struct {
    UINT16 TPM_ALG_SHA1:1;        // BIT0
    UINT16 Reserved:2;            // BITS 1-2
    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_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 0-1
    UINT32 TPM_ALG_RSASSA/2048/SHA1:1   // BIT2: Legacy
    UINT32 TPM_ALG_RSASSA/2048/SHA256:1// BIT3: Suite C
    UINT32 Reserved:2                   // BITS 4-5
    UINT32 TPM_ALG_RSASSA/3072/SHA256:1// BIT6:
    UINT32 TPM_ALG_RSASSA/3072/SHA384:1// BIT7: NCSS 2-15
    UINT32 Reserved:8                   // BITS 8-15
    UINT32 TPM_ALG_SM2/SM2_CURVE:1      // BIT16: Chinese alg
    UINT32 Reserved2:15
} LCP_APPROVED_SIGNATURE_ALG;
```

With the addition of the two above fields, the new TPM NV LCP Policy structure has the following format:

```
typedef struct {
    UINT16    Version;        // 0x0300 == 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_ALG
    UINT32    LcpSignAlgMask  // LCP_APPROVED_SIGNATURE_ALG
    UINT32    Reserved2;
    LCP_HASH2 PolicyHash;
 } LCP_POLICY2;
```

***Version*** updated to value 3.2 to indicate backwards compatible change

The salient changes to this structure, and the *PolicyControl* DWORD defined below from V3 are described in 3.3.2.

# E.2 LCP Policy Data

An LCP *Policy Data File*'s structure has only one change for V3 from V2 [D.2]: mixing of LCP_POLICY_LIST and LCP_POLICY_LIST2 lists is allowed.

## E.2.1 LCP_LIST

```
typedef union {
    LCP_POLICY_LIST  TPM12PolicyList; // See Appendix D
    LCP_POLICY_LIST2 TPM20PolicyList;
} LCP_LIST;
```

## E.2.2 LCP_POLICY_DATA2

```
typedef struct {
    char       FileSignature[32];
    UINT8      Reserved[3];
    UINT8      NumLists;
    LCP_LIST   PolicyLists [NumLists];
} LCP_POLICY_DATA2;
```

# E.3 LCP_POLICY_LIST2

The V2 LCP Policy list structure has been modified to V3 to accommodate algorithm agility.

The V3 LCP list structure has the following format:

```
typedef struct {
    UINT16        Version;
    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;
```

The changes from LCP_POLICY_LIST to LCP_POLICY_LIST2 structures are:

- *Version* is promoted to 2.1
- *SigAlgorithm* field uses TPM2.0 compatible numeric values
- *SigAlgorithm* field is expanded to UINT16 and reserved field is removed
- Signature field has different format specified in E.3.1.
- Mixture of legacy and new *LCP_POLICY_ELEMENT* structures is allowed.

**Notes:**

- It is presumed and required that legacy LCP_POLICY_LIST structures will contain only legacy LCP_POLICY_ELEMENT definitions
- By allowing a mix of legacy and new LCP_POLICY_ELEMENT structures in the same list, each Authority is able to create one single list covering both TPM1.2 and TPM2.0 LCP data
- Since mixed lists of elements are permitted in the version V3, use of legacy LCP_POLICY_LIST structures is not required and is discouraged.

## E.3.1 List Signatures

V3 LCP lists may use the following signature algorithms:

- RSASSA (#define TPM_ALG_RSASSA 0x0014)
- SM2 (#define TPM_ALG_SM2 0x001B)

For V2 LCP lists, the NULL algorithm is typed as 0x0010.

### E.3.1.1 RSASSA

Support will be limited to version RSASSA-PKCS1-v1_5 as defined in "*NIST, FIPS PUB 186-3, Federal Information Processing Standards Publication, Digital Signature Standard (DSS), June 2009"* with reference to "*RSA Labs, PKCS #1 v2.1: RSA Cryptography Standard, June 14, 2002"* and "*RSA Labs, PKCS #1 v2.1 Errata, Revision: 2.0, December, 2005*"

Signatures will be supported only with the following approved hash functions: TPM_ALG_SHA1; TPM_ALG_SHA256; TPM_ALG_SHA384; SM3.

Supported key sizes will be 2048 and 3072 bits.

### E.3.1.2 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*"

## E.3.2 Signature Format

The LCP_SIGNATURE structure layout shown in section D.3.1 above is unchanged, but was aliased to LCP_RSA_SIGNATURE for clarity in its use in the new structure:

```
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;

typedef union {
    LCP_RSA_SIGNATURE RsaSignature;
    LCP_ECC_SIGNATURE EccSignature;
} LCP_SIGNATURE2;
```

# E.4 New Policy Elements

The LCP structures used for TPM 1.2 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.

The overall policy element structure remains unchanged – see D.4

## E.4.1 LCP_Hash

```
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;
```

This structure was elaborated to support additional algorithms beyond SHA1.

## E.4.2 MLE Element

```
#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;
```

## E.4.3 STM Element

```
#define LCP_POLELT_TYPE_STM2 0x14

typedef struct {
    UINT16     HashAlg;    // one of TPM_ALG_*
    UINT16     NumHashes;
    LCP_HASH2  Hashes[NumHashes];
} LCP_STM_ELEMENT2;
```

## E.4.4 PCONF Element

The PCONF Element structure is now designed so it can be created using the TPM2_Quote command and evaluated using the TPM2_PolicyPCR command.

The TPM2_Quote command returns the following structure:

```
typedef struct {
    UINT16     size
    TPMS_ATTEST   attestationData;
} TPM2B_ATTEST;
```

Where TPMS_ATTEST has the following form:

```
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 shall be inserted in big endian format. Rest of the LCP_PCONF_ELEMENT2 structure is in little endian format.

TPML_PCR_SELECTION structure which is sub-component of new LCP_PCONF_ELEMENT2 definition contains “count” field allowing making multiple PCR selections using different *HashAlgIDs*.

“*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.

## E.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 22. AUX Data Structure**

| | | | | |
|---|---|---|---|---|
| Revocation area | BIOS AC registration data | AUX Data Version | Other data | Internal TXT digest area |

Table above diagrammatically illustrates the content of 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 containing fixed size data values. d for TPM 2.0. It uses TPM 2.0-compatible numeric values and indicates the *HashAlgID* of the digests stored in AUX index.

**Internal TXT digest area** - BIOSAC / SINIT intercommunication area containing variable size data values in TPMT_HA format.

## E.6 Structure Endianness

Endianness deals with the sequencing order of stored bytes. There are two common sequencing orders: Little Endian (format used by Intel) and Big Endian. All structures and data are in Little Endian format, even LCP_POLICY, except situation below:

TPMS_QUOTE_INFO structure in PCONF TPM 2.0 definition is a standard TPM 2.0 structure; it should be in Big Endian format, while the rest of PCONF structure is in Little Endian format.

§ §

# Appendix F Platform State upon SINIT Exit and Return to MLE

The following table describes the processor state of the ILP after returning to the MLE from GETSEC[SENTER] and the RLPs after waking from SENTER. This will be the state seen by the MLE.

**Table 23. 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, SSE State←; others unchanged | Unchanged |
| CR4 | 0x00004000 | 0x00004000 |
| EFLAGS | 0x000000XX (XX = Undefined) | 0x000000XX (XX = Undefined) |
| EIP | [MLEHeader.EntryPoint] | [TXT.MLE.JOIN + 12] |
| ESP | Undefined | Undefined |
| EBP | Undefined | Undefined |
| ECX | Ptr to MLE page table Note 1 | 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] | 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 2, 3 | IA32_MISC_ENABLE & 0xFEE324A8 Note 3 |

| Resource | ILP on MLE re-entry point | RLP on MLE re-entry point |
|---|---|---|
| Performance counters and counter control registers | 0 | 0 |
| IA32_APIC_BASE MSR | 35:12 cleared to 0xFEE00 | 35:12 cleared to 0xFEE00, bit 8 (BSP) cleared to 0 |
| **LTCS** | | |
| Private Space | Open | |
| TXT.ERRORCODE | 0xC0000001 | |
| **TPM** | | |
| Locality | No locality active. Locality 2 open, | |
| **PCI** | | |
| PCI Index/Data ports 0xCF8-0xCFF | Undefined | |

***Note:***

1. 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 contents of ECX are undefined
2. Bit 3 (thermal monitor enable) will be set to 1 if it was previously clear.
3. Bit 18 (MONITOR/MWAIT enable) will be set to 1 if it was previously clear, when bit 1 of *OsSinitData.Capabilities* (use of MONITOR for RLP wakeup) is set.
4. GDTR on entry to MLE retains values established by SINIT module and therefore incorrect and unusable for MLE. MLE developers should establish own GDT immediately

The TPM will not have any locality active following SENTER.

§ §

# Appendix G 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.

## G.1 TPM 1.2 Event Log

The log is allocated by system software and the physical address of the container (see Table 24) provided to SINIT in a HEAP_TPM_EVENT_LOG_ELEMENT in the *OsSinitData.ExtDataElements[]* field. An SINIT ACM that supports details/authorities PCR mappings (see - 1.10.2 and Table 2) will support this element type. ACMs that do not support this element type will ignore it. If system software does not provide this element to an SINIT ACM that supports it, SINIT will simply not populate a log.

**Table 24. 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. |
| 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. |

| Field | Offset | Size (bytes) | Description |
|---|---|---|---|
| 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 PCREvent structures (see below) |

## G.1.1 PCR Events

```
typedef struct {
    UINT32    PCRIndex;
    UINT32    Type;
    UINT8     Digest[20];
    UINT32    Size;
    UINT8     Data[Size];
} TPM12_PCREvent;
```

*PCREvents* describe the data whose SHA1 hash (*Digest[]*) was extended into the specified *PCR*. Depending on the event *Type*, the *Data[]* may be the entire hashed object or just a description (e.g. version). PCREvents 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 26.

*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*.

**Table 25. Table PCR Event Log Structure**

| Field | Offset | Size (bytes) | Description |
|---|---|---|---|
| PCRIndex | 0 | 4 | The Index of PCR to which this event was extended. |
| Type | 4 | 4 | Event type |
| Digest | 8 | 20 | SHA1 hash. |
| Size | 28 | 4 | The size of the event data. |
| Data | 32 | *Size* | The data of the event. |

The following event types are defined for SINIT event logging. Events supported in legacy PCR mapping only (LG) are removed from the table.

**Table 26. Event Types**

| Event Type | Value | Digest and Data content | PCR |
|---|---|---|---|
| EVTYPE_BASE (EB) | 0x400 | Base of TXT event types | N/A |
| EVTYPE_PCRMAPPING | EB + 1 | *Digest* = zero digest – 20 bytes of zeros.<br>*Data* = DWORD with bits 31:1 all zeros (reserved for now). Bit 0 = 0 if using legacy PCR Mapping and 1 if using D/A mapping. | Not extended – informative only. PCRIndex field is set to 0xFF. Since LG mapping is not supported this event is reserved. |
| EVTYPE_HASH_START | EB + 2 | *Digest* = SHA1(*Data*)<br>*Data* = ACM ID + EDX – 36 bytes total. | 17 / 17 |
| EVTYPE_MLE_HASH | EB + 4 | *Digest* = *MleHash*<br>*Data* = none | 18 / 17 |
| EVTYPE_BIOSAC_REG_DATA | EB + 10 | *Digest* = BIOS AC registration data – 20 bytes from AUX index<br>*Data* = none | 17 |
| EVTYPE_CPU_SCRTM_STAT | EB + 11 | *Digest* = SHA1(*Data*)<br>*Data* = CPU S-CRTM status DWORD | 17 / 18 |
| EVTYPE_LCP_CONTROL_HASH | EB + 12 | *Digest* = SHA1(*Data*)<br>*Data* = LCP *PolControl* DWORD | 17 / 18 |
| EVTYPE_ELEMENTS_HASH | EB + 13 | *Digest* = SHA1(*Data*)<br>*Data* = array of matching LCP elements data as filled by LCP. Total 4 * (20 + 4) = 96 bytes | 17 |
| EVTYPE_STM_HASH | EB + 14 | *Digest* = *StmHash*<br>*Data* = none<br>Note: This event is created only if IA32_SMM_MONITOR_CTL[0] == 1 | 17 |
| EVTYPE_OSSINITDATA_CAP_HASH | EB + 15 | *Digest* = SHA1(*Data*)<br>*Data* = *OsSinitData.Capabilities* DWORD | 17 / 18 |
| EVTYPE_SINIT_PUBKEY_HASH | EB + 16 | *Digest* = SHA1 (*ACHeader.RSAPubKey*)<br>Data =none | 18 |

| Event Type | Value | Digest and Data content | PCR |
|---|---|---|---|
| EVTYPE_LCP_HASH | EB + 17 | Digest = SHA1(*Data*)<br>*Data* = array of list hashes containing matching LCP elements data as filled by LCP.<br>Total = [1 – 4] * 20 = 20 – 80 bytes | 18 |

# G.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.

## G.2.1 TPM 2.0 Event Log Pointer Element

Like in TPM 1.2 mode under TPM 2.0 the log is allocated by system software and the physical address of the container is provided to SINIT in a HEAP_EVENT_LOG_POINTER_ELEMENT2_1 in a *OsSinitData.ExtDataElements[]* field.

There are no requirements for event log to be in DMA protected memory – SINIT will not enforce this requirement.

## G.2.2 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 of TCG specification will prevail.

```
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 hashing 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 capabilities field of ACM Info Table *ACMInfoTable.Capabilities*[9] added to declare format of supported event log – see Table 2. MLE/SINIT Capabilities Field Bit Definitions with this revision of specification is always forced to "1".

## G.2.3 Event Types Changed and Added for TPM2.0

In addition to the event types given in Table 26, the following types may occur in logs for TPM2 family.

### Table 27. Event Types Changed and Specific to TPM2.0

| Event Type | Value | Digest and Event | PCR | Comments |
|---|---|---|---|---|
| EVTYPE_BASE (EB) | 0x400 | Base of TXT event types | N/A | Not new. Copied here for clarity. |
| EVTYPE_PCR_MAPPING | EB +1 | *Digest* = $ZeroDigest_{HashAlgIDSize}$ of size correponding to hash algorithm.<br>*EventData* = DWORD with bits 31:1 all zeros (reserved for now). Bit 0 = 0 if using legacy PCR Mapping and 1 if using D/A mapping.<br>EventDataSize =4 | 0xFF | Not extended – informative only.<br>Since LG mapping is not supported this event is reserved. |
| EVTYPE_HASH_START | EB +2 | *Digest*=$Hash_{HashAlgID}$ (EventData)<br>*EventData*=SinitHash + EDX.<br>EventDataSize= 36 | 17 | *EventData* is SinitHash and EDX value stored during SINIT authentication process in ACM header scratch area. |
| EVTYPE_COMBINED_HASH | EB +3 | | | Reserved in TPM2.0 mode |
| EVTYPE_MLE_HASH | EB +4 | *Digest*= $Hash_{HashAlgID}$ (Mle)<br>*EventData*=Empty buffer<br>EventDataSize=0 | 17 | |
| EVTYPE_BIOSAC_REG_DATA | EB +10 | *Digest*= $Hash_{HashAlgID}$ (*EventData*)<br>*EventData*=BIOS AC Registration Data<br>*EventDataSize*=32 | 17 | *EventData* is 32 bytes of AUX index starting at offset 0 |
| EVTYPE_CPU_SCRTM_STAT | EB +11 | *Digest*= $Hash_{HashAlgID}$ (*EventData*)<br>*EventData*=CPU S-CRTM Status DWORD<br>EventDataSize=4 | 17 & 18 | |
| EVTYPE_LCP_CONTROL_HASH | EB +12 | *Digest*= $Hash_{HashAlgID}$ (EventData)<br>*EventData*=LCP PolControl DWORD<br>EventDataSize=4 | 17 & 18 | *EventData* is effective PolControl derived from PS and PO policies. |
| EVTYPE_ELEMENTS_HASH | EB +13 | | | Reserved in TPM2.0 mode |

| Event Type | Value | Digest and Event | PCR | Comments |
|---|---|---|---|---|
| EVTYPE_STM_HASH | EB +14 | If STM is present<br>*Digest*= $Hash_{HashAlgID}$ (Stm)<br>If STM is not present<br>*Digest*= $Hash_{HashAlgID}$ (0x00)<br>*EventData*=Empty buffer<br>EventDataSize=0 | 17 | 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)<br>*EventData*=OsSinitData Capabilities DWORD<br>EventDataSize=4 | 17 & 18 | *EventData* is capabilities DWORD passed to SINIT by pre-MLE SW |
| EVTYPE_SINIT_PUBKEY_HASH | EB +16 | Digest= $Hash_{HashAlgID}$ (CS PUBKEY)<br>EventData=Empty buffer<br>EventDataSize=0 | 18 | CS PUBKEY is chipset public key. Hash can be retrieved from LT 0x400 |
| EVTYPE_LCP_HASH | EB +17 | | | Reserved in TPM2.0 mode |
| EVTYPE_LCP_DETAILS_HASH | EB + 18 | *Digest*= $Hash_{HashAlgID}$ (*EventData*)<br>*EventData*=concatenation of digests and policy controls of matching policy elements – see 3.3.8.1<br>*EventDataSize*=sizeof(EventData)<br>If polcy evaluates to ANY Then<br>*Digest*= $Hash_{HashAlgID}$ (0x00)<br>*EventData*=0x00<br>*EventDataSize*=1 | 17 | *EventData* is concatenation of digests, policy controls of all matching elements contributed to effective policy – see 3.3.8.1.<br>Analogous to event type 13 but uses different event format. |

| Event Type | Value | Digest and Event | PCR | Comments |
|---|---|---|---|---|
| EVTYPE_LCP_AUTHORITIES_HASH | EB + 19 | *Digest*= $Hash_{HashAlgID}$ (*EventData*)<br>*EventData*=concatenation of list descriptors containing matching policy elements - see 3.3.8.2<br>EventDataSize=sizeof(EventData)<br>If policy evaluates to ANY Then<br>*Digest*= $Hash_{HashAlgID}$ (0x00)<br>*EventData*=0x00<br>*EventDataSize*=1 | 18 | *EventData* is concatenation of list description structures of all lists containing matching elements contributed to effective policy – see 3.3.8.2<br>Analogous to event type 17 but uses different event format and hashing rules. |
| EVTYPE_NV_INFO_HASH | EB + 20 | *Digest*= $Hash_{HashAlgID}$ (*EventData*)<br>*EventData*=array of TPMS_NV_PUBLIC structures of defined indices - see 3.3.9.<br>EventDataSize=sizeof(EventData) | 17/ 18 | *EventData* is concatenation of TPMS_NV_PUBLIC structures of all defined TPM NV indices. |
| EVTYPE_COLD_BOOT_BIOS_HASH | EB + 21 | *Digest*= $Hash_{HashAlgID}$ (*"TXT cold boot"* BIOS set)<br>*EventData*="TXT cold boot BIOS"<br>EventDataSize=sizeof(*EventData*) | 17 | *Digest* is *"TXT cold boot"* BIOS set digest: SHA1 for TPM 1.2 or TPML_DIGEST_VALUES structure for TPM 2.0. Constituent input digests are taken either from BPM or AUX index<br>*EventData* is string "TXT cold boot BIOS" |
| EVTYPE_KM_HASH | EB + 22 | *Digest*= $Hash_{HashAlgID}$ (Key Manifest)<br>*EventData="*Key Manifest signature".<br>EventDataSize=sizeof(*EventData*) | 17 | *Digest* is analogous to the above entry but hashed object is signature of key manifest.<br>*EventData* is string "Key Manifest signature" |

| Event Type | Value | Digest and Event | PCR | Comments |
|---|---|---|---|---|
| EVTYPE_BPM_HASH | EB + 23 | *Digest*= $Hash_{HashAlgID}$ (Boot Policy Manifest). *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 | *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 *EventData* key manifest descriptor |
| EVTYPE_BPM_INFO_HASH | EB + 25 | *Digest*= $Hash_{HashAlgID}$ (*EventData*) *EventData*=Boot Policy Manifest descriptor. EventDataSize=sizeof(*EventData*) | 18 | *Digest* is analogous to the above entry but hashed object is boot policy manifest descriptor NOTE. *EventData* boot policy manifest descriptor |
| EVTYPE_BOOT_POL_HASH | EB + 26 | *Digest*= $Hash_{HashAlgID}$ (*EventData*) *EventData*=ACM_POLICY_STATUS EventDataSize=sizeof(*EventData*) | 17 & 18 | *Digest* is analogous to the above entry but hashed object is ACM_POLICY_STATUS register NOTE *EventData* is DWORD register M.0x378 |
| EVTYPE_CAP_VALUE | EB +255 | *Digest*= $Hash_{HashAlgID}$ (0x01) *EventData*=0x01 EventDataSize=1 | 17/ 18 | Caps PCR value if digest cannot be computed due to limited embedded SW capabilities. |

***Note:*** Complete description is available in *"Converged Boot Guard and Intel® Trusted Execution Technologies. BIOS Specification for Client CNL and Server ICX platforms"*

§ §



# Appendix H ACM Hash Algorithm Support

## H.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 E.1

## H.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:

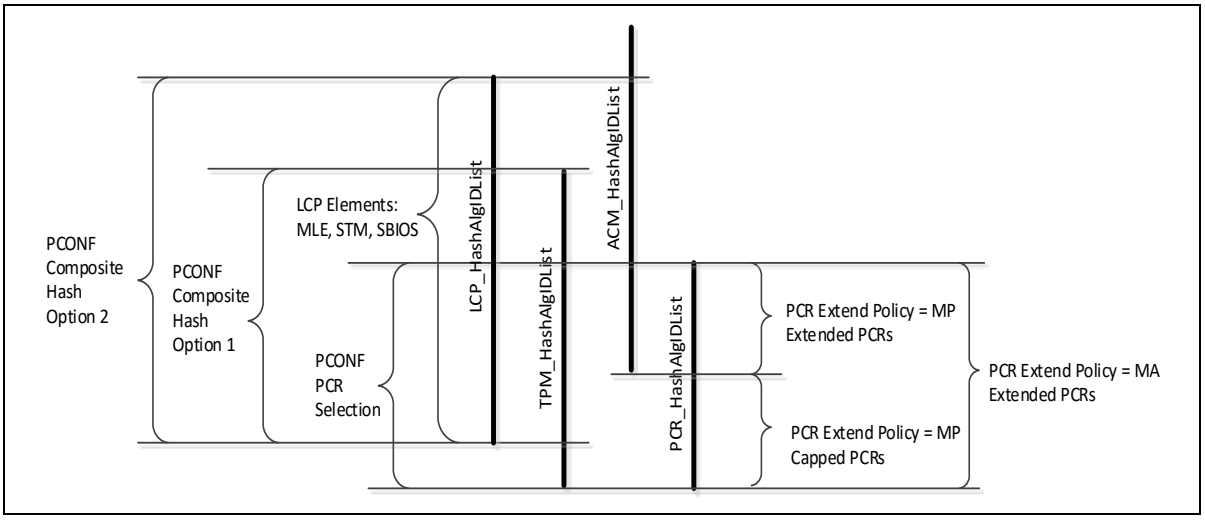
- *TPM_HashAlgIDList* - list of TPM supported *HashAlgIDs*. This is the list of all *HashAlgIDs* supported by given entity of the TPM device. Determined dynamically at run time;
- *PCR_HashAlgIDList* - list of *HashAlgIDs* for which given specimen of TPM device implements dynamic PCRs 17 & 18. Determined 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 11. TXT_ACM_PROCESSOR_ID Format
- *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.

## H.3 Hash Algorithm List Subsets

ACM will detect content of all four *HashAlgID* Lists:

1. *TPM_HashAlgIDList* will be detected by executing the TPM2_GetCapability() command, probing every *HashAlgID* in the TPM2.0 supported list
2. *PCR_HashAlgIDList* will be created as a subset of *TPM_HashAlgIDList* via the TPM2_PCR_Read() command. Both PCR17 and 18 will be read. *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 after effective LCP policy is determined based on standard LCP combining principles. This list will represent Platform Policy.

**Figure H-1. Hash Algorithm List Selection**




Based on the above lists, ACM will create a number of 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 that 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 i.e.
      **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 with value “1”.
- If PCR Extend Policy is selected to be MA, then all TPM PCRs will be extended and none will be capped.



§ §

# Appendix I 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. The final table describes the mapping of register sub-field values to meanings for values whose mappings are stable and potentially useful to the end user.

In order to make updates to this table-set available between releases of this document, the equivalent of the contents of this appendix can be found at http://software.intel.com/en-us/articles/intel-trusted-execution-technology.

**Table 28. 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 29. 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<br>Error #0x10 = ACM SVN |
| 15 | Reserved | 0 |
| 14:0 | Type2 | 0 = Legacy shutdown (non TXT-specific). |
| | | 1 – 4 = Reserved |
| | | 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 |

| Bit | Name | Value / Error |
|---|---|---|
| | | 0xA = Illegal event or *IllegalProcessorState* – collection of various illegal causes.<br>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).<br>2. A non-virtualized INIT event occurs while post-SENTER.<br>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.<br>4. An SENTER, SEXIT, or WAKEUP doorbell is received while post-VMXON.<br>5. A thread wakes from wait-for-SIPI while some other thread in the same package is in AC-mode.<br>#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 30. 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:<br>1. If Class Code = "TPM Access" and Major Error Code = "TPM retuned an error" – Field value = TPM returned error code<br>TPM returned error code is posted in different format for TPM 1.2 and TPM 2.0 modes.<br>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.<br>TPM 2.0: TPM returned error code is posted as is.<br>2. If Class Code = "Launch Control Policy and Major Error Code = "Policy Integrity Fail" – Field value = (LIST_INDEX << 6) + Specific Minor Error Code |

| Bit | Name | Description |
|---|---|---|
| | | 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<br>1 = SINIT |

**Table 31. TXT.ERRORCODE definitions stable among ACM modules**

| Major Error Code | Minor Error Code | Description |
|---|---|---|
| Class code = 1: ACM Entry – BIOS AC and SINIT | | |
| 1 | 0-x | Error in ACM launching:<br>SINIT is launched not via SENTER;<br>Reserved bits in EDX register are not 0;<br>Minor error code contains additional details. |
| 3 | 0 | Client SINIT detected LTSX fused processor or<br>Server SINIT detected non- LTSX fused processor |
| 9 | 0 | ACM is revoked |
| Class code = 2: MTRR Check – BIOS AC and SINIT | | |
| 1 | 0 | MTRR Rule 1 Error |
| 2 | 0 | MTRR Rule 2 Error |
| 3 | 0 | MTRR Rule 3 Error |
| 4 | 0 | MTRR Rule 4 Error |
| 5 | 0 | MTRR Rule 5 Error |
| 6 | 0 | MTRR Rule 6 Error |
| 7 | 0 | Invalid MTRR mask value |
| Class code = 4: TPM Access – BIOS AC and SINIT | | |

| Major Error Code | Minor Error Code | Description |
|---|---|---|
| 1 | TPM Error Code<br>0-0xfff | TPM returned an error. Error is reported as:<br>TPM 1.2 family error code occupies 9 bits.<br>Fatal error codes:<br>— [23:16] – error code;<br>— [24] = 0<br>Non-fatal error codes:<br>— [23:16] – error code & 0xFF;<br>— [24] = 1<br>TPM 2.0 error code occupies 12 bits.<br>All error codes are returned unmodified |
| 5 | 0 | TPM 1.2 disabled |
| 6 | 0 | TPM 1.2 deactivated |
| 0xD | 0 | TPM 2.0 interface type (FIFO/CRB) not supported |
| 0xE | 0 | TPM family (1.2/2.0) not supported |
| 0xF | 0 | Discovered number of TPM 2.0 PCR banks exceeds supported maximum (3) |
| 0x10 | 0 | Required TPM hash algorithm not supported |
| Class code = 6: Launch Control Policy – BIOS AC and SINIT | | |
| 2 | 0-X | SINIT version is below minimum specified in TPM NV policy index.<br>Minor error code contains additional details. |
| 4 | 0-4 | No match is found for Policy Element.<br>Element type is reported via minor error code. |
| 5 | 0 | Auto-promotion failed.<br>BIOS hash differs from hash value saved in AUX index. |
| 6 | 0 | Failsafe boot failed. (FIT table not found or corrupted). |
| 7 | 0-X | PO integrity check failed.<br>Minor error code contains additional details. |
| 8 | 0-X | PS integrity check failed.<br>Minor error code contains additional details. |
| 9 | 0 | No policies are defined to allow NPW execution |
| 0xA | 0 | PS TPM NV policy index is required but not defined. |
| Class code = 9: Heap Table Data -- SINIT | | |
| 1 | 0-X | Invalid size of one of the heap data tables.<br>Minor error code contains additional details. |
| 2 | 0-X | Invalid version of one of the heap data tables<br>Minor error code contains additional details. |
| 3 | 0 | Invalid PMR Low range alignment |
| 4 | 0 | Invalid PMR High range alignment |
| 5 | 0 | Invalid MLE placement (Above 4GB) |
| 6 | 0 | Invalid MLE requested capabilities |

| Major Error Code | Minor Error Code | Description |
|---|---|---|
| 7 | 0-X | Heap region is overfilled.<br>Minor error code contains additional details. |
| 8 | 0 | Unsupported heap extended element type. |
| 9 | 0 | Invalid heap extended element size. |
| 0xA | 0 | Heap table is not terminated by the extended “END” element |
| 0xB | 0-X | Invalid event log pointer.<br>Minor error code contains additional details. |
| 0xC | 0 | Invalid RSDT/RSDP pointer in *OsSinitData* table |
| Class code = 0xE: PMR Configuration -- SINIT | | |
| 1 | 0 | DMA remapping is enabled |
| 2 | 0 | Invalid PMR Low configuration |
| 3 | 0 | Invalid PMR High configuration |
| Class code = 0xF: MLE Header Check -- SINIT | | |
| 1 | 0 | MLE Header linear address conversion error |
| 2 | 0 | Invalid MLE GUID |
| 3 | 0 | Invalid MLE version |
| 4 | 0 | Invalid first page address |
| 5 | 0 | Invalid MLE size |
| 6 | 0 | Invalid MLE entry point address |
| 7 | 0 | Incompatible RLM wake-up method |
| Class code = 0x10: MLE Page Tables Check -- SINIT | | |
| 1 | 0 | Page placement error:<br>— Incorrect page alignment;<br>— Page is not in usable DMA protected DRAM;<br>Pages checked are:<br>— PDPT page<br>— PDT page;<br>— PT page;<br>— MLE page |
| 2 | 0 | MLE page order rule failure – next page is not above previous one. |
| 3 | 0 | Discovered big page (2MB) |
| 4 | 0 | Page Table order rule failure – PDPT, PDT, PT, MLE pages are not in ascending order. |
| 5 | 0 | Invalid MLE hashed size |
| 6 | 0 | Invalid RLP entry point address |
| Class code = 0x14: Event Log -- SINIT | | |
| 1 | 0 | Invalid Log Header GUID |

| Major Error Code | Minor Error Code | Description |
|---|---|---|
| 2 | 0 | Invalid Log Header version |
| 3 | 0 | Inconsistent values of header fields |
| 4 | 0 | Insufficient log size |
| 5 | 0 | Unsupported record version |

§ §

# Appendix J TPM NV

### Table 32. 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:<br>0x50000002<br>IVB after PRT and beyond platforms:<br>0x50000003 | Inter – module mail box<br>BIOSAC registration data | Platform Specific<br>LT-CX Size: 64<br>LT-SX Size: 96 | None | None | None | 3 or 4 | Any | Any |
| SGX SVN | SGX | 0x50000004 | BIOS – MLE mail box. Value of SINIT SVN and flags | Platform Specific Size: 8 | None | None | None | 3 - 0 | 3 - 0 | Any |

### Table 33. TPM Family 2.0 NV Storage Matrix

| Name | Label | Index Value | Description | Public Size in bytes | Attributes | Auth Value | Auth Policy | Locality Write | Locality Read | Locality Delete |
|---|---|---|---|---|---|---|---|---|---|---|
| LCP Platform Owner | PO | 0x1C1_0106 NOTE3 | LCP Structure for Platform Owner | Platform Specific<br>Size: >= 38 + HASHALGID_DIGEST_SIZE Note 1 | TPMA_NV_OWNERWRITE<br>TPMA_NV_POLICYWRITE<br>TPMA_NV_AUTHREAD<br>TPMA_NV_NO_DA | Empty Buffer | Owner policy | Any, controlled by Owner choice | Any | Any, controlled by Owner choice |

| Name | Label | Index Value | Description | Public Size in bytes | Attributes | Auth Value | Auth Policy | Locality Write | Locality Read | Locality Delete |
|---|---|---|---|---|---|---|---|---|---|---|
| Launch Auxiliary | AUX | 0x1C1_0102 NOTE3 | Inter – module mail box<br>BIOSAC registration data | Platform Specific Size: >= 40 + 2 * HASHALGID_DIGEST_SIZE Note 1 | TPMA_NV_POLICYWRITE<br>TPMA_NV_POLICY_DELETE<br>TPMA_NV_WRITE_STCLEAR<br>TPMA_NV_AUTHREAD<br>TPMA_NV_NO_DA<br>TPMA_NV_PLATFORMCREATE | Empty Buffer | Fixed policy Note 2 | 3 or 4 | Any | 3 or 4 |
| SGX SVN | SGX | 0x1C1_0104 NOTE3 | BIOS – MLE mail box. Value of SINIT SVN and flags | Platform Specific Size: 8 | TPMA_NV_AUTHWRITE<br>TPMA_NV_POLICY_DELETE<br>TPMA_NV_AUTHREAD<br>TPMA_NV_NO_DA<br>TPMA_NV_PLATFORMCREATE | Empty Buffer | OEM policy | Any | Any | Any, controlled by OEM policy |

***Note:***

1. $HASH_{ALGID}$_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
2. 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)

   Let policy C to be:
   C = TPM2_PolicyCommandCode (TPM_CC_NV_Write)

Then *authPolicy* can be computed as:

*authPolicy* = {{A} AND {C}} OR {{A} AND {B}}

3. 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.

TXT ACMs will support only 0x1C1_xxxx index set. Indication of supported set is provided via TPM capabilities – see Table 13 but will always be forced to "1"

§ §

# Appendix K Detailed LCP Checklist

## K.1 Policy Validation Checklist

The following checks must be performed by LCP *Policy Engine* to validate LCP data integrity:

### K.1.1 TPM NV AUX Index

1. Index must be provisioned
2. Index attributes must be per Table 32 for V2 index and per Table 33 for V3 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 32 and Table 33 for V2 and V3 structures respectively.
5. Read AUX index data and analyze content.
6. Additional verification steps can be added since AUX index content is platform specific.

### K.1.2 TPM NV PO Index

1. Index may be provisioned.
2. Index attributes must be per Table 32 for V2 index and per Table 33 for V3 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 for V2 index and per 3.3.2 for V3 index
6. *PO.HashAlg* must be per D.1 for V2 structure and one of the supported algorithms per E.1 for V3 structure.
7. TPM 2.0 mode: *PO.HashAlg* must be permitted by *PO.LcpHashAlgMask*
8. Size must be per Table 32 for V2 structure and per Table 33 for V3 structure. 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 5.
11. TPM 2.0 mode: algorithms corresponding to set bits in *PO.LcpHashAlgMask and PO.LcpSignAlgMask* must be supported by ACM per E.1 for V3 structure.

## K.1.3 NPW Mode

Detect whether system in NPW mode is allowed to run:

1. System is in NPW mode if NPW bit is asserted in *Flags* field of SINIT header or BIOS ACM ACM_SVN value is ≤ 2.
   If *AUX.ModuleFlags* value is initial "All Fs" state system is assumed to be in NPW mode.
2. If system is in NPW mode *Eff.PolicyControl.NPW_OK* bit must be asserted – see **Error! Reference source not found.**.

## K.1.4 Policy Data File

1. If *PO.PolicyType* is LCP_POLTYPE_LIST then respective PO *Policy Data File* must exist and be accessible.

## K.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 D.3.1 for V2 structures and in E.3.1 for V3 structures.
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 D.1 for V2 structure and per E.1 for V3 structure and hashing is performed over entire unsigned list data.
   b. For signed list compute digest as *ListDigest[ListNumber] = PO.HashAlg* (*ListPublicKey*), where *PS.HashAlg* is hashing algorithm in PS index per D.1 for V2 structure and per E.1 for V3 structure and *ListPublicKey* is either *PublicKey* field of RSASSA signature or concatenation of *Qx*|*Qy* public key components fields of SM2 signature – signature formats in D.3.1 for V2 signed lists and in E.3.1 for V3 signed lists.
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 PS index.

## K.1.6 PO Policy Data File List Integrity

1. *List.Version* must be per D.3.2 for V2 list and per E.3 for V3 list.

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. During looping check types of elements: V2 style list must contain only V2 style element types. V3 style list may contain mixture of both element styles.
4. Validate hash algorithms used by elements. Scan through all of the elements in list checking *Elt.HashAlgorithm* values. Abort if any is not supported by ACM per D.1 for V2 structures and in E.1 for V3 structures. In TPM 1.2 mode scan only V2 style elements while in TPM 2.0 mode scan only V3 style elements.
5. In TPM 2.0 mode only - for V3 style PCONF elements verify that *Count* field of TPMS_QUOTE_INFO structure which is part of *PCRInfo* definition equals 1.
6. 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 *PO Policy Data File*.

# K.2 Policy Enforcement Checklist

## K.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.

## K.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 V2 and V3 style lists looking for V2 style MLE elements.
   b. TPM 2.0: Scan V3 style lists only looking for V3 style MLE elements.
      i. If list is signed and *List.SignatureAlgorithm* is not permitted by *PO.LcpSignAlgMask*, skip the list.
2. Start of PO 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 5. If greater abort with error – SINIT is revoked.

b. Check *PolicyElementControl.STM_is_required* bit of matching element. If asserted but previous execution has not found STM present, exit with error.

c. MLE policy is satisfied - continue to the next element type.

3. If match is not found yet – continue to the "Start of PO matching loop".

3. End of PO 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.

## K.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 V2 and V3 style lists looking for V2 style PCONF elements.

b. TPM 2.0: Scan V3 style lists only looking for V3 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 PO 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 *PCRInfo* 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> TPM 1.2: Assumed PCONF policy is satisfied – continue to the next element type

b> TPM 2.0: If *PO.PolicyControl.PconfEnforced* bit is de-asserted, assumed PCONF policy is satisfied – continue to the next element type

c> TPM 2.0: If *PO.PolicyControl.PconfEnforced* 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 PO matching loop PASS 1
5. Start of PO matching loop PASS 2. This pass is executed for TPM 2.0 only.
   a. Repeat all steps of “PO 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 PO matching loop PASS 2”
6. End of PO matching loop PASS 2
7. At this point either singe or two matching passes were executed
   a. TPM 1.2: If PCONF_MATCH_REQUIRED_FLAG_PASS_1 is de-asserted, assumed PCONF policy is satisfied – continue to the next element type.
   b. TPM 1.2: Else PCONF_MATCH_REQUIRED_FLAG_PASS_1 is asserted - exit with error.
   c. TPM 2.0: If *PO.PolicyControl.PconfEnforced* 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.
   d. TPM 2.0: If *PO.PolicyControl.PsPconfEnforced* 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.

### K.2.4 STM Element Enforcement

1. If prior execution have not found STM in the system – (based on MSEG enable status – see Table 20) scan of STM elements is not performed. Only if matching MLE element requires STM this will generate an error exit – see K.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 V2 and V3 style lists looking for V2 style STM elements.

b. TPM 2.0: Scan V3 style lists only looking for V3 style STM elements.

i. If list is signed and *List.SignatureAlgorithm* is not permitted by *PO.LcpSignAlgMask*, skip the list.

3. Start of PO 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 PO matching loop"

4. End of PO 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

§ §