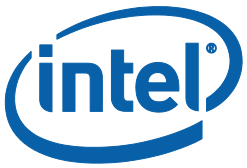


SMI Transfer Monitor (STM)

User Guide

August 2015

Revision 1.00

Contents

Figures

Tables

§

1 Introduction

1.1 Background

Intel® TXT contains several features and is intended to provide protection against several threat classes. While some features may be able to stand on their own, when taken together, the whole is greater than the sum of the parts. This document describes one of these features, namely the SMI Transfer Monitor.

Intel® TXT provides a mechanism to dynamically measure and launch a software environment in a manner that does not extend a security dependency to the pre-existing software environment. This Intel® TXT launched environment is known as the Measured Launched Environment, or MLE.

SMM (System Management Mode) is a special-purpose operating mode of the microprocessor. It is generally used for handling system-wide functions such as processor power management, platform hardware configuration, and proprietary features. The SMM environment is initialized by the BIOS prior to booting the operating system and is entered at runtime when a System Management Interrupt (SMI) is asserted. The SMI itself is neither visible to, nor maskable by the operating system. It does not use the IDT but rather uses other SMM specific CPU mechanisms to transfer control to the BIOS SMI handler. The SMI is often thought of as "transparent" to the operating system.

A normal, non Intel® TXT enabled SMI handler begins execution in an environment similar to real-address mode with the following exceptions: there are no privilege levels, there is no address mapping and SMM can address up to 4GB of memory. The SMI handler may use paging to extend its reach beyond 4GB if required. It has complete access to all I/O and control over interrupts regardless of any protections established by the OS.

The SMI handler is established by platform firmware long before the Intel® TXT launch, persists into the post-launch runtime, and by default has full access to all platform hardware.

In order to fulfill the Intel® TXT security goal that an MLE does not have a security dependency on the pre-existing software environment, the SMI handler must be dealt with.

There are fundamentally three options for satisfying this security goal:

1. Establish trust in the SMI handler.
2. Disable the SMI handler altogether when the MLE is running.
3. De-privilege the SMI handler such that it no longer can damage the integrity of the MLE.

Option 1 is appropriate for platforms where the MLE security profile allows trust in the BIOS and other system firmware, and the associated update mechanisms and

protections provided by the platform. The mechanisms for establishing trust in the platform's SMI handler are beyond the scope of this document.

Option 2 is sufficient from a security perspective, but platform design generally requires an SMI handler for proper operation. This specification describes a mechanism by which this protection mechanism is possible, but it is not expected that it will be widely used.

Option 3 is the focus of this specification. To de-privilege the SMI handler, there must be a *Supervisor SMI Handler*. That is, there must be another piece of software that can maintain the protection policy and prevent the SMI handler from accessing protected resources. At the same time, the SMI handler must have assurances that any and all hardware accesses it makes will succeed to ensure proper operation of the platform. Resolving this tension is the goal of this specification.

1.2 Overview

This *Supervisor SMI Handler* is known as the SMI Transfer Monitor (STM). The STM mediates communication regarding platform resource requirements between the BIOS SMI handler and the security requirements of the MLE and enforces runtime protections of the MLE when the BIOS SMI handler is running.

The STM can be used only when the BIOS loads it into memory and allows it to be initialized by setting a bit for all CPUs. This process of loading and setting the bit to allow the STM to be initialized is called "STM opt-in".

The negotiation for resources favors the BIOS. The SMI handler statically declares its resource requirements to the STM, and the STM is required to honor this access requirement list in its entirety. In other words, the STM must never block SMI handler access to any resource on this list.

The MLE may deem certain resources as sensitive or private and request that the STM protect these resources and exclude them from SMI handler access. The STM is required to honor these requests when the resource isn't already on the BIOS resource declaration list. It must subsequently block SMI handler access to this resource.

The STM must refuse a protection request from the MLE when the resource is on the BIOS resource declaration list. The MLE may then make a policy decision regarding how to proceed. It may be the case that the platform in question is not compatible with that particular MLE.

Once configured, the STM is entered each time an SMI occurs. The STM then invokes the BIOS SMI handler code in a VT environment such that the STM can protect system resources from being observed or modified by the BIOS SMI handler in compliance with the MLE security policy.

This document describes the interfaces, both static and dynamic, between the STM and the BIOS SMI handler, between the STM and the MLE; and describes restrictions, requirements, responsibilities, and interoperability of all three components. If written in compliance with this specification, each of the three components can be supplied by different parties without requiring any additional interfaces or side band channels.

1.3 Rationale

The platform requires correct operation of the SMI handler in order to function. This implies hardware based runtime protection of the SMI handler's critical code and data against non-SMM agents. In addition, the SMI handler may own security sensitive information and code which it wishes to protect from non-SMM code. Therefore, the SMI handler has needs for both integrity and confidentiality over its address space and memory.

Similarly, the MLE has critical code and data which must also be safe from corruption and snooping.

Due to the security need for BIOS SMI handler integrity, the SMI handler will not trust arbitrary code for protection of its integrity and privacy. This is true for all platforms, Intel® TXT enabled or otherwise.

Additionally, some Intel® TXT MLEs may consider BIOS to be an unwelcome intruder, and will not trust the BIOS SMI handler with access that could affect MLE integrity or privacy. While this may not be true for every Intel® TXT platform, it is expected to be true for at least some high assurance Intel® TXT platforms.

Since both the BIOS SMI handler and the Intel® TXT MLE have similar requirements and they are distrustful of each other, an intermediary that is trusted by both is required to negotiate and enforce ownership of platform resources. The gist of the STM is to provide this mediation service in a way that is acceptable to both the BIOS SMI handler and the MLE.

Note that the STM also has integrity requirements, and trusts neither the BIOS nor the MLE. When present, the STM runs with higher privilege than the host's VMX root and the BIOS SMI handler. The STM itself will have full access to all host visible platform resources, whether owned by the SMI handler or the MLE. Therefore, both BIOS and MLE must be able to 'opt-in' to the STM prior to granting it the highly privileged state it occupies.

Unlike the BIOS SMI handler and the MLE, however, frequent changes to the STM are not envisioned. Because of the infrequency of updates, is expected that using well known STM hashes, public verification keys, and certificates are feasible. This will facilitate a reasonable 'opt-in' policy with regard to the STM from both the BIOS and the MLE's point of view.

1.4 Terminology

Term	Description
BIOS SMI handler	The SMM code provided by the system firmware
MSEG	In the upper portion of TSEG. The MSEG region is used for the STM.
MSR	Model-Specific Register
MLE	Measured Launched Environment – This is the software environment which has been launched using the GETSEC(SENTER) process. The MLE image has been measured (e.g. using SHA-1) and registered in the TPM. This term is also used for any code operating in host (non SMM) memory when it is irrelevant whether the code is operating as a VMX root or guest.
MLE root	This is used for the code operating in the MLE in VMX root mode. This is often called the host's VMM.
MLE guest	A guest created by the MLE root. This often called the host's guest.
SINIT AC	SINIT Authenticated Code
SMI	System Management Interrupt
SMM guest	The virtual machine the BIOS SMI handler executes in when invoked from the STM
SMM	System Management Mode
SMRAM Save State	Region of memory where the SMM code can access the context of the thread interrupted by the SMI.
SMX	Safer Mode Extensions
STM	SMI Transfer Monitor
TSEG	A region of memory reserved for SMI handler code. MSEG is in the upper portion of TSEG.
VM	Virtual Machine
VMCALL	VMCALL is a VMX instruction which causes a VMX guest to exit to the VMM. Under special circumstances, it can be invoked from VMX root mode also.
VMCS	Virtual Machine Control Structure
VMX	Virtual Machine Extension

1.5 Related Documents

http://www.intel.com/products/processor/manuals/

Intel® Trusted Execution Technology Software Development Guide

Trusted Execution Technology Overview

Trusted Execution Technology Architectural Overview

http://www.uefi.org/specs/download/UEFI_Spec_2_3_Errata_C.pdf/

Framework Intel® Trusted Execution Technology (Intel® TXT) SMI Transfer Monitor (STM) Reference Code Design Specification and Integration Guide

1.6 Document Conventions

This document adheres to the following typographic convention

BEGIN - informative content – non-normative
As currently written there may be two classes of information contained in this document. Non-normative content may be interspersed within this specification for convenience to the reader. This non-normative content is demarked using shading, as this paragraph demonstrates.
END - informative content – non-normative

1.7 Processor Architecture Support

This specification supports the following processor architectures:

- 5th Generation Intel® Core™ Processors

§

2 Architectural Overview

2.1 Opt-in

The STM can only participate in the SMI if both the BIOS and the MLE opt-in to STM operations. The decision as to whether or not to opt-in is based on trustable measurements of the STM prior to STM operations.

2.1.1 BIOS Opt-in

The BIOS is responsible for loading the STM into SMRAM. It is expected that the BIOS will ensure the STM's identity and integrity are acceptable before copying it into SMRAM and only if the STM passes appropriate BIOS checks will the BIOS opt-in to the STM functionality. The BIOS STM opt-in is supported in hardware via a MSR bit that is only writable from SMM. This bit must be set identically on all CPU threads. This comprises the BIOS STM opt-in operation.

This opt-in does not directly enable the STM rather it simply states there is an STM present that is acceptable to BIOS which may become operational later, only after the MLE has completed its opt-in step.

There is no opportunity during the Intel® TXT or STM launch process for BIOS to re-evaluate the STM image. Therefore, BIOS must protect the STM image, along with the BIOS SMI handler itself, at all times prior to STM launch. This is normal SMRAM protection and is done using available hardware SMRAM protection features.

2.1.2 MLE Opt-in

There are two pieces to MLE opt-in of STM. First is a simple bit in the MLE header, indicating the MLE supports an STM. If this bit is not set, the SINIT-AC module will not configure STM prior to passing control to the MLE.

The second is the MLE evaluation of the measurement of the STM, and subsequent decision whether to continue the launch or not. The MLE obtains a trustable STM measurement via the Intel® TXT launch process. This measurement is performed by the SINIT-AC module and extended into PCR17.

The STM must be written such that TPM localities 2-4 and TXT private space registers are isolated from the BIOS SMI handler at the time the SMI is re-enabled. This enables the initial sealing secrets without the possibility of interference of the BIOS SMI handler. For example, secrets sealed to known good PCR values are inaccessible to other environments. If the launched environment is incorrect, the MLE may then opt to tear-down the environment rather than continue the launch. It is important to note that initial measurement of the MLE itself has similar properties.

2.2 Resource Negotiation

The negotiation of resource allocation between the BIOS SMI handler and MLE is as follows:

The BIOS produces a list of hardware resources that are required by the SMI handler. The STM cannot deny the SMI handler access to any of the resources on this list.

The MLE requests protection of hardware resources that must be protected from the SMI handler. The STM only denies these requests if it is in conflict with declared BIOS resource requirements.

2.3 Runtime Protection

During runtime, the STM only enforces protections of MLE trusted resources. The BIOS SMI handler continues to rely on the aforementioned hardware SMRAM protections to provide protection from the MLE. The addition of the STM must not leak any of the contents of SMRAM to non-SMM agents.

When the BIOS SMI handler is running under the supervision of an STM, any un-granted hardware access will cause a VMEXIT to the STM. The STM will then compare the trapped access with the MLE protection policy.

If the hardware access is precluded by the MLE protection policy, the STM must not permit the access to continue. The STM architecture provides for a BIOS error handler in this case, which will be invoked if the BIOS has registered one with the STM. If there is no error handler registered, the STM must reset the system.

If the hardware access is permitted by the MLE protection policy, the hardware access is allowed. Many STM performance optimizations are possible to minimize the number of protection related VM exits. Generally, such optimizations are outside the scope of this architecture document and are left to the design of the STM implementation.

The STM should exercise caution that it doesn't inadvertently change system state in the course of its operation, for example, modifying MSRs without restoring their original values.

2.4 Hardware

The SMI handler relies on a combination of hardware features to provide the isolation and protection from system software, be it an MLE, or a normal OS. This includes memory isolation features in the memory controller that prevent the SMRAM from being observed or modified by the OS or DMA agent, and an SMRR in the processor that prevents non-SMM software from manipulating cache controls to gain access to the SMRAM.

This does not change in the presence of an STM. In fact, the STM relies on these hardware features to maintain its own integrity. The STM need not know the details of this hardware, however, as the correct configuration will be verified and enforced via the SINIT-AC module during the launch process. The hardware must provide locks for all relevant configuration registers.

The registers listed here are for convenience only. Please refer to appropriate hardware specifications for official definitions.

The following chipset registers and CPU MSRs are defined for the TXT SMM architecture. They are used by the BIOS and the STM during the boot of the system.

Table 2-1: Chipset registers and CPU MSRs

Register Name	Bits	Unit	Access	Description
STS.STS.SEQ_IN_PROCESS bit	17	Chipset TXT config space	Read access by Ring 0 code	Set by hardware when logical CPUs are in SENTER or SEXIT transition. Cleared by hardware when state is synchronized.
IA32_SMM_MONITOR_CTL.VALID	0	CPU	Read access by any code; Write access only during SMM	Set by SMI handler to indicate an STM is loaded in MSEG. Cleared by SMI handler to indicate no STM is present in MSEG.
IA32_SMM_MONITOR_CTL.Reserve	1 11 -3 63 -32	CPU		
IA32_SMM_MONITOR_CTL.Bit2	2	CPU	Read access by any code; Write access only during SMM	Bit 2 determines whether executions of VMXOFF unblock SMIs under the default treatment of SMIs and SMM. Executions of VMXOFF unblock SMIs unless bit 2 is 1 (the value of bit 0 is irrelevant).
IA32_SMM_MONITOR_CTL.MSEG_BASE	31-12	CPU	Read access by any code; Write access only during SMM	Bits 31:12 of the physical base address of the MSEG memory space. Bits 11:0 are implied and are interpreted as 0.
TXT.ERRORCODE		Chipset TXT config space	Read/write access by Ring 0 code	
TXT.CMD.SYS_RESET		Chipset TXT config space	Read/write access by Ring 0 code	

2.5 BIOS

To support an STM, BIOS must do the following:

- Correctly configure SMRAM
- Load an STM image into MSEG
- Set the MSEG base address in the IA32_SMM_MONITOR_CTL.MSEG_BASE field for each processor thread in the system
- Set the opt-in to STM operation via the IA32_SMM_MONITOR_CTL.VALID bit for each processor thread in the system

- Provide a TXT_PROCESSOR_SMM_DESCRIPTOR structure in SMRAM for each processor thread in the system
- Support STM style entry points in the SMI handler
- Provide a SMI handler hardware resource requirements list
- BIOS extended data in TXT heap

BIOS may also optionally provide:

- Error handlers to help debug or resolve resource conflicts
- Independent update of STM image

2.6 MLE

To support an STM, the MLE must do the following:

- Indicate it is capable of supporting an STM
- Identify the presence of the STM
- Provide STM with dynamic updates to protection policy as resources are added to the MLE's TCB
- Start the STM
- Stop and tear down STM as part of taking down the trusted environment
- Reset the system or re-enable SMIs if the STM is not launched

2.7 STM

STM must implement the following:

- Caching support for BIOS SMI handler
- An interface used by the SMI handler to enable mapping MLE memory into the SMI handler's memory space (only when not protected)
- The interface used by the MLE to communicate with the STM

2.8 Structure Packing

All C structs in this specification are packed. There is no padding.

§

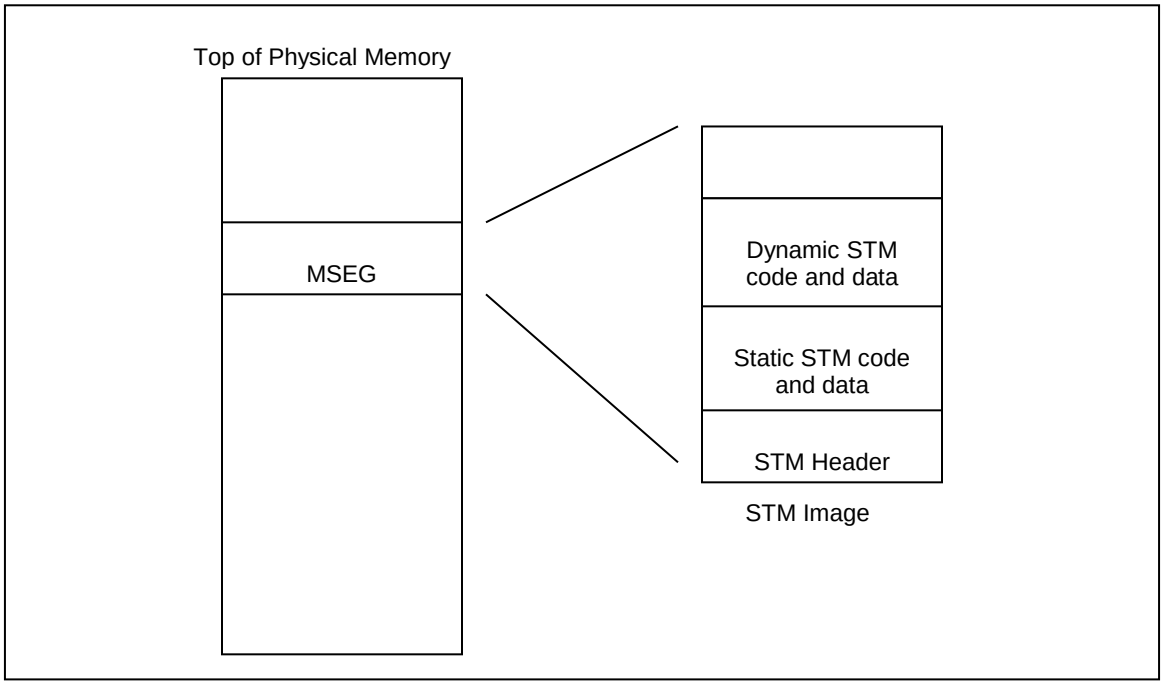
3 STM Image Format

The figure below shows the format of the STM image when loaded in the MSEG memory region. The STM image is loaded into the physically contiguous MSEG memory region by the BIOS. Placement of the STM into non-MSEG memory may conflict with other features.

The BIOS should implement some form of STM acceptability policy control regarding the setting of the IA32_SMM_MONITOR_CTL.VALID bit. For example: If the STM is stored in the system flash part and is protected via the normal BIOS update protection, then the BIOS policy could be to simply accept any STM found in the flash. On the other hand, if the STM were to be dynamically loadable (not currently supported), the BIOS may implement a list of known good hash values for STM images, or even implement a PKI scheme. Regardless of what BIOS policy is implemented, care should be taken to avoid simply "giving SMM away" to unknown arbitrary software.

IA32_SMM_MONITOR_CTL.BASE contains the physical address of the MSEG memory region. The SINIT AC module will hash the static portion of the STM image and extend this hash to the TPM.

Figure 3-1. STM Image Format



The structures below describe the STM header format. The static portion of the STM image starts at the beginning of MSEG and continues for *SwStmHdr.StaticImageSize* bytes. All fields in *HwStmHdr* are defined by the processor and are documented in the

Intel® 64 and IA-32 Architectures Software Development Manual. They are documented here only for convenience.

```
typedef struct {
      UINT32   StmHeaderRevision;
      UINT32   MonitorFeatures;
      UINT32   GdtrLimit;
      UINT32   GdtrBaseOffset;
      UINT32   CsSelector;
      UINT32   EipOffset;
      UINT32   EspOffset;
      UINT32   Cr3Offset;
      UINT8    Reserved[FILL_TO_2K];
} HARDWARE_STM_HEADER;


typedef struct {
      UINT8    StmSpecVerMajor;
      UINT8    StmSpecVerMinor;
      UINT16   Reserved;   // must be zero
      UINT32   StaticImageSize;
      UINT32   PerProcDynamicMemorySize;
      UINT32   AdditionalDynamicMemorySize;
      struct {
         UINT32   Intel64ModeSupported :1;  // bitfield
         UINT32   EptSupported         :1;  // bitfield       UINT32    BGI   :1;
// bitfield
         UINT32   BGM                  :1;  // bitfield
         UINT32   MSR                  :1;  // bitfield
         UINT32   Reserved             :27; // must be 0
      } StmFeatures;
      UINT32   NumberOfRevIDs;
      ...
      UINT32   StmSmmRevIDs[NumberOfRevIDs];
} SOFTWARE_STM_HEADER;


typedef struct {
      HARDWARE_STM_HEADER  HwStmHdr;
      SOFTWARE_STM_HEADER  SwStmHdr;
} STM_HEADER;
```

3.1 StmHeaderRevision

The *StmHeaderRevision* field identifies the revision number (and therefore the size, structure and content) of the remainder of the hardware header. It should be checked by software attempting to access the STM image (e.g. SINIT). It is checked by the VMCALL instruction when invoked from VMX root, which will GPF# if the instruction is not compatible with this specific header revision. In order to find the STM header revision supported by the processor, one can read the IA32_VMX_MISC MSR.

3.2 MonitorFeatures

Bit 0 of the *MonitorFeatures* field is the IA-32e mode SMM feature bit. It indicates whether the logical processor will be in IA-32e mode after the STM is activated.

While the processor allows for a 32 bit STM, this specification assumes IA-32e operation. Therefore, bit 0 must be set.

Bits 31:1 of the *MonitorFeatures* field are reserved and must be zero.

3.3 STM Initialization Fields

These fields describe the processor configuration settings to be used on all invocations of the STM resulting from VMCALL from VMX root. The STM self-initialization process will subsequently change these values as needed. All offsets are defined relative to the MSEG base address.

GdtrLimit: this field contains the value to be loaded into the GDTR limit.

GdtrBaseOffset: this field contains the byte offset of the value to be added with MSEG base and then loaded into the GDTR base.

CsSelector: this field contains the value to be loaded into CS when the STM is initialized.

EipOffset: this field contains the byte offset of the value to be loaded into EIP when the STM is initialized. The value loaded into EIP is a 32-bit value. This is the STM initialization entry point.

EspOffset: this field contains the byte offset of the value to be loaded into ESP when the STM is initialized. The value loaded into ESP is a 32-bit value.

Cr3Offset: this field contains the byte offset of the value to be loaded into CR3 when the STM is initialized. The value loaded into CR3 is a 32-bit value.

3.4 Reserved Field

The reserved field is for future hardware use. Reserved Field size = 2K – sizeof(previous fields in the STM header). This allows the software fields in the STM header to start at offset 2K.

3.5 StmSpecVerMajor.StmSpecVerMinor

Indicates the STM specification version of the software header. Minor versions are backward compatible within the scope of a major version with regard to structure and functional compatibility. These fields are simply binary (not BCD). The version corresponding to this specification is StmSpecVerMajor = 1h, StmSpecVerMinor = 0h.

3.6 StaticImageSize

The size, in bytes, of the static portion of the STM image. This includes the full STM Header structure itself, the STM code image, and any pre-initialized data structures included with the image. The static image will be hashed and registered during an SMX launch.

The *StaticImageSize* field, and all subsequent fields of the STM header are “software use only” fields and are used by SINIT. The hardware does not depend on these fields.

3.7 PerProcDynamicMemorySize

The minimum size, in bytes, required per processor to support the STM's normal execution. This specifically excludes the memory required for each processor's VMCS structures but does include any other dynamic memory (e.g. heap, stack, etc.) that must be allocated for each processor in the system. (The size of the VMCS varies with the processor type. The number of VMCSs corresponds to the number of processors in the system.)

3.8 AdditionalDynamicMemorySize

The maximum memory size, in bytes, required in addition to the per processor dynamic memory, to support the STM's normal execution.

3.9 MSEG Size Calculation

The total size required for MSEG can be calculated as follows:

MSEG_Minimum_Size =

SwStmHdr.StaticImageSize +

SwStmHdr.PerProcDynamicMemorySize * number of processors +

2 * VMCS size for this processor * number of processors +

SwStmHdr.AdditionalDynamicMemorySize

The following fields must be 4K aligned: SwStmHdr.StaticImageSize, SwStmHdr.PerProcDynamicMemorySize, SwStmHdr.AdditionalDynamicMemorySize, and the "VMCS size for this processor."

BIOS is free to allocate more MSEG memory. SINIT will verify that the minimum MSEG has been allocated. If MSEG is insufficient, the SMX launch will fail. SINIT will zero the allocated dynamic area during an SMX launch. SINIT will not hash or extend the allocated dynamic area. SINIT will also zero any remaining area within MSEG beyond the end of the STM allocated memory.

Note that the VMCS size is obtained from the IA32_VMX_BASIC MSR (MSR index 0x480).

3.10 StmFeatures

This 32-bit field contains information about which features the STM implements with regard to its SMM guest.

3.10.1 StmFeatures.Intel64ModeSupported

If set, this bit indicates that the STM supports 64-bit SMM guests and MLEs. STM implementations that are compliant with this specification must set this bit.

3.10.2 StmFeatures.EptSupported

If set, this bit indicates that the STM implements EPT and the SMM guest can control its own CR3 and page tables. This is the recommended implementation on all processors that support EPT.SMM.

3.10.3 StmFeatures.BGI

If set (1), indicates Byte Granular MMIO range support. If clear (0), indicates page granular MMIO range support. See 12.

3.10.4 StmFeatures.BGM

If set (1), indicates Byte Granular Memory range support. If clear (0), indicates page granular memory range support. See 12.

3.10.5 StmFeatures.MSR

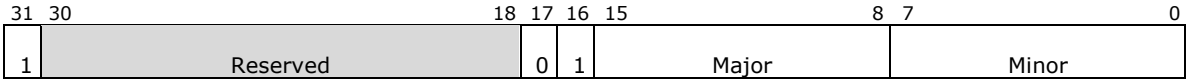
If set (1), indicates bit granular MSR resource support. If clear (0), indicates whole MSR granular resource support only. See 12.

3.11 NumberOfRevIDs

The number of revision IDs supported by this STM. The supported revision IDs are contained in the *StmSmmRevId* array. The *NumberOfRevIDs* field must be 1 or greater.

3.12 StmSmmRevIds

Each field in the *StmSmmRevIds* array indicates the SMM revision IDs that this STM supports. The *StmSmmRevIds* fields are defined as follows:



Bit 31 is always set for revision ID associated with the STM. Bits 30 to 18 are reserved and must be 0. Bit 17 indicates support for SMBASE relocation capability – this bit must be cleared to zero for STM generated SMM save states. Bit 16 must be set to 1 for STM generated save states – this bit indicates support for I/O instruction restart.

STM generated SMRAM save states must be backward-compatible through all minor revisions within the domain of any given major revision. Major revisions are not required to be compatible with each other. See section 10.1.1.

The intended use of this field is to enable software to select an STM that is compatible with the platform in cases where the platform allows dynamic loading of an STM (not currently supported.) The launching software will compare the *RequiredStmSmmRevId* value read from the BIOS extended data in the TXT heap with the *StmSmmRevId* field in the STM header. If the launching software cannot identify an STM that has the

same major revision and a minor revision greater than or equal to the *RequiredStmSmmRevId*, then no STM can be loaded. It is also expected that the BIOS will check the STM header when loading an STM to verify the STM satisfies any BIOS requirement.

§

4 BIOS Management of STM

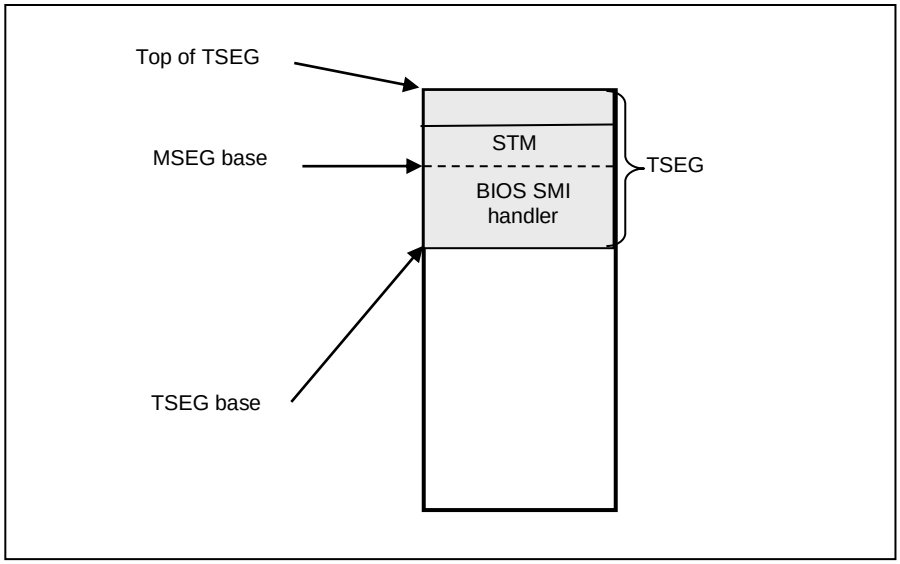
4.1 TSEG and MSEG

PC memory controller designs support several memory regions which can be reserved for SMM use and hidden from all other operating modes. TSEG is one of these memory regions, and while implementations vary to some degree, it is generally located near the top of physical memory.

For Intel® TXT STM usage, the upper part of TSEG is subdivided by a region called MSEG. The MSEG resides in the upper portion of TSEG and is owned and occupied by the STM and related Intel® TXT data. The lower portion of TSEG that is not part of MSEG is owned and occupied by the BIOS SMI handler code and data.

The following figure depicts a simplified memory map and shows the relationship between TSEG and MSEG.

Figure 4-1. TSEG and MSEG



The system BIOS must ensure the TSEG space is sufficiently large to contain both the BIOS SMM code and the STM. The STM size determines the MSEG size requirements. The total size of TSEG is equal to the sum of the space required for the BIOS SMI handler code, MSEG, and any space from the top of MSEG to the top of TSEG.

4.2 Configuring MSEG

As part of its normal boot up operation, BIOS establishes the SMI handler and locks the SMRAM configuration state using a chipset-specific locking mechanism. Prior to

locking the SMRAM configuration, BIOS must configure the MSEG region in the chipset. Additionally, before handing off control to the operating system, BIOS must populate the MSEG region with the STM image and program the IA32_SMM_MONITOR_CTL MSR on each thread.

4.2.1 MSEG Chipset Configuration

The IA32_SMM_MONITOR_CTL.MSEG_BASE defines the lower bound of MSEG. The upper bound of MSEG is defined in a product specific manner that is known to SINIT and to BIOS but is not architecturally defined within the scope of this specification. The calculation for determining the amount of MSEG space the STM requires is given in section 3.9. Prior to locking down the SMM configuration registers, BIOS must program these registers such that:

- IA32_SMM_MONITOR_CTL.MSEG_BASE (on each processor thread) is aligned on a 4K boundary.
- The STM image is completely contained within MSEG
- MSEG is completely contained within TSEG

Any violation of these rules will cause the SINIT-AC module to write an appropriate value to the TXT.ERRORCODE register, followed by a TXT.CMD.SYS_RESET to force a system reset. See the appropriate SINIT-AC module release documentation for SINIT-AC module error codes.

It is generally expected that the BIOS will carry the STM binary in the system flash part, most likely in compressed form. Therefore, BIOS should be able to determine the appropriate MSEG configuration settings prior to locking down the SMRAM configuration.

4.2.2 MSEG CPU Configuration

In addition to the chipset registers, BIOS must program the CPU's IA32_SMM_MONITOR_CTL MSR (index 9BH). In a system with more than one CPU socket, an Intel® multi-core processor, or a processor supporting Intel® Hyper-Threading Technology (Intel® HT Technology), all processor threads must have identical IA32_SMM_MONITOR_CTL MSR settings. Furthermore, these settings must be consistent with chipset MSEG programming. For any SINIT-AC module supporting an STM, an inconsistency between the IA32_SMM_MONITOR_CTL MSR on any logical processor and the chipset's MSEG registers will cause the SINIT-AC module to write an appropriate value to the TXT.ERRORCODE register, followed by a write to TXT.CMD.SYS_RESET to force a system reset.

4.2.3 Populating MSEG with the STM

Prior to passing control the OS loader, BIOS is expected to place the STM image into the MSEG region. While it is likely that this will be done during TXT register configuration, the only hard requirement is that this be done prior to the first invocation of GETSEC[SENTER], which is likely after INT19h or the invocation of an EFI OS loader. BIOS should take care to be sure it is not possible for an unknown STM to be populated into the MSEG region.

Once BIOS has placed the STM image into MSEG and locked the SMM configuration, BIOS must set IA32_SMM_MONITOR_CTL:VALID (bit 0) on all CPU threads to indicate a valid STM is present in MSEG. This bit can only be written from SMM, and indicates the BIOS opt-in to the STM residing in MSEG.

4.3 Heap Extension to BIOS Data Table

BEGIN - informative content – non-normative

```
#define HEAP_EXTDATA_TYPE_STM     4
typedef struct {
   UINT8  StmSpecVerMajor;     /* <major>.<minor> current = 0x00010000 */
   UINT8  StmSpecVerMinor;
   UINT16 BiosSmmFlags;
   UINT16 StmFeatureFlags;
   UINT32 RequiredStmSmmRevId;
   UINT32 Reserved;
   UINT8  GetBiosAcStatusCmd;
   UINT8  UpdateBiosAcCmd;
   UINT8  GetSinitAcStatusCmd;
   UINT8  UpdateSinitAcCmd;
   UINT8  GetStmStatusCmd;
   UINT8  UpdateStmCmd;
   UINT8  Reserved[26];
} HEAP_BIOS_EXT_ELEMENT;
```

The STM data that was in the ITXT table will be instead presented as an ExtDataElement. See HEAP_BIOS_EXT_ELEMENT above.

END - informative content – non-normative

BIOS allocates the TXT heap memory range and populates a portion of it with BIOS specific values. This region is known as the TXT BIOS Data Table.

The STM relevant information is delivered to pre-launch software via a TXT Heap Extended Data Element in the TXT BIOS Data Table. This extended data element is used to declare some static information as well as a number of SMI-based interfaces that can report the revisions of Intel® TXT platform software components (including the STM) and independently update the Intel® TXT-related binaries carried by the platform.

An IO port written by software that causes the chipset to generate an SMI is used to invoke services from the BIOS that are declared in the STM extended data element.

The STM extended data element is optional and is not required for Intel® TXT platform compatibility. However, if it is not present, pre-launch software has no way of applying any pre-launch policy regarding platform-related Intel® TXT software components. Nor is it possible to update any of the Intel® TXT platform software components in a non-proprietary manner. Therefore, inclusion of the STM extended data element is recommended.

Note that the interfaces and data declared in the STM extended data element are informational only and should not be considered to have any security value. The ACPI

Fixed ACPI Description Table (FADT) contains a field called SMI_CMD, which declares a system IO port that will generate a synchronous SMI when it is written. The STM Extended Data Element declares a set of values which can be written to the SMI_CMD port for Intel® TXT and MSEG/STM management.

There are 32 possible commands defined by this table. Each byte value to write to the SMI_CMD port is specified in one of 32 command slots at table offsets 14 through 45. A value of zero indicates there is no command in that slot, or the command is not supported. Therefore, supported commands have values between 0x01 and 0xFF.

Any command slot not explicitly defined by this specification is reserved and must contain the value 0.

As with ACPI all defined writes to the SMI_CMD port, these writes to the SMI_CMD port must be done by the bootstrap processor.

Table 4-1. BIOS Extended data in TXT heap

Field	Byte Length	Byte Offset	Description
StmSpecVer			
Major	1	0	Major version of the STM Heap Element; also serves to version this table.
Minor	1	1	Minor version of the STM Heap Element; also servers to version this table.
TxtSmmFeatureFlags			
BiosSmmFlags	2	2	Indicates BIOS SMI handler attributes.
StmFeatureFlags	2	4	Indicates attributes of BIOS supplied STM.
RequiredStmSmmRevId	4	6	
Reserved	4	10	Reserved – must be 0
SMI_CMD Values			
GetBiosAcStatusCmd	1	14	The value to write to SMI_CMD to determine BIOS AC module status.
UpdateBiosAcCmd	1	15	The value to write to SMI_CMD to update the BIOS AC module.
GetSinitAcStatusCmd	1	16	The value to write to SMI_CMD to determine SINIT-AC module status.
UpdateSinitAcCmd	1	17	The value to write to SMI_CMD to update the SINIT-AC module.
GetStmStatusCmd	1	18	The value to write to SMI_CMD to determine STM and MSEG status.
UpdateStmCmd	1	19	The value to write to SMI_CMD to update the STM.
Reserved	20	20	Reserved, must be 0.

Field	Byte Length	Byte Offset	Description
ReservedForDebug	6	40	These six entries are reserved for use by pre-production STM/BIOS functions. They must be 0 in production systems, but may support additional features to facilitate development and debugging of an STM in pre-production platforms.

4.3.1 Version

The StmSpecVer field is divided into Major and Minor subfields. The combination of Major.Minor describes version of this STM specification to which the included STM is compatible. It also implicitly specifies the layout of the STM Heap Element. The structure above is represented by the Major.Minor value of 1.0.

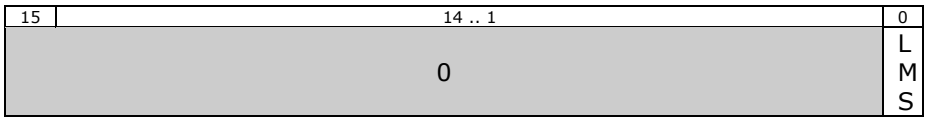
All future minor revisions within the scope of a major revision must be backward compatible. Future major revision changes indicate a compatibility break with regard to the structure definition.

4.3.2 TxtSmmFeatureFlags

The `TxtSmmFeatureFlags` field is divided into two parts. The first 16 bits (`BiosSmmFlags`) describe the BIOS SMI handler capabilities. The second 16 bits (`StmFeatureFlags`) describe the capabilities of the STM that is present in MSEG.

4.3.2.1 BiosSmmFlags

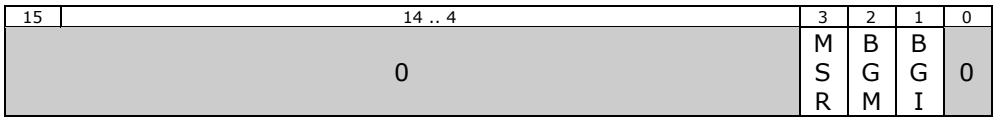
The `BiosSmmFlags` field is defined as follows:



If the LMS (Long Mode Support) bit is set, this indicates 64 bit operation is supported by the BIOS SMI handler and all STM-to-BIOS entry points are long mode entry points. If LMS is clear, then 64 bit operation is not supported and all STM-to-BIOS entry points are 32 bit entry points.

4.3.2.2 StmFeatureFlags

This field is intended for use by pre-launch software to determine the capabilities of an STM that is loaded in MSEG.
MLE root software should use the values returned from the InitializeProtectionVMCALL(). See section 9.1.
The `StmFeatureFlags` field is defined as follows:



15	14 .. 4	3	2	1	0
0		MSR	BGM	BGI	0

BGI: If set (1), indicates Byte Granular MMIO range support. If clear (0), indicates page granular MMIO range support. See 12.

BGM: If set (1), indicates Byte Granular Memory range support. If clear (0), indicates page granular memory range support. See 12.

MSR: If set (1), indicates bit granular MSR resource support. If clear (0), indicates whole MSR granular resource support only. See 12.

4.3.3 RequiredStmSmmRevId

The `RequiredStmSmmRevId` field indicates the minimum STM revision the platform requires. This field is only used if the platform supports dynamic loading of an STM by software (not currently supported.) Software will compare the value of `RequiredStmSmmRevId` with corresponding field in the STM header when selecting an STM to load.

4.3.4 BIOS Hosted TXT Components

BIOS may manage a number of TXT relevant binaries: the BIOS AC module, the SINIT AC module, and the STM binary. The following commands are all similar and consist of a status and update API for each of these binary types. The parameters passed, actions taken, and results returned are generally identical, only the image target itself changes.

Related definitions:

```
#define SHA1       1
#define SHA256     2
typedef struct {
   UINT64     BiosComponentBase;
   UINT32     ImageSize;
   UINT32     HashAlgorithm; // SHA1 or SHA256
   UINT8      Hash[32];
} TXT_BIOS_COMPONENT_STATUS;
```

`Hash` contains the cryptographic hash of the image. The hash algorithm used is given by `HashAlgorithm`.

```
#define PAGE_SIZE 4096
typedef struct {
   UINT32      ImageSize;
   UINT32      Reserved;
   UINT64      ImagePageBase[NumberOfPages];
} TXT_BIOS_COMPONENT_UPDATE;
```

The `ImagePageBase[]` field is an array of page aligned physical addresses, where the image being passed in is broken into page sized blocks (except for the last page). The image itself is reconstructed by BIOS by concatenating all pages in the image.

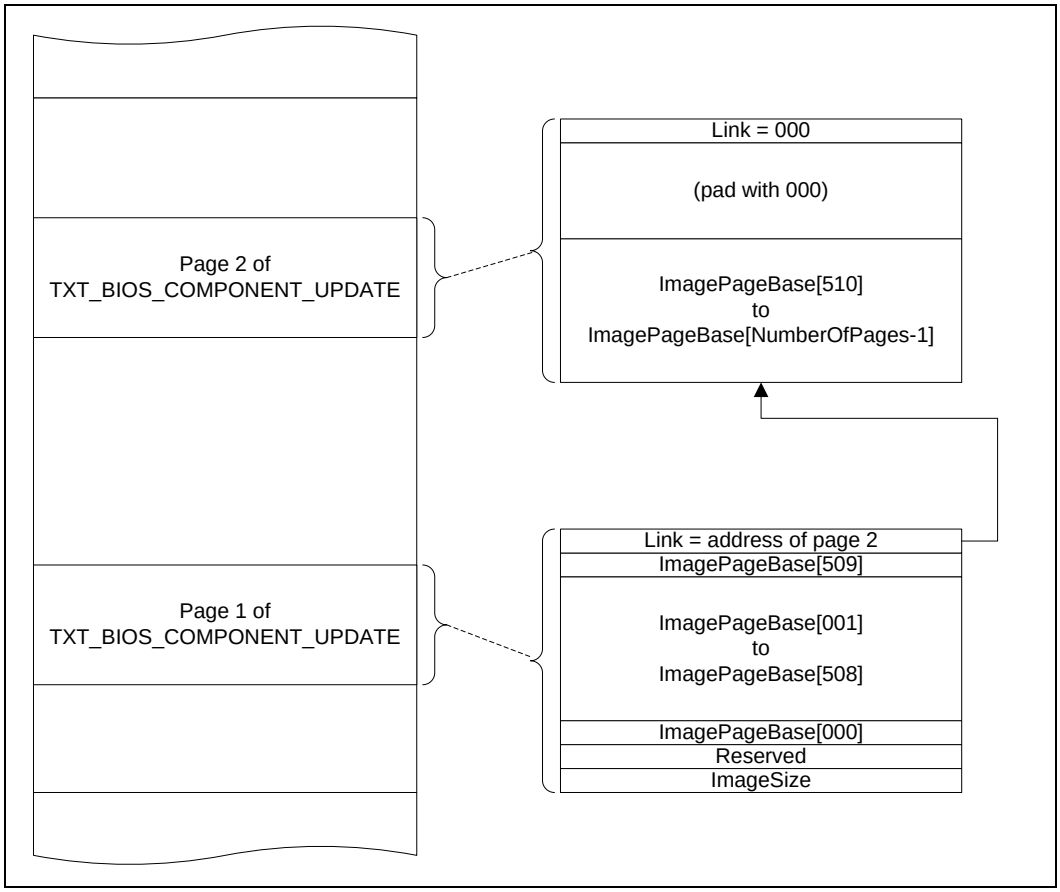
The total space for the image is given by `ImageSize`.

The total number of pages passed in is given by the following algorithm:

```
If (ImageSize % PAGE_SIZE == 0) {
   NumberOfPages = ImageSize / PAGE_SIZE
} else {
   NumberOfPages = ImageSize / PAGE_SIZE + 1
}
```

If `TXT_BIOS_COMPONENT_UPDATE` were constrained to fit entirely within one 4K page, the maximum `ImageSize` would be 511 4K pages, which is slightly smaller than 2M. Because 2M may not be sufficient for all future platforms, `TXT_BIOS_COMPONENT_UPDATE` can be larger than one page. The last 8 bytes in each page of a `TXT_BIOS_COMPONENT_UPDATE` structure are defined to be a physical pointer to another page where the `ImagePageBase[]` field continues. If this link is not used, the last 8 bytes of the page must contain zero. The last 8 bytes never contain an element of `ImagePageBase[]`. If there are unused elements of `ImagePageBase[]` array, these must be padded with zero until the end of the page. This allows chaining of pages and arbitrarily large images to be passed in. See the example below:

Figure 4-2. Example TXT_BIOS_COMPONENT_UPDATE



4.3.4.1 GetBiosAcStatusCmd

`GetBiosAcStatusCmd` returns data regarding the state of the BIOS ACM. While it should be correct, no security or trust is implied with the data returned.

System software invokes `GetBiosAcStatusCmd` by writing the associated value to the SMI_CMD port declared in the FADT ACPI table. A `GetBiosAcStatusCmd` value of zero indicates this command is not supported and must not be attempted.

The `GetBiosAcStatusCmd` takes a 64-bit physical pointer to a caller-allocated `TXT_BIOS_COMPONENT_STATUS` structure in ECX:EBX. This structure is used for output only. The SMI handler must verify the requested output buffer is not within SMRAM. Other than that, no input parameter or state checking by SMI handler is required.

Input registers:

EBX = low 32 bits of host physical address of caller-allocated `TXT_BIOS_COMPONENT_STATUS` structure. The `TXT_BIOS_COMPONENT_STATUS` structure itself is an output.

ECX = high 32 bits of host physical address of caller-allocated TXT_BIOS_COMPONENT_STATUS structure. If TxtSmmFeatureFlags.LMS is clear, ECX must be 0. The TXT_BIOS_COMPONENT_STATUS structure itself is an output.

Output registers:

CF = 0, EAX = SMM_SUCCESS: No error

CF = 1, EAX = ERROR_SMM_BAD_BUFFER: TXT_BIOS_COMPONENT_STATUS structure overlapped SMRAM

CF = 1, EAX = ERROR_SMM_UNSPECIFIED: An unspecified error occurred.

All other registers unmodified.

4.3.4.2 UpdateBiosAcCmd

UpdateBiosAcCmd takes a BIOS AC image as a parameter and stores it in the platform for subsequent use. The UpdateBiosAcCmd is intended to provide a facility for updating the BIOS ACM without requiring a full BIOS update.

System software invokes UpdateBiosAcCmd by writing the associated value to the SMI_CMD port declared in the FADT ACPI table. An UpdateBiosAcCmd value of zero indicates this command is not supported and must not be attempted.

The UpdateBiosAcCmd takes a 64-bit physical pointer to a caller allocated TXT_BIOS_COMPONENT_UPDATE structure in ECX:EBX. This structure indicates the location of the new BIOS ACM to be written.

Input registers:

EBX = low 32 bits of host physical address of caller allocated TXT_BIOS_COMPONENT_UPDATE structure.

ECX = high 32 bits of host physical address of caller allocated TXT_BIOS_COMPONENT_UPDATE structure. If TxtSmmFeatureFlags.LMS is clear, ECX must be 0.

Output registers:

CF = 0, EAX = SMM_SUCCESS: No error

CF = 1, EAX = ERROR_SMM_BAD_BUFFER: TXT_BIOS_COMPONENT_UPDATE structure or BIOS AC image itself was not valid

CF = 1, EAX = ERROR_SMM_UNSPECIFIED: An unspecified error occurred.

4.3.4.3 GetSinitAcStatusCmd

GetSinitAcStatusCmd has the same semantics as GetBiosAcStatusCmd, except that the target is the SINIT AC module rather than the BIOS AC module.

4.3.4.4 UpdateSinitAcCmd

UpdateSinitAcCmd has the same semantics as UpdateBiosAcCmd, except that the target is the SINIT AC module rather than the BIOS AC module.

4.3.4.5 GetStmStatusCmd

GetStmStatusCmd has the same semantics as GetBiosAcStatusCmd, except that the target is the STM binary rather than the BIOS AC module.

4.3.4.6 UpdateStmCmd

UpdateStmCmd has the same semantics as UpdateBiosAcStatusCmd, except that the target is the STM binary rather than the BIOS AC module.

§

5 STM Launch

Pre-SENTER system software is responsible for invoking GETSEC[SENTER] to launch the MLE. See the published Intel® Trusted Execution Technology documents for details on this process.

5.1 IA32e mode STM

See "Enabling the Dual-Monitor Treatment" in the Intel® 64 and IA-32 Architectures Software Development Manual for details regarding the MSEG header.

Bytes 7:4 of the MSEG header contain the SMM-monitor features field. Bits 31:1 of this field are reserved and must be zero. Bit 0 of the field is the IA-32e mode SMM feature bit. If set, it indicates the logical processor will be in IA-32e mode after the STM is activated.

This architecture assumes IA-32e mode operation for a TXT launched STM. Therefore, the IA-32e mode SMM feature bit must be set. This allows for interoperation with both 32 bit and 64 bit host environments, and 32 bit and 64 bit BIOS environments.

To facilitate STM bootstrapping, the initial STM pages must be identity mapped. The initial STM page tables are generated by SINIT and populated by SINIT into 6 contiguous pages within the STM's image. The location of these six pages is indicated by the CR3 offset field in the MSEG header. The space for these six pages must not be within the measured portion of MSEG. Rather, the CR3 offset should be in the dynamic range. This is necessary to avoid measurement of these pages, since they are not actually part of the STM. (See section 3.) These initial page tables will identity map all memory between 0-4G. SINIT is free to choose any format of page table that is compatible with IA32e mode and fits within 6 contiguous pages. Subsequent execution of the STM is free to modify the page tables as necessary within the bounds of IA32e paging.

5.2 The STM launch process

There are several basic steps to enabling and configuring an STM:

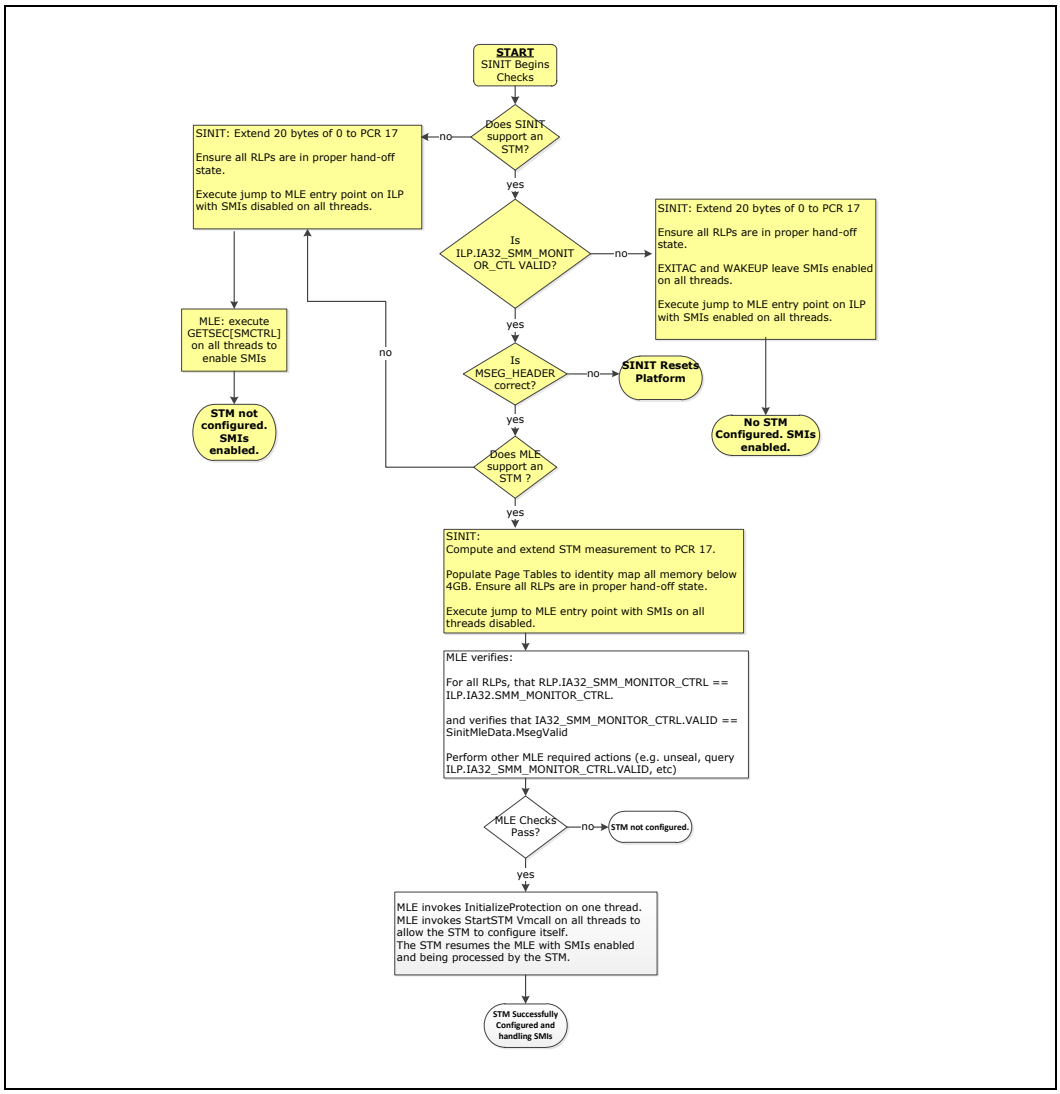
Perform normal TXT launch of MLE via SENTER.

BEGIN - informative content – non-normative

The flow is included in this specification since the behavior of the SINIT-AC module is relevant to the MLE.

Prior to transferring control to the MLE, the SINIT-AC module will perform the following actions depicted below in the flow chart related to the STM.

Figure 5-1. STM Launch Process



END - informative content – non-normative

5.3 SINIT-AC module handoff to MLE

5.3.1 CPU

The CPU state on entry into the MLE and early MLE bring-up is described in the TXT Software Development Guide. The only significant difference introduced by the STM is that when all SINIT-AC module STM checks pass (indicating there is an STM ready for configuration present in memory), SMIs are masked when the MLE gets control.

5.3.2 PCR values in the presence/absence of an STM

SINIT measures the STM into PCR[17]. See the MLE Writer's Guide for details.

Establishing initial trust in the STM is similar to establishing trust in the MLE and is beyond the scope of this specification.

5.4 Internal STM initialization

This process begins when the MLE root invokes InitializeProtectionVMCALL() and ProtectResourceVMCALL() to initialize and create the MLE protection profile. Once these steps have been completed, the MLE root invokes StartStmVMCALL() on all processors. When the StartStmVMCALL() returns execution control to the MLE, the STM will have been configured and started, and will be servicing SMIs. See section 9.1 for details.

During configuration, the STM will perform the following steps:

1. Initialize resource list data structures (InitializeProtectionVMCALL())
2. Build MLE protection profile (ProtectResourceVMCALL())
3. Initialize operational VMX data structures and take control of SMIs (StartStmVMCALL()):
    a) Rendezvous all CPUs
    b) Enable caching of TSEG (if not automatic via SMRR)
    c) Create two VMCS structures for each CPU. One for the SMM guest and one for the MLE. If the CPU supports it, set the "Save VMX-preemption timer value" bit in the VM-Exit Controls of the VMCS that the STM creates for the MLE. Note: The STM should exercise caution over setting up the VMCSs and not trust the values from the MLE's VMCS. For example, STM should set correct value for VM-exit MSR-load/VM-entry MSR-load/VM-exit MSR-store. If those fields are NOT used, STM should zero out them.
    d) Launches the SMM guest using the entry point described by the `TXT_PROCESSOR_SMM_DESCRIPTOR.SmmStmSetupRip` (if specified) for each CPU as defined by BIOS.
    e) Disable caching of TSEG (if not automatic via SMRR)
    f) Launch host software guest with SMIs enabled and STM configured. The STM enables SMIs by clearing the "Blocking By SMI" bit (bit 2) in the GUEST_INTERRUPTIBILITY VMCS field (Encoding 0x4824H) for the VMCS that the STM created for its MLE guest. The STM returns control to the MLE via VMLAUNCH.

Subsequent SMIs will be delivered to the STM, which will forward the SMIs to the SMI handler via a VMRESUME to the `TXT_PROCESSOR_SMM_DESCRIPTOR.SmmSmiHandlerRip`.

§

6 STM Runtime

6.1 CPU state

Prior to configuring an Intel® TXT STM environment, the BIOS SMM code located in TSEG services SMIs directly. After an Intel® TXT STM environment has been configured, the BIOS SMM code continues to service SMIs, but runs within a virtual machine (SMM guest) hosted by the STM.

When the BIOS SMI handler runs in the context of an STM-hosted VM, it uses a paged, protected mode entry point for VM entry, rather than the traditional, big-real mode SMI entry point. The STM uses the paging features to exclude MLE protected pages from BIOS SMI handler access.

Since the SMM guest runs in protected mode with paging enabled (CR0.PE=1 and CR0.PG=1), it must not attempt to clear CR0.PE or CR0.PG. Attempts by the SMM guest to clear CR0.PE or CR0.PG must be blocked by the STM and will result in a protection exception (see below for discussion of protection exceptions) and the STM logging a control register violation (if specified in the event bitmap) in the Event Log

The BIOS declares the SMM guest entry state for each processor via a `TXT_PROCESSOR_SMM_DESCRIPTOR` data structure that is located in SMRAM at SMBASE + 0xFB00.

The STM shall refer to the IA32_SMBASE MSR to determine the location of SMBASE.

Except for the `SmramToVmcsRestoreRequired` and `ReinitializeVmcsRequired` bits which are evaluated on every RSM, the STM uses these data only when setting up the VMCS structures for each processor prior to the initial launch of the SMM guest and does not re-evaluate `TXT_PROCESSOR_SMM_DESCRIPTOR` for each SMI. Therefore, other than the `SmramToVmcsRestoreRequired` and `ReinitializeVmcsRequired` bits, these data should be considered static and not be modified by the BIOS during runtime.

```
typedef struct {
  UINT64 Signature;
  UINT16 Size;
  UINT8 SmmDescriptorVerMajor;
  UINT8 SmmDescriptorVerMinor;
  UINT32 LocalApicId;

  // The following byte defines SMI handler entry state
  struct {
    UINT8  ExecutionDisableOutsideSmrr : 1;   // bitfield
    UINT8  Intel64Mode : 1;                   // bitfield
    UINT8  Cr4Pae : 1;                        // bitfield
    UINT8  Cr4Pse : 1;                        // bitfield
    UINT8  Reserved1 : 4;                     // bitfield
  } SmmEntryState;

  // The following byte defines data passed back to STM on RSM

  struct {
```

```
    UINT8  SmramToVmcsRestoreRequired : 1;// bitfield - BIOS restore hint
    UINT8  ReinitializeVmcsRequired : 1;      // bitfield - BIOS request
    UINT8  Reserved2 : 6;                     // bitfield

  } SmmResumeState;


  // the following byte defines inputs from STM to SMI handler
  struct {
    UINT8  DomainType : 4;           // bitfield - STM input to BIOS on each SMI
    UINT8  XStatePolicy : 2;         // bitfield - STM input to BIOS on each SMI
    UINT8  EptEnabled : 1;           // bitfield
    UINT8  Reserved3 : 1;            // bitfield
  } StmSmmState;

  UINT8  Reserved4;

  UINT16 SmmCs;
  UINT16 SmmDs;
  UINT16 SmmSs;
  UINT16 SmmOtherSegment;
  UINT16 SmmTr;
  UINT16 Reserved5;

  UINT64 SmmCr3;
  UINT64 SmmStmSetupRip;                  // optional, may be zero
  UINT64 SmmStmTeardownRip;               // optional, may be zero
  UINT64 SmmSmiHandlerRip;
  UINT64 SmmSmiHandlerRsp;
  UINT64 GdtPtr;
  UINT32 GdtSize;
  UINT32 RequiredStmSmmRevId;

  STM_PROTECTION_EXCEPTION_HANDLER  StmProtectionExceptionHandler;
  UINT64 Reserved6;
  UINT64 BiosHwResourceRequirementsPtr;
  UINT64 AcpiRsdp;
  UINT8  PhysicalAddressBit;
} TXT_PROCESSOR_SMM_DESCRIPTOR;

typedef struct {
  UINT64 SpeRip;
  UINT64 SpeRsp;
  UINT16 SpeSs;
  struct {
    UINT16 PageViolationException : 1;        // bitfield
    UINT16 MsrViolationException : 1;         // bitfield
    UINT16 RegisterViolationException : 1;    // bitfield
    UINT16 IoViolationException : 1;          // bitfield
    UINT16 PciViolationException : 1;         // bitfield
    UINT16 Reserved1 : 11;                    // bitfield
  };
  UINT32 Reserved2;
} STM_PROTECTION_EXCEPTION_HANDLER;

#define TXT_PROCESSOR_SMM_DESCRIPTOR_SIGNATURE \ // 'TXTPSSIG'
  (('G' << 56) | \
   ('I' << 48) | \
   ('S' << 40) | \
   ('S' << 32) | \
   ('P' << 24) | \
   ('T' << 16) | \
   ('X' << 8)  | \
   ('T'))
#define TXT_PROCESSOR_SMM_DESCRIPTOR_VERSION_MAJOR        1
#define TXT_PROCESSOR_SMM_DESCRIPTOR_VERSION_MINOR        0
```

```
#typedef enum {
  TXT_SMM_PAGE_VIOLATION=1,
  TXT_SMM_MSR_VIOLATION,
  TXT_SMM_REGISTER_VIOLATION,
  TXT_SMM_IO_VIOLATION,
  TXT_SMM_PCI_VIOLATION
} TXT_SMM_PROTECTION_EXCEPTION_TYPE;
```

`Signature` must be set to `TXT_PROCESSOR_SMM_DESCRIPTOR_SIGNATURE`.

`Size` Indicates the size in bytes of the `TXT_PROCESSOR_SMM_DESCRIPTOR`.

`SmmDescriptorVerMajor and SmmDescriptorVerMinor` indicate the version (specification compatibility) of the `TXT_PROCESSOR_SMM_DESCRIPTOR` structure. Minor versions in a given major version number are backwards compatible. They share the same access semantics and any new fields in the data structure are appended to the end. Major versions are not backwards compatible and may have differences in the `TXT_PROCESSOR_SMM_DESCRIPTOR` data structure and access semantics from the previous major version. For the data structure described here, the `SmmDescriptorVerMajor` value must contain the value `TXT_PROCESSOR_SMM_DESCRIPTOR_VERSION_MAJOR` (1). Future specifications may re-define the remainder of the structure and its access semantics by changing the major version in `SmmDescriptorVerMajor`. System software must check this version number before assuming the format of the rest of this data structure. Any future revisions to the `TXT_PROCESSOR_SMM_DESCRIPTOR` data structure must update the version values in a manner that is monotonically increasing with the previous version and must also update the `Size` appropriately.

`LocalApicId` Indicates which processor the `TXT_PROCESSOR_SMM_DESCRIPTOR` instance is associated with. On platforms that support x2APIC, this corresponds to the x2APIC ID. On platforms that don't support x2APIC, this corresponds to the initial local APIC id which is stored in the low byte of the `LocalApicId` with the upper 3 bytes cleared to 0.

`ExecutionDisableOutsideSmrr` If set, this indicates the BIOS SMM code never executes code outside of the SMRR so it is safe to prevent execution by BIOS SMI handler in all non-SMRAM pages. The STM must enforce this policy.

`Intel64Mode` Indicates the SMI handler is an Intel 64 enabled handler and all entry points are 64 bit code. This bit corresponds to IA32_EFER.LME.

`Cr4Pae` Indicates the SMI handler uses PAE. This bit must be set if Intel64Mode is set, since when EFER.LME is set PAE must also be enabled. If Intel64Mode is clear, this bit is used to determine whether BIOS uses 32 bit paging, or PAE paging.

`Cr4Pse` Indicates the BIOS SMI handler uses PSE. Ignored unless 32 bit paging is used (both Intel64Mode and Cr4Pae are clear)

`SmramToVmcsRestoreRequired` The SMI handler sets this bit in order to indicate to the STM that a change to SMRAM state save has been made that needs to be propagated back to the interrupted context's VMCS and process register state. If this bit is not set by the SMM guest, the STM will not attempt to propagate any changes the SMI handler has made to the SMRAM state save area to the interrupted context's VMCS or process register state. This bit is evaluated by the

STM after every RSM prior to resuming the interrupted context. This bit is always cleared by the STM on all threads prior to resuming the interrupted context.

ReinitializeVmcsRequired The SMI handler sets this bit in order to indicate to the STM that upon the next SMI the full BIOS VMCS state should be initialized. Therefore, if on entry into the STM this bit is set, the STM will re-initialize the BIOS VMCS based on the contents of the TXT_PROCESSOR_SMM_DESCRIPTOR. However, if on entry into the STM this bit is clear, the STM will re-initialize only RIP and RSP. This bit must be set by the POST BIOS to ensure the STM will properly initialize the BIOS VMCS on the first SMI handled by the STM. This bit is cleared by the STM after every RSM prior to resuming the interrupted context.

DomainType The STM indicates to the BIOS SMI handler what SMRAM state save area scrubbing policy has been applied. In all cases, the SMM_REV_ID fields of the SMRAM state save area are valid. Consult section 10.3.3 for additional details.

XStatePolicy Indicates the active extended register state access policy. Values are XSTATE_READWRITE, XSTATE_READONLY, and XSTATE_SCRUB. See section 9.7 for details.

EptEnabled Indicates if STM enabled EPT for SMM guest. If EPT is enabled, BIOS is allowed to modify CR3 and control its page tables directly. The state of this bit must be consistent across all processors and must not change between SMI events.

SmmCs Indicates the initial value of CS on entry to the SMM guest. SmmCs must contain a valid selector for the GDT referenced by SmmGdtPtr.

SmmDs Indicates the initial value of DS on entry to the SMM guest. SmmDs must contain a valid selector for the GDT referenced by SmmGdtPtr.

SmmSs Indicates the initial value of SS on entry to the SMM guest. SmmSs must contain a valid selector for the GDT referenced by SmmGdtPtr.

SmmOtherSegment Indicates the initial value of ES, FS, and GS on entry to the SMM guest. SmmOtherSegment must contain a valid selector for the GDT referenced by SmmGdtPtr.

SmmTr Indicates the initial value of TR. SmmTr must contain a valid selector for the GDT referenced by GdtPtr.

SmmCr3 Indicates the initial value of CR3. SmmCr3 must contain the initial value of CR3 that will be used by the STM when setting up the VMCS that represents the SMI handler

SmmStmSetupRip A SmmCs based 64-bit offset which contains the SMM guest entry point which is invoked by the STM when it completes self-initialization but prior to resuming the MLE. If SmmStmSetupRip is zero, this indicates to the STM that there is no SMM specific setup entry point for the thread. If Intel64Mode is clear (0), the upper 32 bits of SmmStmSetupRip must be zero.

SmmStmTeardownRip A SmmCs based 64-bit offset which contains the SMM guest entry point which will be invoked by the STM when it receives a teardown request from the MLE. If SmmStmTeardownRip is zero, this indicates to the STM that there is no

SMM specific teardown entry point for the thread. If `Intel64Mode` is clear (0), the upper 32 bits of `SmmStmTeardownRip` must be zero.

`SmmSmiHandlerRip` A `SmmCs` based 64-bit offset which contains the SMM guest entry point which is invoked by the STM as a result of an SMI. This field is required for all processor threads and must not be zero. If `Intel64Mode` is clear (0), the upper 32 bits of `SmmSmiHandlerRip` must be zero.

`SmmSmiHandlerRsp` A `SmmSs` based 64-bit offset which contains the SMM guest stack pointer at the point it is invoked by the STM as a result of an SMI. This field is required for all processor threads and must not be zero. If `Intel64Mode` is clear (0), the upper 32 bits of `SmmSmiHandlerRsp` must be zero.

`GdtPtr` A 64-bit linear address which contains the starting location of the SMM guest GDT. This GDT structure conforms to the IA-32 Intel® Architecture Software Development Manual. The entire GDT must be contained within the region defined by the SMRR.

`GdtSize` Indicates the size in bytes of the entire GDT. This includes the NULL descriptor at position 0. `GdtSize` bits 2:0 are assumed to be 0, so the actual GDT size is a multiple of 8 bytes.

`RequiredStmSmmRevId` indicates to the STM what SmmRevId is required by the platform. The value of `TXT_PROCESSOR_SMM_DESCRIPTOR.RequiredStmSmmRevId` must be the same as the corresponding value found in the BIOS extended data from the TXT heap. The former is used by the STM, the latter is used by an STM loader to match a compatible STM with the platform in the case where the platform supports dynamic loading of an STM (not currently supported.) See section 3.12 for format details.

`StmProtectionExceptionHandler.SpeRip` (See below)

`StmProtectionExceptionHandler.SpeRsp` (See below)

`StmProtectionExceptionHandler.SpeSs` (See below)

`StmProtectionExceptionHandler.PageViolationException` (See below)

`StmProtectionExceptionHandler.MsrViolationException` (See below)

`StmProtectionExceptionHandler.RegisterViolationException` (See below)

`StmProtectionExceptionHandler.IoViolationException` (See below)

`StmProtectionExceptionHandler.PciViolationException` (See below)

The `STM_PROTECTION_EXCEPTION_HANDLER` provides a mechanism to inform BIOS when the STM blocks SMM access to some hardware resource. The callback is made to `SmmCs:SpeRip` with the stack based at `SpeSs:SpeRsp`. If `Intel64Mode` is clear (0), the upper 32 bits of *`SpeRip`* and *`SpeRsp`* must be zero. Each of the bits (`PageViolationException, MsrViolationException, RegisterViolationException, IoViolationException, PciViolationException`) indicate which of the types of protection exceptions can be handled by BIOS. See section 6.2 for details.

.

BiosHwResourceRequirements is a 64-bit physical pointer to a byte stream of STM_RESOURCE_LIST. BIOS should include all IO ports trapped in this list using the STM_RSC_TRAPPED_IO_DESC entries in the STM_RESOURCE_LIST. BiosHwResourceRequirements and the byte stream it points to must be completely contained within TSEG SMRAM. This memory should be considered static by BIOS and should not be reclaimed for other uses. The STM must copy the contents of the BIOS Resource List into STM-controlled memory that the SMI handler cannot manipulate. See 12 for details on STM_RESOURCE_LIST. All instances of TXT_PROCESSOR_SMM_DESCRIPTOR should have the same value for BiosHwResourceRequirements.

AcpiRsdp is a 64-bit physical pointer to ACPI RSDP table. This field should only be used when STM does non-TXT launch. STM will parse ACPI table during launch time, because STM need information on CPU number or PCI Express Base Address. The STM launcher should guarantee the ACPI table is correct at during launch time. For TXT launch, STM can find related information in TXT heap, and never use this field.

PhysicalAddressBit indicates the max physical address bit supported by this platform. STM may setup EPT page table based on PhysicalAddressBit. Because STM only has limited memory in MSEG, the platform should only report PhysicalAddressBit that cover the physical memory and MMIO. The PhysicalAddressBit can be smaller than the value returned by CPUID(0x80000008).

If the system is going to support STM operation, BIOS must allocate and initialize one instance of the TXT_PROCESSOR_SMM_DESCRIPTOR data structure for each processor thread as part of normal SMM initialization. The structure declares initial register state and GDT location. There are four SMM guest entry points declared. Three are optional, for BIOS notification of STM setup, STM teardown, and BIOS notification of protection exceptions. Only SmmSmiHandlerRip is required, which is used as the SMI handler entry point when servicing SMIs. Except for SmmCs:SpeRip, all BIOS SMI handler code that executes as a result of any of these entry points must eventually return to the STM via the RSM instruction. SmmCs:SpeRip returns to the STM via a special VMCALL interface, described below. Intermediate VMEXITS are possible because of VMCS enabled traps or VMCALLs.

The STM is responsible for using the information from the TXT_PROCESSOR_SMM_DESCRIPTOR structure to set up the initial register state in the SMM guest VMCS prior to invoking the BIOS SMI handler in a VM as a result of STM setup, STM teardown, or as a result of an SMI.

All STM VMRESUME operations that return control back into the SMM guest must maintain the VMCS register state as it was when the VMEXIT occurred except for any changes specifically necessary to support the VMEXIT condition.

With the exception of the SmramToVmcsRestoreRequired and the ReinitializeVmcsRequired bits, the BIOS SMI handler should not change the TXT_PROCESSOR_SMM_DESCRIPTOR structure any time there's an STM configured. Since the STM only reads the data during STM configuration, any changes would not be

recognized. The BIOS is free, however, to reload the GDT or manipulate the contents of the GDT while running within the SMM guest.

There are two supported behaviors related to the preservation of the STM guest VMCS across SMI events. By default, only the RIP and RSP registers are re-initialized on each new SMI; and all of the other state is preserved from the previous SMI. This minimizes the performance overhead related to initializing the VMCS. However, if BIOS has changed its state in a way that is incompatible with the initial RIP and RSP values, it must indicate this to the STM via the `ReinitializeVmcsRequired` bit.

There are a number of requirements on BIOS and on the STM for proper handling of the `ReinitializeVmcsRequired` bit.

1.5 The `ReinitializeVmcsRequired` bit must be set on every thread by the POST BIOS to ensure the STM fully initializes the BIOS VMCS prior to launching the SMM guest.

2.5 The BIOS SMI handler must set this bit prior to RSM on any thread that has evolved to a state incompatible with the initial values of RIP and RSP.

3.5 STM must clear this bit on all threads prior to resuming the interrupted context.

4.5 The STM must set this bit prior to completing teardown, to ensure a subsequent re-launch of the STM operates smoothly.

5.5 The STM must implement an "invalid guest state" handler, such that if entry into the SMM guest ever fails, it will reinitialize the entire guest state in the VMCS based on `TXT_PROCESSOR_SMM_DESCRIPTOR` and try launching the SMM guest again.

6.2 STM protection exceptions

In general, the BIOS SMI handler should not be accessing protected resources. The STM will deny the BIOS SMI handler access to all protected pages, MSRs, and other hardware registers with trust implications per the MLE's protection policy.

The BIOS SMI handler may optionally declare a "protection exception handler" which will be invoked by the STM via a VMRESUME operation if a VMEXIT occurs as a result of a protection violation. The BIOS SMI handler enables exception delivery for each class of protection exception by setting the corresponding bit in the `StmProtectionExceptionHandler` field of the `TXT_PROCESSOR_SMM_DESCRIPTOR` structure. If the BIOS SMI handler does not support protection exception callbacks, it must ensure that the entire `StmProtectionExceptionHandler` structure is cleared to zero.

If the BIOS SMI handler has not declared an STM protection exception handler or enabled the particular exception class, the STM responds to a protection violation by writing `STM_CRASH_PROTECTION_EXCEPTION` to the TXT.ERRORCODE register and asserting TXT.CMD.SYS_RESET.

If, however, the BIOS SMI handler has declared and enabled an STM protection exception handler as described above, the STM will resume SMM guest execution in this handler using `SmmCs:SpeRip` and `SmmSs:SpeRsp` defined in the

StmProtectionExceptionHandler structure. The STM passes the full register state of the offending code to the STM protection exception handler on the stack. The STM must validate that the size of the full register state is not within the STM image. The stack frame differs depending on the state of the Intel64Mode bit.

If Intel64Mode is clear (0), a 32 bit stack frame is used and all stack locations are 32 bit. Unused portions of stack locations due to registers smaller than 32 bits in size are cleared to zero. If Intel64Mode is set (1), a 64 bit stack frame is used. All stack locations are 64 bit. Unused portions of stack locations due to registers smaller than 64 bits are cleared to zero.

ErrorCode indicates the TXT_SMM_PROTECTION_EXCEPTION_TYPE value that caused the exception.

Table 6-1. 32-bit protection exception stack frame

	SpeSs:SpeRsp initial stack location
SS	
ESP	
EFLAGS	
CS	
EIP	
ErrorCode	TXT_SMM_PROTECTION_EXCEPTION_TYPE
VMCS_EXIT_QUALIFICATION	64 bits. See processor reference.
VMX_EXIT_INSTRUCTION_LENGTH	See processor reference.
VMX_EXIT_INSTRUCTION_INFO	See processor reference.
CR0	
CR2	
CR3	
EAX	
EBX	
ECX	
EDX	
EBP	
ESI	
EDI	Location of stack pointer when handler is entered

Table 6-2. Intel 64 protection exception stack frame

	SpeSs:SpeRsp initial stack location
SS	
RSP	
EFLAGS	

CS	
RIP	
ErrorCode	
VMX_EXIT_QUALIFICATION	64 bits. See processor reference.
VMX_EXIT_INSTRUCTION_LENGTH	32 bits (upper 32 bits zero). See processor reference.
VMX_EXIT_INSTRUCTION_INFO	32 bits (upper 32 bits zero). See processor reference.
CR0	
CR2	
CR3	
CR8	
RAX	
RBX	
RBX	
RDX	
RBP	
RSI	
RDI	
R8	
R9	
R10	
R11	
R12	
R13	
R14	
R15	Location of stack pointer when handler is entered

Once the BIOS completes handling the exception, it returns to the STM via the VMCALL interface `StmReturnFromProtectionException`. Based on the inputs to `StmReturnFromProtectionException`, the STM then either writes STM_CRASH_PROTECTION_EXCEPTION to the TXT.ERRORCODE register and asserts TXT.CMD.SYS_RESET, or resumes the SMM guest using the potentially modified register state contained in the stack frame that was passed into the exception handler.

6.3 SMRAM ranges and cache-ability

Substantial SMI performance gains can be realized by allowing the processor to cache SMRAM. When running under an STM, control of TSEG caching is done via the STM. Depending on processor SMRR capabilities, this may involve only controlling access to the SMRR, or may include controlling access to multiple MTRRs.

If IA32_MTRRCAP[11] is set, this indicates SMRR functionality is implemented in the CPU. The implementation of the SMRR in Intel® Core™ i7 and later processors is architecturally defined.

6.3.1 SMRR for Intel® Core™ i7 Processors and later

The SINIT-AC module will verify the SMRR programming and the hardware manages all SMRAM cache-ability without assistance from the STM.

STM must prevent the BIOS SMI handler from writing the IA32_SMRR_PHYSBASE and IA32_SMRR_PHYSMASK MSRs to maintain the integrity of the SMRR feature.

6.4 Monitor trap flag handling in STM

There exists an edge condition with regard to VMENTRY into a guest with the monitor trap flag (MTF) set and a simultaneous SMI. In order to ensure transparent handling of the monitor trap flag (MTF) in the MLE, the STM must observe this condition and re-inject the MTF back into the interrupted guest when it is resumed.

The STM can detect this condition by looking at bit 28 of the exit reason in the VMCS of the interrupted context. If this bit is set, it indicates that a parallel exit to the STM occurred with a pending MTF VMEXIT.

In this case, prior to resuming the interrupted guest, the STM must set the VMENTRY interrupt-information field in the interrupted contexts VMCS to 80000070H (inject "other event" number 0). This will cause an MTF VMEXIT to be pended and delivered immediately after completion of the VMRESUME from the STM.

If the STM doesn't do this re-injection, the guest will execute two instructions, rather than one, before the MTF VMEXIT occurs. This may have undesirable effects on the MLE and must be avoided.

6.5 Performance Monitoring

Performance monitoring requires special treatment by the STM. The STM shall disable the CPUs performance monitoring while in SMM. Upon receiving control due to an SMI, the STM shall save the contents of the IA32_PERF_GLOBAL_CTRL MSR, disable any enabled bits in the IA32_PERF_GLOBAL_CTRL MSR, by clearing the "load IA32_PERF_GLOBAL_CTRL" VM-exit control and use the VM-exit MSR-store area and VM-exit MSR-load area to save and load the MSR.

6.6 Microcode Patch

Since microcode patches must be loaded in VMX Root Mode, the STM must assist the SMI handler guest to accomplish this functionality. For microcode patch, the SMI handler should list the IA32_BIOS_UPDT_TRIG MSR in its required resource list with bit 0 of Attributes set to 1 so that the STM will process an access request to this MSR on behalf of the SMI handler. If the MLE’s policy does not preclude the SMI handler’s

access to the IA32_BIOS_UPDT_TRIG MSR, the STM will execute the microcode patch on behalf of the SMI handler. However, if the MLE's policy does preclude the SMI handler's access, the STM will drop a write request to this MSR.

§

7 STM Teardown

STM teardown is generally the reverse of the startup and is recommended to be as follows:

1. MLE removes all secrets from memory and writes to TXT.CMD.NO-SECRETS
2. MLE caps dynamic PCRs
3. MLE invokes `StopStmVMCALL` on every logical processor. (see section 9.3)
4. MLE executes VMXOFF on every logical processor
5. MLE executes GETSEC[SEXIT] on the BSP

If preparing for S3 sleep, the recommended sequence is slightly different:

1. MLE encrypts all secrets with key K
2. MLE seals key K to dynamic PCRs with monotonic counter
3. MLE removes all secrets from memory, including the key K and writes to TXT.CMD.NO-SECRETS
4. MLE computes hash of all TCB memory
5. MLE seals hash to dynamic PCRs with monotonic counter
6. MLE caps dynamic PCRs
7. MLE invokes `StopStmVMCALL` on every logical processor. (see section 9.3)
8. MLE executes VMXOFF on every logical processor
9. MLE executes GETSEC[SEXIT] on the BSP

When preparing for S3 sleep, step 9 above will cause legacy SMI to be re-enabled. It is important that if this step is delayed or omitted, the MLE must execute GETSEC[SMCTRL] on every logical processor prior to writing the ACPI sleep enable register.

Note that bit 2 (if supported by the CPU) of the IA32_SMM_MONITOR_CTL MSR (Index 9BH) governs SMI unmasking behavior when VMXOFF is executed. The MLE directs the STM how to set this bit in `StartStmVmcall`.

The teardown implementation should take care that MSEG should be returned to a pristine state in which the STM can later be re-launched with the same hash value.

§

8 VMCALL Interfaces Between BIOS SMI Handler and STM

This section defines all VMCALL style interfaces the STM should implement to facilitate interoperability with the BIOS SMI handler.

The VT architecture enables a controlled way to switch from a virtual machine back into the monitor via the VMCALL instruction. The instruction itself takes no parameters so all registers are available for API definition.

Any services needed by the BIOS SMM guest from the STM are supported either implicitly (i.e. page fault to the STM on SMM access to unmapped page or other VMEXIT), or explicitly via a VMCALL interface to the STM. By convention, the STM VMCALL interface will encode a function number in EAX. All other input registers and all output registers (including EAX) are defined by the particular function being invoked.

All parameters and data are passed either in registers, or in caller allocated buffers pointed to by registers. There is no provision for stack based parameter passing.

8.1 Optimizing BIOS Resources

During initialization, the STM obtains a list of resources required by the BIOS SMI handler. To minimize the time it takes to service an SMI, wherever possible these resources should be statically allocated to the BIOS in order to avoid access faults later.

All memory pages contained in the BIOS resource list should be statically mapped into the SMM guest's page table. Other resources declared in the BIOS resource list (e.g. MSR access) should also be statically enabled for BIOS access whenever possible for the same reason.

If the BIOS SMI handler accesses a resource outside of this required resource list that has not been protected by the MLE and is not part of MSEG itself, the STM must grant BIOS access to the resource. If the resource access is denied, then a protection exception is signaled to the SMI handler (if registered), or the system is reset (see section 6.2).

For example: If the BIOS SMI handler accesses a page outside of the BIOS required resource list, a page fault will occur causing a VMEXIT back to the STM. If the faulting page has not been protected by the MLE and is not within the region from the base of MSEG to the top of TSEG, the STM must map the page into the BIOS SMI handler's address space and resume the SMI handler.

Once a resource is added to the BIOS SMI handler's resource pool in this demand-based manner, it need not be removed unless the MLE subsequently requests protection of that resource. It is expected that optimal system performance will be obtained if the BIOS SMI handler is allowed to accumulate resources over time. The

system will "settle" and BIOS will have access to the resources it commonly touches without inducing faults. However, this STM design approach induces the possibility that subsequent protection requests from the MLE could take more time as the STM must synchronously remove the resource from the BIOS space.

Because the STM treats the protection policy as global, it is not necessary to make a protection request on more than one CPU thread. However, the MLE root must rendezvous all CPU threads after making a protection request to ensure all CPU threads recognize the new protection policy.

8.2 Memory resources

8.2.1 Basic rules for SMM guest memory visibility

The STM is responsible for controlling SMM guest access to pages of memory in order to protect the integrity and confidentiality of the MLE. In addition, the STM must present a consistent and functional view of the system to the SMI handler.

The STM must guarantee that all pages within the processor's native address space which are not protected are presented to the SMM guest with a 1:1 guest physical to host physical mapping. This includes normal memory, device memory, I/O APIC, local APIC, PCI express configuration space, etc. In this way, the SMI handler has the same view of memory as in the non-STM model. Page faults while running the SMI handler are permissible as long as they are resolved transparently by the STM. Non-identity mappings are possible if BIOS specifically requests them using the `StmMapAddressRangeVMCALL` (see below).

The STM must guarantee the SMM guest has mappings to all of SMRAM except from the base of MSEG to the top of TSEG. In other words, the STM must ensure normal BIOS-owned SMRAM pages are present when executing the SMM guest.

The STM should determine the location and size of TSEG by consulting SMRR.

The STM must not map any MLE protected pages into the SMM guest's address space. This includes the MLE initial memory footprint and any pages STM has subsequently granted to the MLE for protection.

It is strongly recommended that the STM implementation enables Extended Page Tables (EPT) if EPT is supported by the processor.

If EPT is used by STM, the STM will setup a 1:1 EPT mapping and use EPT to provide protection. In such case, the STM allows direct control of CR3 by the BIOS SMI handler. The BIOS SMI handler will run only with paging on. That is, CR0.PG must be set to 1 by the STM on VMLAUNCH / VMRESUME from the point-of-view of the processor and SMM guest.

Unless EPT under the control of the STM are used (not all CPUs support this feature), the STM must not allow the SMM guest to turn off paging, or directly manipulate the page tables in any other way (e.g. reload CR3). All page table manipulations required by the SMM guest must be done by the STM as a result of either a page fault, or because of a specific request from the SMM guest via a VMCALL back to the STM.

8.2.2 StmMapAddressRange VMCALL

`StmMapAddressRange` enables a SMM guest to create a non-1:1 virtual to physical mapping of an address range into the SMM guest's virtual memory space. Example usages:

`StmMapAddressRange` can be used to pre-load the SMM guest's page tables during `SmmStmSetupRip` to avoid page fault induced latency when servicing an SMI.

It may also be used to map a physical address > 4G into a 32 bit SMM guest's virtual address space or to create other non 1:1 mappings.

It could be used to facilitate execution of SMM code from a fixed virtual address in order to avoid address fix-ups for code images in SMM

It could be used to change the caching parameters of a given address range during SMM.

*** WARNING ***

Incorrect SMM implementation can cause cache incoherency.

*** WARNING ***

The SMM guest is responsible for ensuring cache and TLB coherency. It must flush caches and invalidate TLBs as necessary prior to making this call. If this done incorrectly, cache incoherencies and unpredictable or incorrect system behavior may result.

Since the mechanisms typically used by operating systems for maintaining cache coherency and doing TLB shoot-downs are generally not available to the SMM guest (it may not have direct access to its page tables and cannot easily use interrupts or SIPI to coordinate between processors), it is recommended that multi-threading of SMI handlers be extremely limited. If the BSP handles all of the work required to service an SMI, and the APs restrict their memory accesses to regions of SMRAM that are not going to be modified for either their cache attributes or page mappings, then all cache issues will be contained to the BSP only. This greatly simplifies the problem since the BSP can manage cache flushes and TLB invalidations for itself. This is the recommended implementation.

If the `TXT_PROCESSOR_SMM_DESCRIPTOR.EptEnabled` bit is set by the STM, BIOS can control its own page tables. In this case, the STM implementation may optionally return `ERROR_STM_FUNCTION_NOT_SUPPORTED`.

Input registers:

EAX = `STM_API_MAP_ADDRESS_RANGE`

EBX = low 32 bits of physical address of caller allocated `STM_MAP_ADDRESS_RANGE_DESCRIPTOR` structure.

ECX = high 32 bits of physical address of caller allocated `STM_MAP_ADDRESS_RANGE_DESCRIPTOR` structure. If `Intel64Mode` is clear (0), ECX must be 0.

Note: All fields of STM_MAP_ADDRESS_RANGE_DESCRIPTOR are inputs only. They are not modified by StmMapAddressRange.

Output registers:

CF = 0: No error, EAX set to STM_SUCCESS. The memory range was mapped as requested.

CF = 1: An error occurred, EAX holds relevant error value.

EAX = ERROR_STM_SECURITY_VIOLATION: The requested mapping contains a protected resource.

EAX = ERROR_STM_CACHE_TYPE_NOT_SUPPORTED: The requested cache type could not be satisfied.

EAX = ERROR_STM_PAGE_NOT_FOUND: Page count must not be zero.

EAX = ERROR_STM_FUNCTION_NOT_SUPPORTED: STM supports EPT and has not implemented *StmMapAddressRange()*.

EAX = ERROR_STM_UNSPECIFIED: An unspecified error occurred.

All other registers unmodified.

Related definitions:

```
STM_API_MAP_ADDRESS_RANGE See Appendix B

typedef struct {
  UINT64          PhysicalAddress;
  UINT64          VirtualAddress;
  UINT32          PageCount;
  UINT32          PatCacheType;
} STM_MAP_ADDRESS_RANGE_DESCRIPTOR;

// Allowable values for PatCacheType
#define ST_UC 0x00 // uncached strong ordering
#define WC    0x01 // write combining
#define WT    0x04 // write through
#define WP    0x05 // write protect
#define WB    0x06 // write back
#define UC    0x07 // uncached strong ordering, may
                   // be overridden to WC by an MTRR.
#define FOLLOW_MTRR 0xFFFFFFFF
```

PhysicalAddress Indicates the host physical address requested. This address is assumed to be 4K aligned. Bits 11:0 are ignored and assumed to be 0.

VirtualAddress Indicates the virtual address requested. This address is assumed to be 4K aligned. Bits 11:0 are ignored and assumed to be 0.

PageCount Indicates the number of 4K pages of physical address space requested.

PatCacheType Indicates the requested cache type. The STM must satisfy all cache type requests using the PAT only. MTRR manipulations are not allowed. Allowable values for PatCacheType are ST_UC, WC, WT, WP, WB, UC; or the pseudo type of

FOLLOW_MTRR. FOLLOW_MTRR indicates the PAT should be programmed the same as the governing MTRR.

8.2.3 StmUnmapAddressRange VMCALL

StmUnmapAddressRange enables a SMM guest to remove mappings from its page table.

If TXT_PROCESSOR_SMM_DESCRIPTOR.EptEnabled bit is set by the STM, BIOS can control its own page tables. In this case, the STM implementation may optionally return ERROR_STM_FUNCTION_NOT_SUPPORTED.

Input registers:

EAX = STM_API_UNMAP_ADDRESS_RANGE

EBX: low 32 bits of virtual address of caller allocated STM_UNMAP_ADDRESS_RANGE_DESCRIPTOR structure.

ECX: high 32 bits of virtual address of caller allocated STM_UNMAP_ADDRESS_RANGE_DESCRIPTOR structure. If Intel64Mode is clear (0), ECX must be zero.

Output registers:

CF = 0: No error, EAX set to STM_SUCCESS. The memory range was unmapped as requested.

CF = 1: An error occurred, EAX holds relevant error value.

EAX = ERROR_STM_FUNCTION_NOT_SUPPORTED: STM supports EPT and has not implemented *StmUnmapAddressRange()*.

EAX = ERROR_STM_UNSPECIFIED: An unspecified error occurred.

All other registers unmodified.

Related definitions:

```
STM_API_UNMAP_ADDRESS_RANGE See Appendix B

typedef struct {
  UINT64          VirtualAddress;
  UINT32          Length;
} STM_UNMAP_ADDRESS_RANGE_DESCRIPTOR;
```

VirtualAddress indicates the virtual to be unmapped. The STM will round VirtualAddress down to the nearest page boundary.

Length Indicates the number of bytes to unmap. STM will round (VirtualAddress + Length) up to the nearest 4K page boundary.

All fields of STM_UNMAP_ADDRESS_RANGE_DESCRIPTOR are inputs only. They are not modified by StmUnmapAddressRange.

Note: The STM un-maps pages unconditionally. For example, if the SMM guest requests that the page it is executing in be unmapped, the STM will do it. This is most likely a bug in the SMM guest as this is not a reasonable thing to do. Even so, if the original page which was deleted had a 1:1 physical mapping, it may not appear to the SMM guest that is was removed because the SMM guest will immediately VMEXIT with a page fault and the STM will re-map the page back again.

8.2.4 StmAddressLookup VMCALL

Since the normal OS environment runs with a different set of page tables than the SMM guest, virtual mappings will certainly be different. In order to do a guest virtual to host physical translation of an address from the normal OS code (EIP for example), it is necessary to walk the page tables governing the OS page mappings. Since the SMM guest has no direct access to the page tables, it must ask the STM to do this page table walk. This is supported via the StmAddressLookup VMCALL. All OS page table formats need to be supported, (e.g. PAE, PSE, Intel64, EPT, etc.)

StmAddressLookup takes a CR3 value and a virtual address from the interrupted code as input and returns the corresponding physical address. It also optionally maps the physical address into the SMM guest's virtual address space. This new mapping persists ONLY for the duration of the SMI and if needed in subsequent SMIs it must be remapped. PAT cache types follow the interrupted environment's page table.

If EPT is enabled, OS CR3 only provides guest physical address information, but the SMM guest might also need to know the host physical address. Since SMM does not have direct access rights to EPT (it is protected by the STM), SMM can input InterruptedEptp to let STM help to walk through it, and output the host physical address.

Input registers:

EAX = STM_API_ADDRESS_LOOKUP

EBX: low 32 bits of virtual address of caller allocated STM_ADDRESS_LOOKUP_DESCRIPTOR structure.

ECX: high 32 bits of virtual address of caller allocated STM_ADDRESS_LOOKUP_DESCRIPTOR structure. If Intel64Mode is clear (0), ECX must be zero.

Output registers:

CF = 0: No error, EAX set to STM_SUCCESS.

PhysicalAddress contains the host physical address determined by walking the interrupted SMM guest's page tables.

SmmGuestVirtualAddress contains the SMM guest's virtual mapping of the requested address.

CF = 1: An error occurred, EAX holds relevant error value.

EAX = ERROR_STM_SECURITY_VIOLATION: The requested page was a protected page.

EAX = ERROR_STM_PAGE_NOT_FOUND: The requested virtual address did not exist in the page given page table.

EAX = ERROR_STM_BAD_CR3: The CR3 input was invalid. CR3 values must be from one of the interrupted guest, or from the interrupted guest of another processor.

EAX = ERROR_STM_PHYSICAL_OVER_4G: The resulting physical address is greater than 4G and no virtual address was supplied. The STM could not determine what address within the SMM guest's virtual address space to do the mapping. STM_ADDRESS_LOOKUP_DESCRIPTOR field PhysicalAddress contains the physical address determined by walking the interrupted environment's page tables.

EAX = ERROR_STM_VIRTUAL_SPACE_TOO_SMALL: A specific virtual mapping was requested, but SmmGuestVirtualAddress + Length exceeds 4G and the SMI handler is running in 32 bit mode.

EAX = ERROR_STM_UNSPECIFIED: An unspecified error occurred.

All other registers unmodified.

Related definitions:

```
STM_API_ADDRESS_LOOKUP  See Appendix B

typedef struct {
  UINT64          InterruptedGuestVirtualAddress;
  UINT32          Length;
  UINT32          Reserved;
  UINT64          InterruptedCr3;
  UINT64          InterruptedEptp;
  struct {
     UINT32          MapToSmmGuest          :2; // bitfield
     UINT32          InterruptedCr4Pae      :1; // bitfield
     UINT32          InterruptedCr4Pse      :1; // bitfield
     UINT32          InterruptedIa32eMode   :1; // bitfield
     UINT32          Reserved1              :27; // bitfield
  };
  UINT32          Reserved2;
  UINT64          PhysicalAddress;
  UINT64          SmmGuestVirtualAddress;
} STM_ADDRESS_LOOKUP_DESCRIPTOR;

#define DO_NOT_MAP   0
#define ONE_TO_ONE   1
#define VIRTUAL_ADDRESS_SPECIFIED                    3
```

InterruptedVirtualAddress indicates the virtual address to be looked up. This is an input only and is not modified by the STM.

Length indicates the size in bytes to be mapped. Length is ignored if MapToSmmGuest == DO_NOT_MAP. This is an input only and is not modified by the STM.

InterruptedCr3 contains the address of the page table with which to do the lookup. This is an input only and is not modified by the STM.

`InterruptedEptp` contains the address of the EPT with which to do the lookup. This is an input only and is not modified by the STM.

`MapToSmmGuest` controls SMM guest mapping. It must be set to `DO_NOT_MAP`, `ONE_TO_ONE`, or `VIRTUAL_ADDRESS_SPECIFIED`. This is an input only and is not modified by the STM.

If `MapToSmmGuest = DO_NOT_MAP`, then `PhysicalAddress` is populated with the address determined by walking the interrupted environment's page tables and `SmmGuestVirtualAddress` input is ignored and not modified.

If `MapToSmmGuest = ONE_TO_ONE`, then `PhysicalAddress` is populated with the address determined by walking the interrupted environment's page tables and `SmmGuestVirtualAddress` is ignored on input and is modified to contain the SMM guest's virtual mapping on output, which is the same as `PhysicalAddress`. If `PhysicalAddress` is > 4G, the function fails and returns with CF=1 and EAX = `ERROR_STM_PHYSICAL_OVER_4G`. In this case, `PhysicalAddress` is still populated with the address determined by walking the interrupted environment's page tables.

If `MapToSmmGuest = VIRTUAL_ADDRESS_SPECIFIED`, then `PhysicalAddress` is populated with the address determined by walking the interrupted environment's page tables and mapped to the SMM guest virtual address specified by the `SmmGuestVirtualAddress` input.

`InterruptedCr4Pae` contains the CR4.PAE information. This is an input only and is not modified by the STM.

`InterruptedCr4Pse` contains the CR4.PSE information. This is an input only and is not modified by the STM.

`InterruptedIa32eMode` contains the Ia32eMode information. This is an input only and is not modified by the STM.

`SmmGuestVirtualAddress` is not modified. Calling software must ensure sufficient virtual address space to accommodate the request.

8.2.5 StmReturnFromProtectionException VMCALL

When returning from a protection exception (see section 6.2), the SMM guest can instruct the STM to take one of two paths. It can either request a value be logged to the TXT.ERRORCODE register and subsequently reset the machine (indicating it couldn't resolve the problem), or it can request that the STM resume the SMM guest again with the specified register state.

Unlike other VMCALL interfaces, `StmReturnFromProtectionException` behaves more like a jump or an IRET instruction than a "call". It does not return directly to the caller, but indirectly to a different location specified on the caller's stack (see section 6.2) or not at all.

If the SMM guest STM protection exception handler itself causes a protection exception (e.g. a single nested exception), or more than 100 un-nested exceptions occur within the scope of a single SMI event, the STM must write

STM_CRASH_PROTECTION_EXCEPTION_FAILURE to the TXT.ERRORCODE register and assert TXT.CMD.SYS_RESET. The reason for these restrictions is to simplify the code requirements while still enabling a reasonable debugging capability.

The semantics of the VMCALL are as follows:

Input registers:

EAX = STM_API_RETURN_FROM_PROTECTION_EXCEPTION

EBX = 0: resume SMM guest using register state found on exception stack.

EBX = 1 to 0x0F: EBX contains a BIOS error code which the STM must record in the TXT.ERRORCODE register and subsequently reset the system via TXT.CMD.SYS_RESET. The value of the TXT.ERRORCODE register is calculated as follows:

TXT.ERRORCODE = (EBX & 0x0F) | STM_CRASH_BIOS_PANIC

EBX = 0x10 to 0xFFFFFFFF – reserved, do not use.

Related definitions:

STM_API_RETURN_FROM_PROTECTION_EXCEPTION See Appendix B

§

9 VMCALL Interfaces Between MLE and STM

All MLE-facing VMCALL interfaces require a VMCALL from VMX root mode in the MLE. Parameters follow the convention that an API number is encoded in EAX, and ECX:EBX are used to pass a physical address of a parameter data structure. In all cases, the parameter structure must be 4K aligned, and must not cross a 4K boundary.

Before making any of these calls, the MLE root should verify that the IA32_SMM_MONITOR_CTL.VALID (bit 0) is set. If a VMCALL is attempted from VMX root mode (CPL=0) when IA32_SMM_MONITOR_CTL.VALID is clear this indicates there is no STM present in MSEG and the VMCALL will fail. The STM configuration lifecycle is as follows:

1. SENTER->SINIT->MLE: MLE begins execution with SMIs disabled (masked).
2. MLE root invokes `InitializeProtectionVMCALL()` to prepare the STM for setup of the initial protection profile. This is done on a single CPU and has global effect.
3. (Optional) MLE may invoke `GetBiosResourceVMCALL()` to inspect the list of resources that the BIOS has access to. This can be done on a single CPU.
4. MLE root invokes `ProtectResourceVMCALL()` to define the initial protection profile. The protection profile is global across all CPUs.
5. MLE root invokes `StartStmVMCALL()` to enable the STM to begin receiving SMI events. This must be done on every logical CPU.
6. MLE root may invoke `ProtectResourceVMCALL()` or `UnProtectResourceVMCALL()` during runtime as many times as necessary.
7. MLE root invokes `StopStmVMCALL()` to disable the STM. This must be done on every logical CPU. SMI is again masked following `StopStmVMCALL()`.

9.1 InitializeProtectionVMCALL()

`InitializeProtectionVMCALL()` prepares the STM for setup of the initial protection profile which is subsequently communicated via one or more invocations of `ProtectResourceVMCALL()`, prior to invoking `StartStmVMCALL()`. It is only necessary to invoke `InitializeProtectionVMCALL()` on one processor thread. `InitializeProtectionVMCALL()` does not alter whether SMIs are masked or unmasked. The STM should return back to the MLE with "Blocking by SMI" set to 1 in the GUEST_INTERRUPTIBILITY field for the VMCS the STM created for the MLE guest.

Input registers:

EAX = STM_API_INITIALIZE_PROTECTION

Output registers:

CF = 0: No error, EAX set to STM_SUCCESS, EBX bits set to indicate STM capabilities as defined below. The STM has set up an empty protection profile, except for the resources that it sets up to protect itself. The STM must not allow the SMI handler to map any pages from the MSEG Base to the top of TSEG. The STM must also not allow SMI handler access to those MSRs which the STM requires for its own protection.

CF = 1: An error occurred, EAX holds relevant error value.

EAX = ERROR_STM_ALREADY_STARTED: The STM is already configured and active. The STM remains active and guarding the previously enabled resource list.

EAX = ERROR_STM_UNPROTECTABLE: The STM determines that based on the platform configuration, the STM is unable to protect itself. For example, the BIOS required resource list contains memory pages in MSEG.

EAX = ERROR_STM_UNSPECIFIED: An unspecified error occurred.

All other registers unmodified.

Related definitions:

The return value of EBX when CF==0 is defined as follows:

31	30…4	3	2	1	0
0		MSR	BGM	BGI	0

BGI: If set (1), indicates byte granular MMIO range support. If clear (0), indicates page granular MMIO range support. See 12.

BGM: If set (1), indicates byte granular memory range support. If clear (0), indicates page granular memory range support. See 12.

MSR: If set (1), indicates bit granular MSR resource support. If clear (0), indicates whole MSR granular resource support only. See 12.

STM_API_INITIALIZE_PROTECTION	See Appendix B
ERROR_STM_ALREADY_STARTED	See Appendix C
ERROR_STM_WITHOUT_SMX_UNSUPPORTED	See Appendix C

9.2 StartStmVMCALL()

StartStmVmcall() is used to configure an STM that is present in MSEG. SMIs should remain disabled from the invocation of GETSEC[SENTER] until they are re-enabled by StartStmVMCALL(). When StartStmVMCALL() returns, SMI is enabled and the STM has been started and is active. Prior to invoking StartStmVMCALL(), the MLE root should first invoke InitializeProtectionVMCALL() followed by as many iterations of

`ProtectResourceVMCALL()` as necessary to establish the initial protection profile. StartStmVmcall() must be invoked on all processor threads.

It is necessary to process the initial protection profile for only one of the CPUs, since there is only one profile. All return values must be the same across all CPUs.

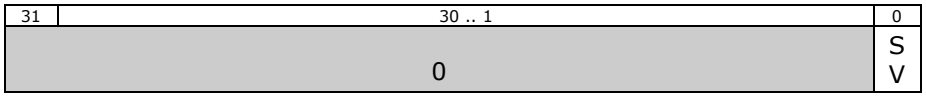
The STM enables SMIs by returning back to the MLE with "Blocking by SMI" set to 0 in the GUEST_INTERRUPTIBILITY field for the VMCS the STM created for the MLE guest.

Input registers:

EAX = `STM_API_START`

EDX = STM configuration options. These provide the MLE with the ability to pass configuration parameters to the STM.

The STM configuration options are defined as follows:



SMI VMXOFF (SV): This configuration option bit directs the STM on how to set bit 2 of the IA32_SMM_MONITOR_CTL MSR (Index 9BH.) Bit 2 of the IA32_SMM_MONITOR_CTL MSR prevents SMI unblocking following the VMXOFF instruction. Consult the Intel Software Developer Manuals for additional guidance on this bit.

If supported by the CPU, the STM will set bit 2 of the IA32_SMM_MONITOR_CTL MSR to the value of the SMI VMXOFF configuration option bit.

Output registers:

CF = 0: No error, EAX set to `STM_SUCCESS`. The STM has been configured and is now active and the guarding all requested resources.

CF = 1: An error occurred, EAX holds relevant error value.

EAX = `ERROR_STM_ALREADY_STARTED`: The STM is already configured and active. STM remains active and guarding previously enabled resource list.

EAX = `ERROR_STM_WITHOUT_SMX_UNSUPPORTED`: The `StartStmVMCALL()` was invoked from VMX root mode, but outside of SMX. This error code indicates the STM or platform does not support the STM outside of SMX. The SMI handler remains active and operates in legacy mode. See Appendix C

EAX = `ERROR_STM_UNSUPPORTED_MSR_BIT`: The CPU doesn't support the MSR bit. The STM is not active.

EAX = `ERROR_STM_UNSPECIFIED`: An unspecified error occurred.

All other registers unmodified.

Related definitions:

STM_API_START	See Appendix B
ERROR_STM_ALREADY_STARTED	See Appendix C
ERROR_STM_WITHOUT_SMX_UNSUPPORTED	See Appendix C

9.3 StopStmVMCALL()

The StopStmVMCALL() is invoked by the MLE to teardown an active STM. This is normally done as part of a full teardown of the SMX environment when the system is being shut down. At the time the call is invoked, SMI is enabled and the STM is active. When the call returns, the STM has been stopped and all STM context is discarded and SMI is disabled.

Upon completion of StopStmVMCALL(), SMIs are masked. The MLE must re-enable SMIs either by executing GETSEC[SEXIT] or GETSEC[SMCTRL]. Failure to do this can result in system instability.

Upon completion of StopStmVMCALL(), all resource protections are removed. Any future launch of the STM must build a new protected resource list.

This must be invoked on all CPUs. The STM invokes the SmmStmTeardownRip (if specified) during teardown.

If the MLE subsequently issues a VMXOFF, SMIs may be unmasked. If the processor supports it, MSR 9B (bit 2) controls whether SMIs are unmasked after VMXOFF. On processors that don't support this bit, SMIs will be unconditionally unmasked following the VMXOFF instruction. The Intel Software Development Manuals describe this in additional detail.

Input registers:

EAX = STM_API_STOP

Output registers:

CF = 0: No error, EAX set to STM_SUCCESS. The STM has been stopped and is no longer processing SMI events. SMI is blocked.

CF = 1: An error occurred, EAX holds relevant error value.

EAX = ERROR_STM_STOPPED: The STM was not active.

EAX = ERROR_STM_UNSPECIFIED: An unspecified error occurred.

All other registers unmodified.

Related definitions:

STM_API_STOP	See Appendix B

9.4 ProtectResourceVMCALL()

The `ProtectResourceVMCALL()` is invoked by the MLE root to request protection of specific resources. The request is defined by a `STM_RESOURCE_LIST`, which may contain more than one resource descriptor. Each resource descriptor is processed separately by the STM. Whether or not protection for any specific resource is granted is returned by the STM via the `ReturnStatus` bit in the associated `STM_RSC_DESC_HEADER`.

A resource protection request will be denied by the STM if the resource in question intersects a resource on the BIOS required resource list.

The MLE protection policy is applied to all CPU threads, therefore it is only necessary to invoke `ProtectResourceVMCALL()` on one CPU thread. However, the MLE must rendezvous all CPUs after the `ProtectResourceVMCALL()` returns before the change to requested protections can be considered effective. Failure to do this could result in inadvertently exposing secrets to the BIOS SMI handler. The MLE must wait for this rendezvous before treating a resource can be considered protected. `ProtectResourceVMCALL()` does not alter whether SMI is masked or enabled.

Input registers:

EAX = `STM_API_PROTECT_RESOURCE`

EBX = low 32 bits of physical address of caller allocated `STM_RESOURCE_LIST`. Bits 11:0 are ignored and assumed to be zero, making the buffer 4K aligned.

ECX = high 32 bits of physical address of caller allocated `STM_RESOURCE_LIST`.

Note: all fields of `STM_RESOURCE_LIST` are inputs only, except for the `ReturnStatus` bit. On input, the `ReturnStatus` bit must be clear. On return, the `ReturnStatus` bit is set for each resource request granted, and clear for each resource request denied. There are no other fields modified by `ProtectResourceVMCALL()`. The `STM_RESOURCE_LIST` must be contained entirely within a single 4K page.

Output registers:

CF = 0: No error, EAX set to `STM_SUCCESS`. The STM has successfully merged the entire protection request into the active protection profile. There is therefore no need to check the `ReturnStatus` bits in the `STM_RESOURCE_LIST`.

CF = 1: An error occurred, EAX holds relevant error value.

EAX = `ERROR_STM_UNPROTECTABLE_RESOURCE`: At least one of the requested resource protections intersects a BIOS required resource. Therefore, the caller must walk through the `STM_RESOURCE_LIST` to determine which of the requested resources was not granted protection. The entire list must be traversed since there may be multiple failures.

EAX = `ERROR_STM_MALFORMED_RESOURCE_LIST`: The resource list could not be parsed correctly, or did not terminate before crossing a 4K page boundary. The caller must walk through the `STM_RESOURCE_LIST` to determine which of the requested resources was not granted protection. The entire list must be traversed since there may be multiple failures.

EAX = `ERROR_STM_OUT_OF_RESOURCES`: The STM has encountered an internal error and cannot complete the request.

EAX = `ERROR_STM_UNSPECIFIED`: An unspecified error occurred.

All other registers unmodified.

Related definitions:

STM_API_PROTECT_RESOURCE	See Appendix B
STM_RESOURCE_LIST	See 12
ERROR_STM_UNPROTECTABLE_RESOURCE	See Appendix C
ERROR_STM_MALFORMED_RESOURCE_LIST	See Appendix C
ERROR_STM_WITHOUT_SMX_UNSUPPORTED	See Appendix C

9.5 UnProtectResourceVMCALL()

The UnProtectResourceVMCALL() is invoked by the MLE root to request that the STM allow the SMI handler access to the specified resources.

The MLE protection profile is applied to all CPU threads, therefore it is only necessary to invoke UnProtectResourceVMCALL() on one CPU thread. However, the MLE root must rendezvous all CPUs after UnProtectResourceVMCALL() returns before the change to protections can be considered valid.

The STM does not ensure there was previous protection of the specified resources, only that these resources are not protected when UnProtectResourceVMCALL() returns.

Input registers:

EAX = STM_API_UNPROTECT_RESOURCE

EBX = low 32 bits of physical address of caller allocated STM_RESOURCE_LIST. Bits 11:0 are ignored and assumed to be zero, making the buffer 4K aligned.

ECX = high 32 bits of physical address of caller allocated STM_RESOURCE_LIST.

Note: all fields of STM_RESOURCE_LIST are inputs only, except for the ReturnStatus bit. On input, the ReturnStatus bit must be clear. On return, the ReturnStatus bit is set for each resource processed. For a properly formed STM_RESOURCE_LIST, this should be all resources listed. There are no other fields modified by UnProtectResourceVMCALL(). The STM_RESOURCE_LIST must be contained entirely within a single 4K page.

Output registers:

CF = 0: No error, EAX set to STM_SUCCESS. The requested resources are not being guarded by the STM.

CF = 1: An error occurred, EAX holds relevant error value.

EAX = ERROR_STM_MALFORMED_RESOURCE_LIST: The resource list could not be parsed correctly, or did not terminate before crossing a 4K page boundary. The caller must walk through the STM_RESOURCE_LIST to determine which of the requested resources were not able to be unprotected. The entire list must be traversed since there may be multiple failures.

EAX = ERROR_STM_UNSPECIFIED: An unspecified error occurred.

All other registers unmodified.

Related definitions:

```
STM_API_UNPROTECT_RESOURCE        See Appendix B
STM_RESOURCE_LIST                 See 12
ERROR_STM_MALFORMED_RESOURCE_LIST See Appendix C
ERROR_STM_WITHOUT_SMX_UNSUPPORTED    See Appendix C
```

9.6 GetBiosResourcesVMCALL()

The GetBiosResourcesVMCALL() is invoked by the MLE root to request the list of BIOS required resources from the STM.

The BIOS required resources are the same on all CPU threads, therefore it is only necessary to invoke GetBiosResourcesVMCALL() on one CPU thread. Furthermore, the BIOS required resource list is static, so the MLE can assume that data returned by GetBiosResourcesVMCALL() will not dynamically change. This enables the MLE to evaluate whether its protection profile is viable given the BIOS resource demands.

The BIOS STM_RESOURCE_LIST is returned in a caller allocated 4K aligned buffer. The buffer is 4K in size and 4K aligned (the low 12 bits of the buffer address must be zero). If the BIOS resource list is larger than 4K in size, GetBiosResourcesVMCALL() should be made iteratively. The return value of EDX will be incremented to indicate the next page index. When EDX is returned cleared to 0, this indicates the entire list has been retrieved.

If this request is made at a time when SMIs are enabled, the MLE should put the buffer in a protected memory area inaccessible to the SMI handler.

See description of TXT_PROCESSOR_SMM_DESCRIPTOR.BiosHwResourceRequirementsPtr in section 6.1 for details regarding the BIOS STM_RESOURCE_LIST.

Input registers:

EAX = STM_API_GET_BIOS_RESOURCES

EBX = low 32 bits of physical address of caller allocated destination buffer. Bits 11:0 are ignored and assumed to be zero, making the buffer 4K aligned.

ECX = high 32 bits of physical address of caller allocated destination buffer.

EDX = indicates which page of the BIOS resource list to copy into the destination buffer. The first page is indicated by 0, the second page by 1, etc.

Output registers:

CF = 0: No error, EAX set to STM_SUCCESS. The destination buffer contains the BIOS required resources. If the page retrieved is the last page, EDX will be cleared to 0. If there are more pages to retrieve, EDX is incremented to the next page index. Calling software should iterate on GetBiosResourcesVMCALL() until EDX is returned cleared to 0.

CF = 1: An error occurred, EAX holds relevant error value.

EAX = ERROR_STM_PAGE_NOT_FOUND: The page index supplied in EDX input was out of range.

EAX = ERROR_STM_UNSPECIFIED: An unspecified error occurred.

EDX = Page index of next page to read. A return of EDX=0 signifies that the entire list has been read. (Note: EDX is both an input and an output register.)

All other registers unmodified.

Related definitions:

```
STM_API_GET_BIOS_RESOURCES        See Appendix B
STM_RESOURCE_LIST                 See 12
ERROR_STM_PAGE_NOT_FOUND          See Appendix C
ERROR_STM_WITHOUT_SMX_UNSUPPORTED    See Appendix C
```

9.7 ManageVmcsDatabaseVMCALL()

See section 10.3 for a description of protected domains.

The ManageVmcsDatabaseVMCALL() is invoked by the MLE root to add or remove an MLE guest (including the MLE root) from the list of protected domains. Input registers:

EAX = STM_API_MANAGE_VMCS_DATABASE

EBX = low 32 bits of physical address of caller allocated STM_VMCS_DATABASE_REQUEST. Bits 11:0 are ignored and assumed to be zero, making the buffer 4K aligned.

ECX = high 32 bits of physical address of caller allocated STM_VMCS_DATABASE_REQUEST.

Note: all fields of STM_VMCS_DATABASE_REQUEST are inputs only. They are not modified by ManageVmcsDatabaseVMCALL().

Output registers:

CF = 0: No error, EAX set to STM_SUCCESS.

CF = 1: An error occurred, EAX holds relevant error value.

EAX = ERROR_STM_INVALID_VMCS – indicates a request to remove a VMCS from the database was made, but the referenced VMCS was not found in the database.

EAX = ERROR_STM_VMCS_PRESENT – indicates a request to add a VMCS to the database was made, but the referenced VMCS was already present in the database.

EAX = ERROR_INVALID_PARAMETER – Indicates non-zero reserved field.

EAX = ERROR_STM_UNSPECIFIED: An unspecified error occurred.

All other registers unmodified.

Related definitions:

```
STM_API_MANAGE_VMCS_DATABASE            // See Appendix B
```

The database entries contain information that cannot be derived from normal VMCS context, or register state at entry into the STM. See **Error! Reference source not found.** for full description of domain types.

```
typedef struct {
    UINT64   VmcsPhysPointer;     // bits 11:0 are reserved and must be 0
     struct {
        UINT32                    DomainType:4;         // bitfield
        UINT32                    XStatePolicy:2;       // bitfield
        UINT32                    DegradationPolicy:4;  // bitfield
        UINT32                    Reserved1:22;   // bitfield - Must be 0
     };
    UINT32   AddOrRemove;
} STM_VMCS_DATABASE_REQUEST;
```

`VmcsPhysPointer` – This field contains the VMCS pointer. As all VMCSes must be 4k-aligned bits 11-0 must be 0.

`DomainType` – These bits indicate the state save area generation and propagation rules. See section 10 and **Error! Reference source not found.** for details.

`XStatePolicy` – This bit field indicates the access policy for extended register state. If DomainType is `UNPROTECTED`, then the `XStatePolicy` field is ignored and assumed to be `XSTATE_READWRITE`. For all other domain types, it must have one of the following values:

`XSTATE_READWRITE`: The interrupted context's extended state is fully visible and modifiable by the BIOS SMI handler. See section 10.

`XSTATE_READONLY`: Indicates the STM must save the extended state prior to invoking the BIOS SMI handler (e.g. perform XSAVE with all bits in EAX:EDX set to save interrupted context). The extended state is not scrubbed. The STM must restore the saved extended state prior to resuming the interrupted context. See section 10.

`XSTATE_SCRUB`: Indicates the STM must save and scrub the extended state prior to invoking the BIOS SMI handler (e.g. perform XSAVE with all bits in EAX:EDX set to save interrupted context, then perform XRSTOR using a zeroed XSAVE buffer to clear all associated registers prior to entry into BIOS SMM handler); and then restore the saved extended state prior to resuming the interrupted context. See section 10.

`DegradationPolicy` – This bit field indicates the minimum protection domain type permissible by the MLE if domain type degradation occurs. Valid values are the same as DomainType. See section 10.

`AddOrRemove` – A value of 1 indicates that the referenced VMCS is to be added to the protected VMCS database. A value of 0 indicates that the referenced VMCS is to be removed from the protected VMCS database. All other values are reserved. Note: to modify an entry, it must first be removed and then added again.

Related definitions:

```
// Values for DomainType
#define  UNPROTECTED                0x00;
```

```
#define   INTEGRITY_PROT_OUT_IN        0x04;
#define   FULLY_PROT_OUT_IN            0x0C;
#define   FULLY_PROT                   0x0F;

// Values for XStatePolicy
#define   XSTATE_READWRITE             0x00;
#define   XSTATE_READONLY              0x01;
#define   XSTATE_SCRUB                 0x03;
```

9.8 ManageEventLogVMCALL()

The `ManageEventLogVMCALL()` is invoked by the MLE root to control the logging feature. It consists of several sub-functions to facilitate establishment of the log itself, configuring what events will be logged, and functions to start, stop, and clear the log.

The log itself is a circular buffer that is allocated from MLE memory. The log buffer's location is communicated to the STM via an array of the physical page addresses that the MLE allocated. All pages of the log must be pinned in memory any time the log is active. The format of the log is described in Appendix E.

Input registers:

EAX = `STM_API_MANAGE_EVENT_LOG`

EBX = low 32 bits of physical address of caller allocated `STM_EVENT_LOG_MANAGEMENT_REQUEST`. Bits 11:0 are ignored and assumed to be zero, making the buffer 4K aligned.

ECX = high 32 bits of physical address of caller allocated `STM_EVENT_LOG_MANAGEMENT_REQUEST`.

Output registers:

CF = 0: No error, EAX set to `STM_SUCCESS`.

CF = 1: An error occurred, EAX holds relevant error value. See subfunction descriptions below for details.

All other registers unmodified.

Related definitions:

```
STM_API_MANAGE_EVENT_LOG   // See Appendix B

#define   LOG_NEW_LOG                  1;
#define   LOG_CONFIGURE_LOG            2;
#define   LOG_START_LOG                3;
#define   LOG_STOP_LOG                 4;
#define   LOG_CLEAR_LOG                5;
#define   LOG_DELETE_LOG               6;

typedef enum {
    EVT_LOG_STARTED,
    EVT_LOG_STOPPED,
    EVT_LOG_INVALID_PARAMETER_DETECTED,
    EVT_PROTECTION_EXCEPTION,
    EVT_HANDLED_PROTECTION_EXCEPTION,
```

```
    EVT_BIOS_ACCESS_TO_UNCLAIMED_RESOURCE,
    EVT_MLE_RESOURCE_PROTECTION_GRANTED,
    EVT_MLE_RESOURCE_PROTECTION_DENIED,
    EVT_MLE_RESOURCE_UNPROTECT,
    EVT_MLE_RESOURCE_UNPROTECT_ERROR,
    EVT_MLE_DOMAIN_TYPE_DEGRADED
} EVENT_TYPE;
```

The parameter STM_EVENT_LOG_MANAGEMENT_REQUEST parameter structure is defined as follows:

```
typedef struct {
    UINT32          SubFunctionIndex;   // One of LOG_* above
    union {
        struct {
            UINT32   PageCount;
            UINT64   Pages[];           // number of elements is PageCount
        } LogBuffer;
        UINT32      EventEnableBitmap; // bitmap of EVENT_TYPE
    } ;
} STM_EVENT_LOG_MANAGEMENT_REQUEST;
```

SubFunctionIndex – The ManageEventLogVMCALL supports several subfunctions. The SubFunctionIndex is used to differentiate between them.

If SubFunctionIndex == NEW_LOG, the STM will associate a set of MLE allocated pages with the log. The value of LogBuffer.PageCount indicates the total number of pages declared in LogBuffer.Pages[], which contains an ordered list of physical addresses of pages that represent the log buffer. Each element of u.LogBuffer.Pages[] is assumed to be 4K aligned (bits 11:0 are ignored and assumed to be 0). The entire STM_EVENT_LOG_MANAGEMENT_REQUEST must fit within a single 4K page. Each STM_LOG_ENTRY is stored at 256 byte offsets on the page. The STM will truncate any STM_LOG_ENTRY that is greater than 256 bytes. This subfunction will fail if the log has already been allocated.

Error conditions:

EAX = ERROR_STM_PAGE_NOT_FOUND: STM determined that one of the pages of the event log is not within MLE owned memory.

EAX = ERROR_STM_LOG_ALLOCATED: The event log has already been allocated.

EAX = ERROR_STM_INVALID_PAGECOUNT: LogBufferPageCount is zero or the entire STM_EVENT_LOG_MANAGEMENT_REQUEST wouldn’t fit on a single 4K page.

EAX = ERROR_STM_UNSPECIFIED: An unspecified error occurred.

If SubFunctionIndex == CONFIGURE_LOG, the value of EventEnables is a bitmap which indicates the types of events to log. A set bit indicates that event will be logged. A cleared bit indicates that event will not be logged. All bits not explicitly defined are reserved and must be 0. This subfunction will fail if called with the event log in the running state.

The bit numbers in `u.EventEnables` correspond to the `EVENT_TYPE`. For example, to enable logging of handled protection exceptions, the caller would define `u.EventEnables |= (1 << EVT_HANDLED_PROTECTION_EXCEPTION)`.

Error conditions:

EAX = `ERROR_STM_RESERVED_BIT_SET:` A reserved bit was set in the bitmap.

EAX = `ERROR_STM_LOG_NOT_ALLOCATED:` The event log has not been allocated

EAX = `ERROR_STM_LOG_NOT_STOPPED:` The event log is already started

EAX = `ERROR_STM_UNSPECIFIED`: An unspecified error occurred.

If `SubFunctionIndex == START_LOG`, the logging function is started. If this is the first time the logging function is started since the log was established or cleared, new log entries will be placed at the beginning of the log. Otherwise, new log entries continue are appended to the log at the next available log entry beyond the most recent log entry. This subfunction will fail if no log buffer has been declared. This subfunction will fail if no events are enabled for logging. This subfunction will fail if called with the event log in the running state.

Error conditions:

EAX = `ERROR_STM_NO_EVENTS_ENABLED:` There are no events enabled to log

EAX = `ERROR_STM_LOG_NOT_ALLOCATED:` The event log hasn't been allocated.

EAX = `ERROR_STM_LOG_NOT_STOPPED:` The event log is already running.

EAX = `ERROR_STM_UNSPECIFIED`: An unspecified error occurred.

If `SubFunctionIndex == STOP_LOG`, the logging function is stopped. This subfunction will fail if called with the event log in the stopped state, or if the log has not been allocated.

EAX = `ERROR_STM_LOG_NOT_STARTED:` The event log wasn't running.

EAX = `ERROR_STM_LOG_NOT_ALLOCATED:` The event log wasn't allocated.

EAX = `ERROR_STM_UNSPECIFIED`: An unspecified error occurred.

If `SubFunctionIndex == CLEAR_LOG`, all log entries will be erased and the next log entry will be placed at the beginning of the log. This subfunction will fail if no log buffer has been declared. This subfunction will fail if called with the event log in the running state, or if the log has not been allocated.

EAX = `ERROR_STM_LOG_NOT_STOPPED:` The event log is currently running.

EAX = `ERROR_STM_LOG_NOT_ALLOCATED:` The event log hasn't been allocated.

EAX = `ERROR_STM_UNSPECIFIED`: An unspecified error occurred.

If `SubFunctionIndex == DELETE_LOG`, any previously declared log is abandoned and the association of MLE pages with an event log is dissolved. This subfunction will fail if called with the event log in the running state.

EAX = `ERROR_STM_LOG_NOT_STOPPED:` The event log is currently running.

EAX = `ERROR_STM_UNSPECIFIED`: An unspecified error occurred.

§

10 SMRAM context handling

10.1 SMRAM state save map generation

When an SMI occurs in the absence of an STM (e.g. opt out), the processor itself locates the SMRAM state save area via the SMBASE value and saves register contents to it prior to beginning execution of the SMI handler.

When an STM is running, the CPU performs a VMEXIT to transfer execution into the STM. It is the STM's responsibility to construct the SMRAM state save for each processor. The STM constructs the state at the expected location (SMBASE+8000H) using values gleaned from the VMCS or persisting in the registers themselves when the STM begins execution.

10.1.1 STM generated SMRAM state save map

The STM generated SMRAM state save area differs slightly from the CPU generated SMRAM state save map for Intel 64 Architecture. The STM generated SMRAM state save area is shown below. The differences between the STM generated SMRAM state save map and the Intel 64 architecture state save map are highlighted in yellow.

Table 10-1. STM generated SMRAM state save

Offset (Added to SMBASE + 8000H)	Register	Writable?
7FF8H	CR0	No
7FF0H	CR3	No
7FE8H	RFLAGS	Yes
7FE0H	IA32_EFER	No
7FD8H	RIP	Yes
7FD0H	DR6	No
7FC8H	DR7	No
7FC4H	TR SEL[1]	No
7FC0H	LDTR SEL[1]	No
7FBCH	GS SEL[1]	No
7FB8H	FS SEL[1]	No
7FB4H	DS SEL[1]	No
7FB0H	SS SEL[1]	No
7FACH	CS SEL[1]	No
7FA8H	ES SEL[1]	No

Offset (Added to SMBASE + 8000H)	Register	Writable?
7FA4H	IO_MISC	No
7F9CH	IO_MEM_ADDR	No
7F94H	RDI	Yes
7F8CH	RSI	Yes
7F84H	RBP	Yes
7F7CH	RSP	Yes
7F74H	RBX	Yes
7F6CH	RDX	Yes
7F64H	RCX	Yes
7F5CH	RAX	Yes
7F54H	R8	Yes
7F4CH	R9	Yes
7F44H	R10	Yes
7F3CH	R11	Yes
7F34H	R12	Yes
7F2CH	R13	Yes
7F24H	R14	Yes
7F1CH	R15	Yes
7F1BH-7F04H	Reserved	No
7F02H	Auto HALT Restart Field (Word)	Yes
7F00H	I/O Instruction Restart Field (Word)	Yes
7EFCH	SMM Revision Identifier Field (Doubleword)	No
7EF8H	SMBASE Field (Doubleword)	No
7EF7H-7EE4H	Reserved	No
7EE0H	Setting of “enabled EPT” VM-execution control	No
7ED8H	Value of EPTP VM-execution control field	No
7ED7H-7EA0H	Reserved	No
7E9CH	LDT Base (lower 32 bits)	No
7E98H	Reserved	No
7E94H	IDT Base (lower 32 bits)	No
7E90H	Reserved	No
7E8CH	GDT Base (lower 32 bits)	No
7E8BH-7E44H	Reserved	No
7E40H	CR4	No

Offset (Added to SMBASE + 8000H)	Register	Writable?
7E3FH-7DF0H	Reserved	No
7DE8H	IO_EIP	Yes[2]
7DE7H-7DDCH	Reserved	No
7DD8H	IDT Base (Upper 32 bits)	No
7DD4H	LDT Base (Upper 32 bits)	No
7DD0H	GDT Base (upper 32 bits)	No
7DCFH-7C00H	Reserved	No

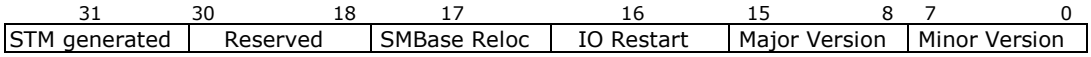
NOTES:

1. The two most significant bytes are reserved.
2. IO_EIP only writeable when an unprotected context is interrupted.

10.1.2 SMM_REV_ID

The STM generated SMM_REV_ID field of the SMRAM state save has the following format:

Figure 10-1. STM generated SMM_REV_ID



Bit 31 of the SMM_REV_ID indicates the state save region is generated by an STM rather than hardware.

STM generated SMRAM state save version is divided into major and minor components. Within the scope of any given major version, subsequent minor versions are backward compatible. No backward compatibility guarantee is given if the major version number is different.

The SMRAM state save version described in section 10.1.1 must have a value of 80010100H. This indicates an STM generated SMM state save area with 1.0 version number, IO Restart supported, and SMBase Relocation unsupported.

10.2 Asynchronous and synchronous SMI

There are generally two classes of SMI: Asynchronous and synchronous.

Asynchronous SMIs occur as a result of a system event that is not directly related (if not completely orthogonal) to the currently executing code stream, e.g. a thermal trip point can generate an SMI. Handling an asynchronous SMI does not require BIOS SMI handler visibility into the register state of the interrupted context.

Synchronous SMIs occur directly as a result of some action by software. This occurs when BIOS has enabled a particular SMI trigger event in the system's chipset. When a synchronous SMI occurs, the CPU is interrupted on the next instruction boundary relative to the instruction that caused the SMI. The SMI handler can then inspect and

in some cases modify the interrupted code's CPU register context via the SMI state save area. Synchronous SMI is used for patching silicon issues, emulating legacy hardware with USB peripherals, and implementing custom BIOS interfaces. Without this capability, it is likely that many systems would not run properly.

Synchronous SMIs occur on IO instructions to IO ports, e.g. IN, and OUT and their variants. There are two variants of synchronous SMI trap usage. These are "normal synchronous SMI traps", and "synchronous SMI APIs". This distinction is relevant to the STM's behavior as it relates to protected domains.

10.2.1 Normal Synchronous SMI Traps

In most cases, the SMI handler must follow the semantics of the IO instruction that caused the trap as described in the Intel Software Developer's Manual. This is usually the case when the SMI handler is patching or emulating hardware. The SMI handler should not assume any register or memory state that is not architecturally defined by the IO instruction in question. Similarly, it should not modify any register or memory state in the interrupted MLE that is not architecturally defined by the IO instruction in question.

In other words, from the perspective of the interrupted context, nothing unexpected should be observed. It should be as if the IO was serviced by hardware and there was no SMI trap.

10.2.2 Synchronous SMI APIs

In some cases an SMI IO trap is, in practice, an interface trigger between host software and the BIOS SMI handler rather than an emulation of IO hardware. A common port used for this purpose is port B2H. In these cases, register state beyond the IO instruction's definition can be used or modified, as can memory. What state is required by the SMI handler for this sort of trap is specific to the API that is implemented over the IO port in question.

If an SMI-based hardware workaround requires access to state beyond the architecturally defined state of the IO instruction in question, it is actually a synchronous SMI API. This could be the case, for example, if the SMI handler is used to patch driver software.

10.3 Domain protections

A goal of TXT is to make it possible to create software domains that have both confidentiality and integrity properties (as used below, from the perspective of the BIOS SMI handler) and to attest to the contents of such a domain. This leads to three possible domain types:

- "Unprotected domain" – A domain with neither integrity nor confidentiality properties.
- "Integrity protected domain" – A domain with the integrity property, but lacking confidentiality.

- "Fully protected domain" - A domain with both integrity and confidentiality properties.

The fourth potential combination, a domain with confidentiality but without integrity, is not possible since integrity is required in order to maintain confidentiality.

If a protected domain is interrupted by SMI and the SMI handler is given unfettered access to the interrupted context's register state, there is a possibility that information leakage or register manipulation could expose the domain to attacks from a rogue SMI handler. The role of the STM is to facilitate preservation of integrity and confidentiality properties from threats resulting from this situation.

Therefore, the STM must govern the access to the interrupted domain's register state in a way that appropriately protects the interrupted context. This requires enforcing a set of SMRAM state save rules as well as controlling access to the extended register state which is not included in the MLE state save area (e.g. floating point, mmx, etc.).

Protected domains provide a mechanism for the STM to limit the SMI handler's register access to only that which is required in the Intel instruction set architecture (ISA) to process the IO instruction.

10.3.1 BIOS guaranteed access

The BIOS **must** declare all IO ports that are armed to trap to SMM in the Required Resource List using `STM_RSC_TRAPPED_IO_DESC` entries, as well as any other resources that must be accessed by the SMI handler in the course of servicing SMIs. **If this is done correctly, under no circumstance can BIOS be denied access to the registers or other resources necessary to handle synchronous SMIs. If the BIOS resource list is incorrect, the MLE may protect a resource needed by BIOS, potentially resulting in platform failures. Therefore, it is critically important that BIOS correctly declare its resource list.**

10.3.2 Synchronous SMI during execution of protected software

A protected domain is free to access IO ports that are not on BIOS' Required Resource List, as these should not trap to SMM.

There are two error conditions that can result in a synchronous SMI trap during execution of a protected domain:

- The protected domain accesses an IO port that is on the Required Resource List. In this case, the STM must degrade the protections to allow the BIOS sufficient access to process the SMI.
- The BIOS arms an SMI trap for a port that is not listed on the Required Resource List, and this port is subsequently touched by a protected domain. In this case, the STM must enforce the policy described below. Depending on the reason BIOS has trapped the IO, it may not have sufficient information or access to execute the corresponding handler.

10.3.3 State save area generation and propagation rules

When an SMI occurs, the STM will create the normal SMM state save area based on content in the interrupted domain's VMCS and register values. The state save area may be partially or completely scrubbed based on policy maintained by the STM in the VMCS database (see section 10.4).

The classes of protection domains and associated rules are given in the table below. Any bit combinations for domain type that are not listed below are reserved and should not be used. This set of rules provides flexibility to allow for controlled IO operations from otherwise protected domains.

Prior to the first entry into the BIOS SMI handler for each SMI event, the STM must program the bit fields `DomainType` and `XStatePolicy` located in the `TXT_PROCESSOR_SMM_DESCRIPTOR` to inform BIOS of the active protection policy. This must be done for all logical processors. BIOS must declare all IO ports that have been armed for SMI trapping in order to be assured that the required hardware access will be granted by the STM. If this is done correctly, the SMI handler should not need to be concerned with checking these fields.

Irrespective of `DomainType`, the STM must always correctly populate SMM_REV_ID field. When `DomainType != UNPROTECTED`, all fields except for those explicitly identified in the table below, must be cleared to 0. In any case where propagation of state save is allowed, the STM must check the `TXT_PROCESSOR_SMM_DESCRIPTOR.SmramToVmcsRestoreRequired` bit for each logical processor prior to resuming the interrupted context. If this bit is set, the STM must copy the SMRAM state save area back to the VMCS and register state so that any changes made will be reflected in the resumed context. The STM must take care to ensure proper propagation rules are followed.

The table below depicts the register scrubbing and register change propagation rules for the different domain types. Note: EAX/AX/AL widths depend on IO width. The table's columns signify the following:

"Security Attribute: Confidentiality" and "Security Attribute Integrity" columns: indicate whether the domain has confidentiality and/or integrity properties respectively

"Disallowed IO: In" and "Disallowed IO: Out" columns: indicate which IO operation (e.g. OUT Byte, IN WORD, etc) are disallowed.

"Registers Populated" column: describes which save state registers the STM makes accessible to the SMI handler.

"Changes Propagated" column: shows which register changes are propagated by the STM back to the MLE or guest of the MLE.

"Extended State Handling" column: shows the rules for whether the extended register state is visible/modifiable by the SMI handler.

Table 10-2. Domain types/register scrubbing and propagation rules

Domain Type Name	Security Attribute: Confidentiality	Security Attribute: Integrity	Disallowed IO: In	Disallowed IO: Out	Registers Populated	Changes Propagated	Extended State Handling	Notes
UNPROTECTED (0x00)	0	0	0	0	All	All legal changes	Visible and modifiable by SMI handler	
INTEGRITY_ PROT_OUT_IN (0x04)	0	1	0	0	All	See notes	Per VMCS DB	No change propagation for IN from port not on BIOS trap list Changes to EAX/AX/AL propagated for IN from port on BIOS IO trap list. No change propagation for OUT
FULLY_PROT_ OUT_IN (0x0C)	1	1	0	0	See Notes	See Notes	Per VMCS DB	Only SMM_REV_ID populated for any IO not on BIOS trap list Only SMM_REV_ID, IO_MISC, IO_MEM_ADDR, and DX register populated for IN on BIOS trap list Only SMM_REV_ID, IO_MISC, IO_MEM_ADDR, DX, and EAX/AX/AL populated for OUT on BIOS trap list. No changes propagated for OUT. Changes to EAX/AX/AL propagated for IN from port on BIOS IO trap list.
FULLY_PROT (0x0F)	1	1	1	1	SMM_REV_ID	None	Per VMCS DB	

10.3.4 Asynchronous SMIs and Protected Domains

The treatment of asynchronous SMIs requires special handling in protected domains. Asynchronous SMIs could occur at any time during execution and the STM needs to protect the register state of interrupted protected domains. As previously noted, the SMI handler doesn't need access to CPU register state to process these SMIs.

The following table depicts the register scrubbing and register change propagation rules.

Table 10-3. Asynchronous SMIs and Protected Domains

Domain Type	Registers Populated	Changes Propagated	Extended State Handling
UNPROTECTED (0x00)	All	All legal changes	Visible and modifiable by the SMI handler
INTEGRITY_PROT_OUT_IN (0x04)	SMM_REV_ID	None	Per VMCS DB
FULLY_PROT_OUT_IN (0x0C)	SMM_REV_ID	None	Per VMCS DB
FULLY_PROT (0x0F)	SMM_REV_ID	None	Per VMCS DB

10.3.5 I/O Instruction Restart

In all cases where an IO is trapped that is on the BIOS trap list and the domain type doesn't preclude the IO:

- IO_MISC (and IO_MEM_ADDR if IO_MISC[0] is set) area are valid.
- The value of I/O Instruction Restart field in the SMM state save area after RSM must be honored by the STM

If an IO is trapped that is not on the BIOS trap list:

- If the domain type does not preclude the IO, the I/O Instruction Restart field is ignored by the STM.
- If the domain type precludes the IO, but the IO in question has not been protected by the MLE, the value of I/O Instruction Restart field in the SMM state save area after RSM must be honored by the STM
- If the domain type precludes the IO and the IO in question has been protected by the MLE, the I/O Instruction Restart field is ignored by the STM.

10.3.6 Domain type degradation rules

The domain degradation feature facilitates profiling of protected domains to verify that they are operating within the restrictions imposed by their domain type.

As previously noted, the BIOS will be granted access to any resource that it specifies in its Required Resource List. This section describes the STM's treatment of scenarios

where there is an incompatibility between the domain type of a protected domain and the SMI handler's declared access to a trapped IO resource.

Synchronous SMIs result from trapped IO instructions and the BIOS SMI handler is very likely to require some access to register state in the state save area.

Therefore, a FULLY_PROT domain should avoid IO operations that are declared by BIOS via an STM_RSC_TRAPPED_IO_DESC. If such an IO operation occurs, the STM will degrade the protection properties, if allowed by the DegradationPolicy, from FULLY_PROT to FULLY_PROT_OUT_IN based on the rules given in **Error! Reference source not found.**. If the DegradationPolicy doesn't allow the degradation, the STM will reset the platform.If any domain performs an IO operation that is declared by BIOS to be a synchronous SMI API via an STM_RSC_TRAPPED_IO_DESC, the domain type is degraded to UNPROTECTED, if allowed by the DegradationPolicy, otherwise the STM resets the platform. Therefore, IO operations to these ports should be avoided by protected domains since the effect of such an operation is to open the domain's TCB to the SMI handler.

If logging is enabled and the event bitmap specifies it, the STM must record all domain type degradations using an ENTRY_EVT_MLE_DOMAIN_TYPE_DEGRADED log entry. See Appendix E.

If in any case domain type degradation is needed, but precluded by the MLE's DegradationPolicy in the VMCS database, the STM must write STM_CRASH_DOMAIN_DEGRADATION_FAILURE to the TXT.ERRORCODE register, followed by a TXT.CMD.SYS_RESET to force a system reset. This is due to a fundamental conflict between the platform and the specified policy.

Table 10-4 depicts the domain degradations that occur based on the starting protected domain type (e.g. the protected domain type at the time of the SMI), the minimum allowed domain type as specified by the MLE's DegradationPolicy, and the type of synchronous SMI to a trapped IO port.

The degradation logic is as follows:

- If the starting domain type permits an IN or OUT instruction and the domain generates an IN or OUT to a trapped port on the SMI handler's required resource list, no degradation occurs as this is permitted by the starting domain type.
- If the starting domain doesn't permit an IN or OUT (e.g. a FULLY_PROT domain) but an IN or OUT occurs to a trapped port on the SMI handler's required resource list, the domain will degrade to the next lower DegradationPolicy level (e.g. FULLY_PROT_OUT_IN.) The levels are (from highest to lowest) FULLY_PROT, FULLY_PROT_OUT_IN, INTEGRITY_PROT_OUT_IN, UNPROTECTED.
- If an IN to a trapped SMI API port occurs, this results in a degradation to the UNPROTECTED domain type, assuming this is permitted by the DegradationPolicy. (If this is not permitted, the STM will trigger a platform reset.) The SMI handler would likely need additional register state to process the SMI and this isn't permitted, except in the UNPROTECTED domain type. This logic also applies similarly to OUT instructions.

- If the minimum domain type (based on DegradationPolicy) precludes a transition to a lower protection level that would be required to process the IO, the STM will trigger a platform reset. For example, if a FULLY_PROT domain performs an IN to a trapped IO port but the minimum permitted domain type is FULLY_PROT, the STM is not permitted to degrade the protection level and can only reset the platform. This logic also applies similarly to OUT instructions.

Table 10-4. Protected domain degradations

Starting Domain Type	Min. Domain Type	Result of IN: Sync SMI Trap	Result of OUT: Sync SMI Trap
FULLY_PROT	FULLY_PROT	Reset	Reset
" "	FULLY_PROT_OUT_IN	FULLY_PROT_OUT_IN	FULLY_PROT_OUT_IN
" "	INTEGRITY_PROT_OUT_IN	FULLY_PROT_OUT_IN	FULLY_PROT_OUT_IN
" "	UNPROTECTED	FULLY_PROT_OUT_IN	FULLY_PROT_OUT_IN
FULLY_PROT_OUT_IN	FULLY_PROT_OUT_IN	FULLY_PROT_OUT_IN	FULLY_PROT_OUT_IN
" "	INTEGRITY_PROT_OUT_IN	FULLY_PROT_OUT_IN	FULLY_PROT_OUT_IN
" "	UNPROTECTED	FULLY_PROT_OUT_IN	FULLY_PROT_OUT_IN
INTEGRITY_PROT_OUT_IN	INTEGRITY_PROT_OUT_IN	INTEGRITY_PROT_OUT_IN	INTEGRITY_PROT_OUT_IN
" "	UNPROTECTED	INTEGRITY_PROT_OUT_IN	INTEGRITY_PROT_OUT_IN
UNPROTECTED	UNPROTECTED	UNPROTECTED	UNPROTECTED

10.3.7 IA32_EFER handling

The STM must populate the value of IA32_EFER in the SMRAM state save area for UNPROTECTED contexts. Furthermore, since the STM itself will load IA32_EFER on VMEXIT, it must be able to determine the interrupted context's value for IA32_EFER in order to ensure it can be properly restored on resume. Depending on the processor features implemented, the STM may be able to obtain the interrupted value of IA32_EFER directly from the VMCS. In other cases, the STM will need to use the VMX exit control for MSRs and put IA32_EFER in the list of stored MSRs to obtain the interrupted value of IA32_EFER.

If bit 20 (Save IA32_EFER) of the IA32_VMEXIT_CTLS MSR is set, this indicates that the processor will store the value of IA32_EFER in the VMCS on VMEXIT when the corresponding VMEXIT control is set. The STM should enable and use this feature when available, since it will result in better SMI performance than the alternate mechanism described below.

If bit 20 of the IA32_VMEXIT_CTLS MSR is clear, this indicates that saving of IA32_EFER in the VMCS is not supported. In this case, the STM should use the control for MSRs to construct an MSR store list and add IA32_EFER to this list. It should also ensure that IA32_EFER is restored properly using the corresponding load mechanism.

The same methods apply for VMCALLs and protection exceptions from the BIOS SMI handler.

10.3.8 MLE root or guest extended register state

The processor supports many registers in addition to the register context represented by the save state area (floating point, MMX, XMM, etc.); these registers also represent potential confidentiality and integrity leaks unless properly insulated by the STM.

The STM maintains a separate extended register state policy for each VMCS in the VMCS database. Except for `UNPROTECTED` domains which are open, the default STM policy is to completely isolate the interrupted domain's extended register state from the BIOS SMI handler (`XSTATE_SCRUB`.) This policy can be modified using the `ManageVmcsDatabaseVMCALL()`. See section 9.7.

Typical STM implementations will use XSAVE/XRSTOR to manage extended state save and restore operations. For performance reasons, the MLE should explicitly turn off extended state protections if the domain in question does not use any extended state registers (`XSTATE_READWRITE`.) The scrub of extended processor state can be omitted when the interrupted domain doesn't use these registers.

The STM is not required to save or restore processor extended state if it is not indicated to do so in the VMCS database. If the STM itself uses any of these registers, this usage must be transparent to both the MLE and the BIOS SMI handler.

The expected flow of the STM regarding processor extended state is as follows:

```
Lookup interrupted context's VMCS in the VMCS database
If (policy is XSTATE_SCRUB)
    Save current register context using XSAVE with all bits set in the mask
    Clear all register context using XRSTOR with zeroed save buffer
Endif

If (policy is XSTATE_READONLY)
    Save current register context using XSAVE with all bits set in the mask
Endif
…
VMRESUME SMM guest
RSM
…

If (policy is XSTATE_SCRUB  or policy is XSTATE_READONLY)

    Restore previously saved processor extended state using XRSTOR with all bits set in the
    mask
Endif
…
VMRESUME interrupted MLE
```

See the Intel® 64 and IA-32 Architectures Software Developer's Manual for details about the XSAVE/XRSTOR instructions and the processor extended state.

10.4 VMCS database

The STM cannot derive the domain type of an interrupted context directly from the interrupted VMCS and register context. In order to make correct policy decisions the STM needs some assistance from the MLE. The MLE provides these hints using the `ManageVmcsDatabaseVMCALL()`. The STM will maintain an internal VMCS database to

maintain the association between VMCS and domain type and extended state handling policy.

During initialization, the MLE root should invoke `ManageVmcsDatabaseVMCALL()` to establish the protection rules for the root VMCS that defines the MLE itself. It should also invoke `ManageVmcsDatabaseVMCALL()` when the MLE root is created and each time an MLE guest is created before the MLE guest is allowed to execute, and each time an MLE guest is destroyed after no more execution in that MLE guest is possible. Failure to do this can result in performance degradation and incorrect protection policies being applied.

When an SMM exit occurs due to an SMI pin event, the STM determines whether or not to fully populate the SMRAM state save based on the interrupted entity's VMCS database entry as follows:

1. The STM invokes VMREAD to obtain the interrupted entity's controlling VMCS pointer from the executive-VMCS pointer field of the SMI VMCS.

2. The STM searches its internal VMCS database until it locates the interrupted context's entry or the database has been completely searched and no match is found.

3. If a match is found, the STM uses the matching record to determine the state save area generation and propagation rules prior to transferring control to the BIOS SMI handler. If no match is found, the STM must assume a type of `FULLY_PROT` with extended state access denied and `DegredationPolicy` set to `FULLY_PROT_OUT_IN`.

§

11 Fatal error handling

Fatal errors detected by the STM in its own operation must result in a TXT reset. Before the STM writes to the TXT.CMD.SYS-RESET register, it must write an error code into the TXT.ERRORCODE register. The error code has the following format:

Table 11-1: STM Error Code Format

Bit 31	Valid	must be set to 1
Bits 30	External/internal	0=processor, 1=software; STM must set this bit to 1
Bit 16:29	Reserved	must be written with all 0's
Bit 15	AC/Monitor	set to '1' for VMM or STM errors, clear for AC module errors; STM must set this bit to 1
Bit 14	MLE/STM	0=MLE, 1=STM; STM must set this bit to 1
Bit 13: 0	Error Code	see Appendix D

This yields a TXT.ERRORCODE value of 1100-0000-0000-0000--11xx-xxxx-xxxx-xxxx.

Values in the set {0xC000Dxxx, 0xC000Exxx, 0xC000Fxxx} are reserved by this specification. All other values are available to the STM writer for proprietary error codes (0xC000Cxxx).

§

12 Support of a non-TXT launch

According to the IA32 SDM, the STM can be launched without TXT.

If the STM needs to support a non-TXT launch, the STM should check the TXT.STS.SENTER.DONE bit. If TXT.STS.SENTER.DONE is zero, it means the current launch is a non-TXT launch.

In a non-TXT launch, the STM needs to find some information from alternate sources. See Table 12-1.

Table 12-1: STM non-TXT launch

Information	**TXT launch**	**Non-TXT launch**
CPU Number	BiosToOsData.NumLogProcs	ACPI MADT table
PCI Express Base	SinitToMleData.SinitMdrTable PCIe configuration region	ACPI MCFG table
Reset register	TXT.CMD.SYS_RESET	ACPI FADT table
Error Code	TXT.ERRORCODE	N/A

§

Appendix A STM_RESOURCE_LIST

A.1 Overview

The BIOS SMI handler statically communicates its hardware resource requirements to the STM via a byte stream located in SMRAM which contains a variable list of hardware resources.

The MLE dynamically communicates its hardware resource protection requirements to the STM via VMCALL-based APIs exposed by the STM to the MLE.

Both interfaces use the same data description stream format which is given in Backus-Naur form below:

```
<STM_RESOURCE_LIST> ::=
                <STM_RSC_ALL>
              | <STM_RSC_DIFFERENTIATED_RESOURCE_LIST>

<STM_RSC_ALL> ::= <STM_RSC_ALL_RESOURCES_DESC> <STM_RSC_END>

<STM_RSC_DIFFERENTIATED_RESOURCE_LIST> ::=
              { <STM_RSC_DIFFERENTIATED_RESOURCE > } <STM_RSC_END>

<STM_RSC_DIFFERENTIATED_RESOURCE > ::=
                <STM_RSC_MEM_DESC>
              | <STM_RSC_IO_DESC>
              | <STM_RSC_MMIO_DESC>
              | <STM_RSC_MSR_DESC>
              | <STM_RSC_PCI_CFG_DESC>
              | <STM_RSC_TRAPPED_IO_DESC>
```

The root data structures for the `STM_RESOURCE_LIST` byte stream are given below:

```
// resource descriptor types are given below
#define END_OF_RESOURCES     0
#define MEM_RANGE            1
#define IO_RANGE             2
#define MMIO_RANGE           3
#define MACHINE_SPECIFIC_REG 4
#define PCI_CFG_RANGE        5
#define TRAPPED_IO_RANGE     6
#define ALL_RESOURCES        7
#define REGISTER_VIOLATION   8
#define MAX_DESC_TYPE        8

typedef struct {
   UINT32                    RscType;
   UINT16                    Length;
   UINT16                    ReturnStatus:1;   // bitfield
   UINT16                    Reserved:14;      // bitfield; must be 0
   UINT16                    IgnoreResource:1; // bitfield
```

```
} STM_RSC_DESC_HEADER;
```

`RscType` - This indicates the type of resource being described and is common to all resource descriptors. This field must have a value no greater than `MAX_DESC_TYPE`.

`Length` - This indicates the length in bytes of the type of resource being described, including both the header and data portions of the descriptor. The `Length` field must always be correct, even if the `IgnoreResource` bit is set.

`ReturnStatus` – This bit is set to 1 to indicate success or cleared to 0 to indicate failure for each resource passed into `ProtectResourceVMCALL()` or `UnProtectResourceVMCALL()`. It is an output only and is ignored on input. This bit is reserved for all other uses of `STM_RSC_DESC_HEADER`.

`IgnoreResource` – If this bit is set to 1, the STM will not process the data but will skip to the next resource descriptor in the stream. If the `IgnoreResource` bit is set to 1, the remainder of the resource descriptor header is ignored, except for the `Length` field which must be valid regardless of the state of the `IgnoreResource` bit. The intent of this bit is to allow for implementation optimizations that require statically defined resource lists that can have elements within them easily switched on and off.

A.2 Resource types

A.2.1 STM_RSC_END

```
typedef struct {
   STM_RSC_DESC_HEADER     Hdr;
   UINT64                  ResourceListContinuation;
} STM_RSC_END;
```

`Hdr.RscType` - Must be `END_OF_RESOURCES`

`Hdr.Length` - Must be sizeof (`STM_RSC_END`), which is 16.

`ResourceListContinuation` – If `ResourceListContinuation == 0` then this marks the end of the resource list. However, if `ResourceListContinuation != 0`, then the value of `ResourceListContinuation` indicates the physical address of the next descriptor in the resource list. This allows for a physically discontiguous resource list.

A.2.2 STM_RSC_MEM_DESC

```
typedef struct {
   STM_RSC_DESC_HEADER     Hdr;
   UINT64                  Base;
   UINT64                  Length;
   struct {
      UINT32               RWXAttributes:3; // bitfield
      UINT32               Reserved:29;     // bitfield, must be 0
   };
   UINT32                  Reserved_2;      // must be 0
} STM_RSC_MEM_DESC;
```

Hdr.RscType - Must be MEM_RANGE

Hdr.Length - Must be sizeof (STM_RSC_MEM_DESC), which is 32

Base - Indicates the physical base address of the memory range being described. While this data structure specifies a byte granular range, this does not imply that the STM necessarily supports byte granular memory ranges. If the STM does not support byte granular memory ranges, then any byte claimed within a 4K page implies the entire page is included in the STM_RESOURCE_LIST and the STM operates on a page granular basis. The STM capabilities are exposed to the MLE via the InitializeProtectionVMCALL() (see section 9.1).

Length - Indicates the length in bytes of the range being described. The STM will consider a length of 0 to be an error.

RWXAttributes - When used by the BIOS SMI handler to declare required memory resources, the RWXAttributes field indicates the access types (read, write, execute) required by the BIOS. When used by the MLE to request protection of a given memory range, the RWXAttributes field indicates what types of access are prohibited. For both, when a bit is set, it indicates the request is asserted, and when the bit is clear, the request is not asserted.

Bit 0: indicates read access

Bit 1: indicates write access

Bit 2: indicates execute access

Allowable values:

000: no access

001: Read access; no write; no execute

011: Read and write access; no execute

101: Read and execute; no write

111: Full access

A.2.3 STM_RSC_IO_DESC

```
typedef struct {
   STM_RSC_DESC_HEADER     Hdr;
   UINT16                  Base;
   UINT16                  Length;
   UINT32                  Reserved;     // must be 0
} STM_RSC_IO_DESC;
```

Hdr.RscType - Must be IO_RANGE

Hdr.Length - Must be sizeof (STM_RSC_IO_DESC), which is 16

`Base` - Indicates the base IO port number of the range being described

`Length` - Indicates the length in bytes of the range being described. The STM will consider a length of 0 to be an error.

In the case of a BIOS Required Resource List, this type is used to indicate the IO ports that the BIOS SMI handler may read from/write to as part of its operation. This is distinct from the IO ports that BIOS configures to trap into the SMI handler (those are specified by the STM_RSC_TRAPPED_IO_DESC type).

A.2.4 STM_RSC_MMIO_DESC

```
typedef struct {
   STM_RSC_DESC_HEADER     Hdr;
   UINT64                  Base;
   UINT64                  Length;
   struct {
      UINT32               RWXAttributes:3; // bitfield
      UINT32               Reserved:29;     // bitfield, must be 0
   };
   UINT32                  Reserved_2;      // must be 0
} STM_RSC_MMIO_DESC;
```

`Hdr.RscType` - Must be `MMIO_RANGE`

`Hdr.Length` - Must be sizeof (`STM_RSC_MMIO_DESC`), which is 32

`Base` - Indicates the physical base address of the range being described. While this data structure specifies a byte granular range, this does not imply that the STM necessarily supports byte granular MMIO ranges. If the STM does not support byte granular MMIO, then any byte claimed within a 4K page implies the entire page is included in the `STM_RESOURCE_LIST` and the STM operates on a page granular basis. The STM capabilities are exposed to the MLE via the `InitializeProtectionVMCALL()` (see section 9.1).

`Length` - Indicates the length in bytes of the range being described. The STM will consider a length of 0 to be an error.

`RWXAttributes` - When used by the BIOS SMI handler to declare required MMIO resources, the `RWXAttributes` field indicates the access types (read, write, execute) required by the BIOS. When used by the MLE to request protection of a given MMIO range, the `RWXAttributes` field indicates what types of access are prohibited. For both, when a bit is set, it indicates the request is asserted, and when the bit is clear, the request is not asserted.

Bit 0: indicates read access

Bit 1: indicates write access

Bit 2: indicates execute access

Allowable values:

000: no access

001: Read access; no write; no execute

011: Read and write access; no execute

101: Read and execute; no write

111: Full access

A.2.5 STM_RSC_MSR_DESC

```
typedef struct {
   STM_RSC_DESC_HEADER     Hdr;
   UINT32                  MsrIndex;
   struct {
      UINT8                Attributes:1; //Bit 0 is VMX Root mode request
      UINT8                Reserved1:7;
   };
   UINT8                   Reserved2[3];
   UINT64                  ReadMask;
   UINT64                  WriteMask;
} STM_RSC_MSR_DESC;
```

`Hdr.RscType` - Must be `MACHINE_SPECIFIC_REG`

`Hdr.Length` - Must be `sizeof (STM_RSC_MSR_DESC)`, which is 32

`MsrIndex` - Indicates which MSR is being described

`Attributes –` Describes handling options for this MSR.

Bit 0: Indicates that the MSR requires VMX Root Mode execution (for example IA32_BIOS_UPDT_TRIG MSR.) The STM will configure the MSR bitmap such that an SMI handler's access of this MSR will trigger a VM Exit. The STM will then process the request on behalf of the SMI handler guest. If a GP fault occurs as a result of the STM's access, the STM will return control to the BIOS exception handler.

`ReadMask` - Indicates for which bits in the MSR read access (if from BIOS SMI handler) or read protection (if from MLE) is being requested. If the STM does not support bit granular MSR resource control, then any bit claimed within the whole MSR implies the entire MSR is included in the `STM_RESOURCE_LIST` and the STM operates on a whole MSR granular basis. The STM capabilities are exposed to the MLE via the `InitializeProtectionVMCALL()` (see section 9.1).

`WriteMask` - Indicates for which bits in the MSR write access (if from BIOS SMI handler) or write protection (if from MLE) is being requested. If the STM does not support bit granular MSR resource control, then any bit claimed within the whole MSR implies the entire MSR is included in the `STM_RESOURCE_LIST` and the STM operates on a whole MSR granular basis. The STM capabilities are exposed to the MLE via the `InitializeProtectionVMCALL()` (see section 9.1).

A.2.6 STM_RSC_PCI_CFG_DESC

```
typedef struct {
   STM_RSC_DESC_HEADER     Hdr;
   struct {
      UINT16               RWAttributes:2; // bitfield
      UINT16               Reserved:14;    // bitfield; must be 0
   };
   UINT16                  Base;
   UINT16                  Length;
   UINT8                   LastNodeIndex;
   UINT8                   OriginatingBusNumber;
   STM_PCI_DEVICE_PATH_NODE PciDevicePath[LastNodeIndex + 1];
} STM_RSC_PCI_CFG_DESC;
```

`Hdr.RscType` - Must be `PCI_CFG_RANGE`

`Hdr.Length` – Must be the size in bytes of the whole `STM_RSC_PCI_CFG_DESC`. The statically sized portion is 16 bytes. The size of the `PciDevicePath` array varies depending on how many elements are in the array. Each element of the `PciDevicePath` array is 6 bytes long. Therefore `Hdr.Length` must be assigned the value `16 + (LastNodeIndex + 1) * 6`.

`RWAttributes` - When used by the BIOS SMI handler to declare required PCI configuration resources, the `RWAttributes` field indicates the access types (read, write) required by the BIOS. When used by the MLE to request protection of a given PCI configuration resource, the `RWAttributes` field indicates what types of access are prohibited. For both, when a bit is set, it indicates the request is asserted, and when the bit is clear, the request is not asserted.

Bit 0: indicates read access

Bit 1: indicates write access

`Base` - Indicates the offset of the first PCI configuration register being described.

`Length` - Indicates the number of bytes of PCI configuration space. The STM will consider a length of 0 to be an error.

`LastNodeIndex` - Indicates the array index of the last element in the array of `STM_PCI_DEVICE_PATH_NODE`.

`OriginatingBusNumber`- Indicates the starting bus number for the PCI device

`PciDevicePath` - Indicates which PCI function is associated with the range. The definition for `STM_PCI_DEVICE_PATH_NODE` is taken from section 9.3.2.1 of the Unified Extensible Firmware Interface Specification, Version 2.3, Errata C. This specification can be obtained from http://www.uefi.org/specs/download/UEFI_Spec_2_3_Errata_C.pdf.

For implementation convenience, the STM_PCI_DEVICE_PATH_NODE definition given here is identical to the referenced UEFI specification definition. In the event that the UEFI definition diverges from the STM definition due to future changes in either specification, the STM usage will remain as described in this specification. In other words, for the purposes of STM, the definition given here is normative.

```
typedef struct {
   UINT8      Type;      // must be 1, indicating Hardware Device Path
   UINT8      Subtype;   // must be 1, indicating PCI
   UINT16     Length;    // sizeof(STM_PCI_DEVICE_PATH_NODE) which is 6
   UINT8      PciFunction;
   UINT8      PciDevice;
} STM_PCI_DEVICE_PATH_NODE;
```

The PciDevicePath field is an array of STM_PCI_DEVICE_PATH_NODE structures. The array has LastNodeIndex + 1 elements, where the smallest legal PciDevicePath array has a single element (LastNodeIndex == 0). In the event that there is more than one element in the array, all elements in the array except for the last one represent bridge devices. The last element of the array always indicates the PCI device and function described by this STM_RSC_PCI_CFG_DESC.

There are two differences between an STM PCI device path and a UEFI PCI device path:

1) The STM PCI device path length is given explicitly by LastNodeIndex + 1, while UEFI uses an end marker node.

2) The STM PCI device path assumes a system topology with legacy PCI configuration addresses at io ports 0xCF8 and 0xCFC; and where the PCIe* memory mapped configuration base address is derived from the hardware by SINIT and given to the MLE via the PCIe Memory Descriptor Record in the SINIT to MLE data table portion of the TXT heap. In contrast, UEFI requires that PCI device paths begin with an ACPI device path node to identify the host bridge. Using an ACPI node would be contrary to STM security objectives.

A.2.7 STM_RSC_TRAPPED_IO_DESC

```
typedef struct {
   STM_RSC_DESC_HEADER     Hdr;
   UINT16                  Base;
   UINT16                  Length;
   struct {
      UINT16               In:1;          // bitfield
      UINT16               Out:1;         // bitfield
      UINT16               Api:1;         // bitfield
      UINT16               Reserved1:13;  // bitfield; must be 0
   };
   UINT16               Reserved2;

} STM_RSC_TRAPPED_IO_DESC;
```

Hdr.RscType - Must be TRAPPED_IO_RANGE

Hdr.Length - Must be sizeof (STM_RSC_TRAPPED_IO_DESC), which is 24.

`Base` - Indicates the physical base address of the range being described

`Length` - Indicates the length in bytes of the range being described. The STM will consider a length of 0 to be an error.

`Out` - Indicates the BIOS traps OUT operations to the specified port. This information is used by the STM when deriving state save generation rules. See section 10.3.3.

`In` - Indicates the BIOS traps IN operations from the specified port range. This information is used by the STM when deriving state save generation and propagation rules. See section 10.3.3.

`Api` - Indicates the port range in question is a synchronous SMI API. See section 10.2.2

The `TRAPPED_IO_RANGE` is different than other types of resources in that it doesn't describe resources consumed or reserved; rather it describes specific hardware behavior associated with IO ports that are enabled to generate synchronous SMIs. An instance of `STM_RSC_TRAPPED_IO_DESC` describes an IO port(s) that will generate synchronous SMIs when accessed from non-SMM software. If the SMI handler must also access the port, then the port must also be described as a normal IO Range. The MLE should never use `STM_RSC_TRAPPED_IO_DESC` when calling `ProtectResourceVMCALL()`. Such a protection request makes no sense and must always be denied by the STM.

A.2.8 STM_RSC_ALL_RESOURCES_DESC

```
typedef struct {
   STM_RSC_DESC_HEADER     Hdr;
} STM_RSC_ALL_RESOURCES_DESC;
```

`Hdr.RscType` - Must be `ALL_RESOURCES`

`Hdr.Length` - Must be sizeof (`STM_RSC_ALL_RESOURCES_DESC`), which is 8.

An instance of `STM_RSC_ALL_RESOURCES_DESC` is shorthand for describing all resources not explicitly reserved by the BIOS. For high security environments, it facilitates a simple method for an MLE to request "protect all", or "unprotect all" and alleviates the need to build complete resource lists for all things not claimed by BIOS. It is also useful for testing the correctness of the BIOS required resource list.

BIOS should not include `STM_RSC_ALL_RESOURCES_DESC` in the required resources list as this would render the entire STM moot.

A.2.9 STM_REGISTER_VIOLATION_DESC

```
typedef struct {
    STM_RSC_DESC_HEADER     Hdr;
    UINT32                  RegisterType; // RegisterCr0, RegisterCr2…
    UINT32                  Reserved;
    UINT64                  ReadMask;
```

```
   UINT64                  WriteMask;
} STM_REGISTER_VIOLATION_DESC;
```

The STM_REGISTER_VIOLATION_DESC is used solely for logging improper control register accesses (CR0, CR2, etc) by the SMI handler. Unlike the resource type descriptors, it is not used to specify any resources to be protected or resources that the SMI handler requires access to.

The following enum lists the possible values for RegisterType.

```
#typedef enum {
  CR0,
  CR2,
  CR3,
  CR4,
  CR8
} REGISTER_VIOLATION_TYPE;
```

§

Appendix B VMCALL API Numbers

```
// API number convention: BIOS facing VMCALL interfaces have bit 16 clear
#define STM_API_MAP_ADDRESS_RANGE                       0x00000001
#define STM_API_UNMAP_ADDRESS_RANGE                     0x00000002
#define STM_API_ADDRESS_LOOKUP                          0x00000003
#define STM_API_RETURN_FROM_PROTECTION_EXCEPTION        0x00000004

// API number convention: MLE facing VMCALL interfaces have bit 16 set
#define STM_API_START                                   0x00010001
#define STM_API_STOP                                    0x00010002
#define STM_API_PROTECT_RESOURCE                        0x00010003
#define STM_API_UNPROTECT_RESOURCE                      0x00010004
#define STM_API_GET_BIOS_RESOURCES                      0x00010005
#define STM_API_MANAGE_VMCS_DATABASE                    0x00010006
#define STM_API_INITIALIZE_PROTECTION                   0x00010007
#define STM_API_MANAGE_EVENT_LOG                        0x00010008
```

§

Appendix C Return codes

```
#define STM_SUCCESS                                  0x00000000
#define SMM_SUCCESS                                  0x00000000

// all error codes have bit 31 set

// STM errors have bits 31 and 16 set
#define ERROR_STM_SECURITY_VIOLATION                 0x80010001
#define ERROR_STM_CACHE_TYPE_NOT_SUPPORTED           0x80010002
#define ERROR_STM_PAGE_NOT_FOUND                     0x80010003
#define ERROR_STM_BAD_CR3                            0x80010004
#define ERROR_STM_PHYSICAL_OVER_4G                   0x80010005
#define ERROR_STM_VIRTUAL_SPACE_TOO_SMALL            0x80010006
#define ERROR_STM_UNPROTECTABLE_RESOURCE             0x80010007
#define ERROR_STM_ALREADY_STARTED                    0x80010008
#define ERROR_STM_WITHOUT_SMX_UNSUPPORTED            0x80010009
#define ERROR_STM_STOPPED                            0x8001000A
#define ERROR_STM_BUFFER_TOO_SMALL                   0x8001000B
#define ERROR_STM_INVALID_VMCS_DATABASE              0x8001000C
#define ERROR_STM_MALFORMED_RESOURCE_LIST            0x8001000D
#define ERROR_STM_INVALID_PAGECOUNT                  0x8001000E
#define ERROR_STM_LOG_ALLOCATED                      0x8001000F
#define ERROR_STM_LOG_NOT_ALLOCATED                  0x80010010
#define ERROR_STM_LOG_NOT_STOPPED                    0x80010011
#define ERROR_STM_LOG_NOT_STARTED                    0x80010012
#define ERROR_STM_RESERVED_BIT_SET                   0x80010013
#define ERROR_STM_NO_EVENTS_ENABLED                  0x80010014
#define ERROR_STM_OUT_OF_RESOURCES                   0x80010015
#define ERROR_STM_FUNCTION_NOT_SUPPORTED             0x80010016
#define ERROR_STM_UNPROTECTABLE                      0x80010017
#define ERROR_STM_VMCS_PRESENT                       0x80010018
#define ERROR_STM_UNSUPPORTED_MSR_BIT                0x80010019
#define ERROR_STM_UNSPECIFIED                        0x8001FFFF



// SMM errors have bits 31 and 17 set
#define ERROR_SMM_BAD_BUFFER                         0x80020001
#define ERROR_SMM_UNSPECIFIED                        0x8002FFFF

// Errors that apply to both have bits 31, 15, 16, and 17 set
#define ERROR_INVALID_API                            0x80038001
#define ERROR_INVALID_PARAMETER                      0x80038002
```

§

Appendix D STM TXT.ERRORCODE crash codes

```
#define STM_CRASH_PROTECTION_EXCEPTION                0xC000F001
#define STM_CRASH_PROTECTION_EXCEPTION_FAILURE        0xC000F002
#define STM_CRASH_DOMAIN_DEGRADATION_FAILURE          0xC000F003

#define STM_CRASH_BIOS_PANIC                          0xC000E000
```

§

Appendix E Event log

E.1 Overview

This section defines the format and associated data structures of the STM's event log. Section 9.8 describes the STM function ManageEventLogVMCALL, which is used to configure and control the STM's logging capabilities.

The event type EVT_HANDLED_PROTECTION_EXCEPTION indicates there was a protection exception that occured that was handled by the BIOS provided protection exception handler. Note: Any protection exceptions that are not handled result in a platform reset.

The event type EVT_BIOS_ACCESS_TO_UNCLAIMED_RESOURCE indicates that the BIOS has accessed a hardware resource that was not claimed via the STM_RESOURCE_LIST indicated by BiosHwResourceRequirements (see section 6.1). The resource in question was also unclaimed by the MLE. The STM will allow this BIOS access. If logging of this event is enabled, the STM must log the first access to the resource in question. It may, but is not required to, log subsequent BIOS accesses to the same resource.

The event type EVT_MLE_RESOURCE_PROTECTION_GRANTED indicates that the STM added the resource to its list of resources to be protected.

The event type EVT_MLE_RESOURCE_UNPROTECT indicates that the STM has removed the resource from its protected resource list.

The event type EVT_MLE_RESOURCE_UNPROTECT_ERROR indicates that the STM encountered an error while attempting to unprotect a resource.

The STM will map into its own address space the array of physical pages in the order they are given by the MLE. This linear space then represents a contiguous array of STM_LOG_ENTRY[n], where n is the number of entries that will fit into the pages provided by the MLE for the log. This array is used in a circular manner. The first entry written is to STM_LOG_ENTRY[0], then next to STM_LOG_ENTRY[1], etc. When the end of the log is reached (STM_LOG_ENTRY[n-1]), the STM returns to the beginning of the log and writes the next entry at the beginning (STM_LOG_ENTRY[0]).

Related definitions:

```
typedef struct {
    LOG_ENTRY_HEADER  Hdr;
    LOG_ENTRY_DATA    Data;
} STM_LOG_ENTRY;

typedef struct {
    UINT32            EventSerialNumber;
    UINT16            Type;      // EVENT_TYPE
    struct {
        UINT16        Lock       :1; // bitfield
```

```
        UINT16        Valid     :1; // bitfield
        UINT16        ReadByMle :1; // bitfield
        UINT16        Wrapped   :1; // bitfield
        UINT16        Reserved  :12; // bitfield
    };
} LOG_ENTRY_HEADER;
```

`EventSerialNumber` is a monotonically increasing event count number. The first event to occur when the event log is created is given the value of `EventSerialNumber` = 0. The second event is given the value of `EventSerialNumber` = 1. The third, `EventSerialNumber` = 2. Etc.

`Type` indicates what type of event is being logged. Type must be a member of `EVENT_TYPE` (see section 9.8).

`Lock` is a semaphore used to prevent concurrent access to any given record by STM and MLE. When the STM or MLE wishes to access any given record (read or write), it must acquire the `Lock` semaphore using the BTS instruction. No updates to any record are allowed unless carry flag indicates that the semaphore was not locked when the BTS instruction was executed.

In all cases the `Lock` bit in the header must be acquired before reading or writing contents of any given event log entry. This is true for both the MLE and the STM. To avoid contention, both the STM and the MLE must avoid acquiring the `Lock` bit for more than one record at a time.

`Valid` is clear (0) indicates the `STM_LOG_ENTRY` is not valid. All contents of it should be ignored by the MLE.

`ReadByMle` must be cleared (0) by the STM when a record is initially created. When the MLE reads the entry, it must first acquire the `Lock`, and then set (1) `ReadByMle` before releasing the `Lock`. The `ReadByMle` bit is used by the STM to detect when the event log has wrapped and unread events are being overwritten.

`Wrapped` indicates that the log has wrapped and a new entry has overwritten another valid entry that had not been read by the MLE.

```
typedef union {
    ENTRY_EVT_LOG_STARTED               Started;
    ENTRY_EVT_LOG_STOPPED               Stopped;
    ENTRY_EVT_LOG_INVALID_PARAM         InvalidParam;
    ENTRY_EVT_LOG_PROTECTION_EXCEPTION  ProtectionException;
    ENTRY_EVT_LOG_HANDLED_PROTECTION_EXCEPTION  HandledProtectionException;
    ENTRY_EVT_BIOS_ACCESS_UNCLAIMED_RSC BiosUnclaimedRsc;
    ENTRY_EVT_MLE_RSC_PROT_GRANTED      MleRscProtGranted;
    ENTRY_EVT_MLE_RSC_PROT_DENIED       MleRscProtDenied;
    ENTRY_EVT_MLE_RSC_UNPROT            MleRscUnprot;
    ENTRY_EVT_MLE_RSC_UNPROT_ERROR      MleRscUnprotError;
    ENTRY_EVT_MLE_DOMAIN_TYPE_DEGRADED  MleDomainTypeDegraded;
} LOG_ENTRY_DATA;

typedef struct {
    UINT32          Reserved;
} ENTRY_EVT_LOG_STARTED;

typedef struct {
```

```
    UINT32          Reserved;
} ENTRY_EVT_LOG_STOPPED;

typedef struct {
    UINT32          VmcallApiNumber;
} ENTRY_EVT_LOG_INVALID_PARAM;

typedef struct {
    STM_RSC         Resource;
} ENTRY_EVT_LOG_PROTECTION_EXCEPTION;

typedef struct {
    STM_RSC         Resource;
} ENTRY_EVT_LOG_HANDLED_PROTECTION_EXCEPTION;

typedef struct {
    STM_RSC         Resource;
} ENTRY_EVT_BIOS_ACCESS_UNCLAIMED_RSC;

typedef struct {
    STM_RSC         Resource;
} ENTRY_EVT_MLE_RSC_PROT_GRANTED;

typedef struct {
    STM_RSC         Resource;
} ENTRY_EVT_MLE_RSC_PROT_DENIED;

typedef struct {
    STM_RSC         Resource;
} ENTRY_EVT_MLE_RSC_UNPROT;

typedef struct {
    STM_RSC         Resource;
} ENTRY_EVT_MLE_RSC_UNPROT_ERROR;

typedef struct {
    UINT64    VmcsPhysPointer;
    UINT8     ExpectedDomainType;
    UINT8     DegradedDomainType;
} ENTRY_EVT_MLE_DOMAIN_TYPE_DEGRADED;
```

E.2 Event Logging Flow

The flow chart below describes the algorithm the STM follows when logging events. The Get_Next_Empty algorithm pseudo-code follows this diagram.

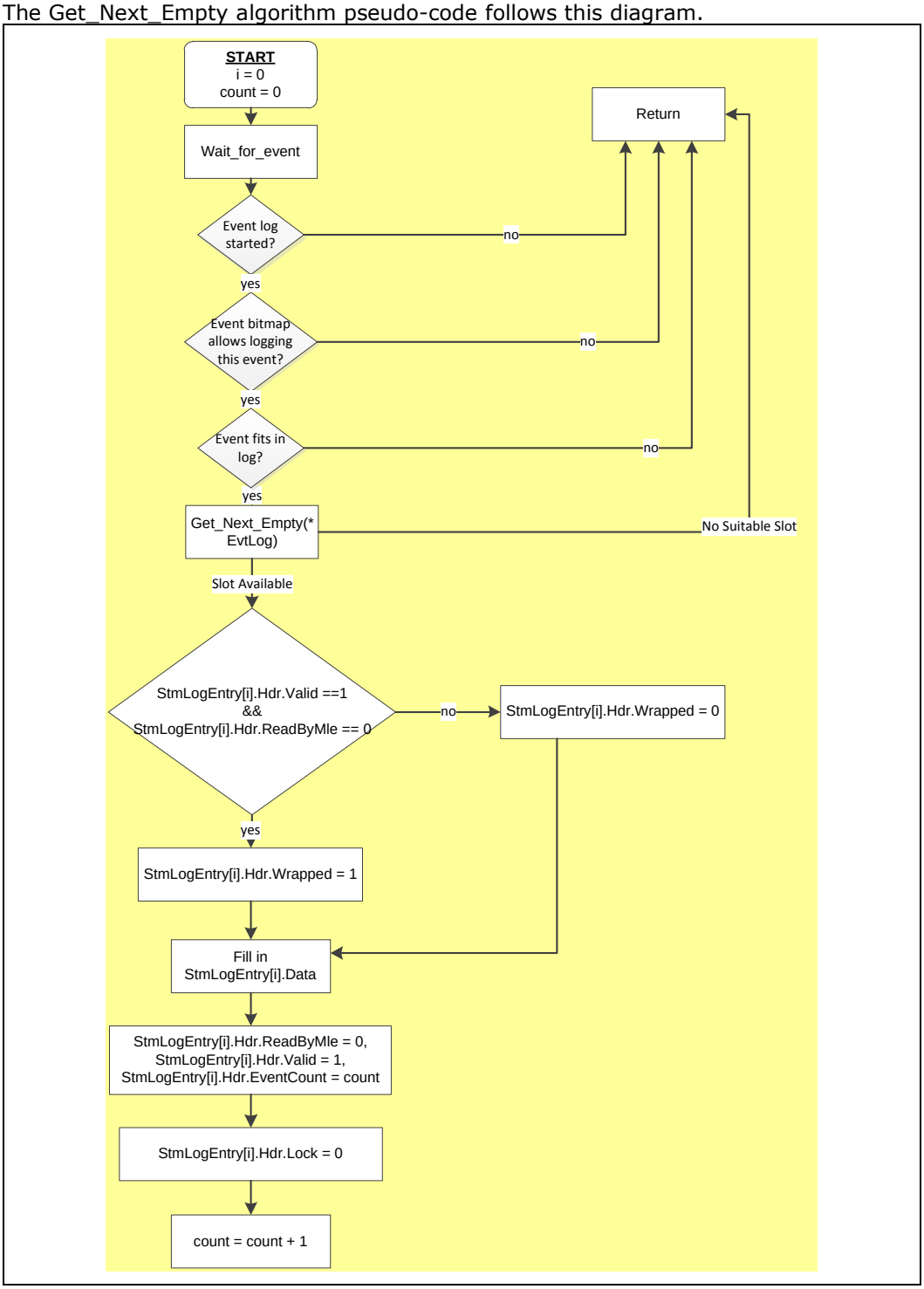


Figure E-1: Logging flowchart

BEGIN - informative content – non-normative

This is an example of one way to find the next slot for a new event

```
STM_LOG_ENTRY *GetNextEmpty(MLE_EVENT_LOG_STRUCTURE *EventLog)
{
  Step 1: Look for an Invalid Entry
  FOR each page in array of event log pages
    FOR each event log entry on the page
       IF successful testing and setting the event's lock THEN
          Check header for valid/invalid
             IF header is invalid, THEN
                Return this log entry.
             ELSE
                Reset the lock and check next event log entry on page
             END IF
       ELSE (unsuccessful test/set on lock)
          Check next event log entry on page
       END IF
    END FOR
  END FOR
  // No invalid entries found, proceed to next step.

  Step 2: Look for entry already read by MLE
  FOR each page in array of event log pages
    FOR each event log entry on the page
       IF successful testing and setting the event log entries lock THEN
          Check header for "read by the MLE"
             IF the entry has been read by the MLE, THEN
```

```
                Return this log entry.
            ELSE
                Reset the lock and check next event log entry on page
            END IF
     ELSE (unsuccessful test/set on lock)
         Check next event log entry on page
     END IF
  END FOR
END FOR
// No entries were already read by MLE, proceed to next step.


Step 3: Overwrite an event not read by MLE
 FOR each page in array of event log pages
   FOR each event log entry on the page
     IF successful testing and setting the event log entries lock THEN
         Check header for event log not wrapped indicator
             IF the entry's log not wrapped indicator is not set THEN
                Return this log entry.
             ELSE
                Reset the lock and check next event log entry on page
             END IF
     ELSE (unsuccessful test/set on lock)
         Check next event log entry on page
     END IF
  END FOR
END FOR
// No entries had log not wrapped indicator, proceed to next step.
Step 4: Take the first unlocked event
FOR each page in array of event log pages
  FOR each event log entry on the page
```

```
        IF successful testing and setting the event log entries lock THEN
            Return this log entry.
        ELSE
           Reset the lock and check next event log entry on page
        END IF
    END FOR
  END FOR
  // No suitable event log entries can be used
  return NULL
}
```

END - informative content – non-normative

§

Appendix F Debugging/ Development Functions

F.1 Overview

This section defines a set of functions that are recommended for development of an STM, the SMI handler or the MLE. **These functions are for the purpose of development and should be removed from production components.**

To assist development, debugging and validation of all components (SMI handler, STM and MLE), the SMI handler should implement the following features.

The SMI handler shall provide a hook to allow the MLE to cause changes to the SMI handler required resources and to cause the SMI handler to access an arbitrary resource. This feature must be implemented such that it exists only in debug versions of the SMI handler (e.g., #ifdef DEBUG type statements). It should be removed from production BIOS implementations. This feature shall be implemented using the SMI_CMD port (i.e., 0xB2H). An MLE application sends these commands to the SMI handler by writing to the SMI_CMD port. Parameters will be passed using CPU registers. The SMI handler developer shall determine the specific command (the data of the out instruction) used. This command instructs a debug SMI handler to perform the following development / debug / validation operations.

NOTE: Access to SMI_CMD should generally be done from an unprotected domain. A different domain type may cause STM to block the input or output registers.

There will be three types of commands. The first command type is used to modify the BIOS resource list. This command can only be executed prior to the first call to ProtectResourceVMCALL() and prior to the first call to GetBiosResourcesVMCALL(). The second command type is used to request that the SMI handler access a particular resource. The third command type is used to load a new STM image into MSEG. This command can only be executed before launching the MLE.

Commands will be differentiated by the value written to the SMI_CMD port and additional function number passed via APM_STS port (0xB3)

Command values will be exposed to the caller via the BIOS extended data from the TXT heap in a manner identical to BIOS services specified in section 4.3.

1. HandleBiosResourcesCmd - respective SMI_CMD value will be displayed at offset ReservedForDebug[0] in the BIOS extended Data in the TXT heap. See **Error! Reference source not found.**.

2. AccessResourcesCmd – respective SMI_CMD value will be displayed at offset ReservedForDebug[1] in the BIOS extended Data in the TXT heap. See **Error! Reference source not found.**.

3. LoadStmCmd - respective SMI_CMD value will be displayed at offset ReservedForDebug[2] in the BIOS extended Data in the TXT heap.

All debug commands will share the following input/output format specification:

12.1.1.1.1 Input registers:

The SMI_CMD port will contain the command value. APM_STS port will contain the function value. The Debug SMI handler will branch based on SMI_CMD:APM_STS 16-bit value to the appropriate handler function.

ECX:EBX will contain the 64-bit physical address of the caller allocated buffer. Buffer content and structure will be function-specific. The buffer must be 4KB page aligned and at least 4KB in size. The STM will return an error if this condition is not met.

12.1.1.1.2 Output registers:

CF = 0, EAX = SMM_SUCCESS: NO error

CF = 1, EAX = Error code – see related definitions later in this section.

F.2 Commands

F.2.1 HandleBiosResourcesCmd

HandleBiosResourcesCmd will have the following functions:

- AddRuntimeResourcesFunc. This function will update the BIOS-provided resource list. It will merge new resources with existing records if possible observing attributes or will append new resources to the existing list if a merge is not possible. In cases when given new resource matches exactly one of the records with the exception of IgnoreResource bit, the record will be updated to the state of IgnoreResource bit of the new resource. This allows the caller to selectively control BIOS resources by asserting of clearing of the IgnoreResource bit on record-by-record basis. This function fails if ProtectResourcesVMCALL() has been successfully called.
- ReadBiosResourcesFunc. This function will read the whole list of BIOS resources into the caller's buffer.
- ReplaceBiosResourcesFunc. This function will instruct the SMI handler to replace the entire existing BIOS Resource list with the one pointed to by the input buffer. Though the same functionality can be achieved with AddRuntimeResourcesFunc function, this function may be simpler to use for basic testing. This function fails if ProtectResourcesVMCALL() has been successfully called.

F.2.1.1 AddRuntimeResourcesFunc

The input buffer must contain the TXT_BIOS_DEBUG structure where the data field must be a list of resource structures ended by the `STM_RSC_END` structure.

```
UINT8 Data[] = RESOURCE List[];
```

The function will use BufferSize value to prevent endless looping and a crash in case of a list not being terminated by STM_RSC_END structure. The function will also make a plausible guess about the validity of the passed-in buffer – if the value of BufferSize field appears to be too large, which is most likely the result of a caller error, it will return SMM_INVALID_BUFFER_SIZE error. The 16KB value selected as maximum buffer size is presumed to be sufficiently large for any real BIOS.

Return values (EAX):

```
SMM_SUCCESS             // No errors

SMM_INVALID_RSC         // Invalid resource structure detected

SMM_INVALID_BUFFER_SIZE // BufferSize value exceeds 16KB

SMM_INVALID_LIST        // Resource list is not terminated

SMM_OUT_OF_MEMORY       // BIOS resource memory exhausted

SMM_AFTER_INIT          // Illegal after ProtectResourcesVMCALL

ERROR_SMM_UNSPECIFIED   // An unspecified error occurred
```

F.2.1.2 ReadBiosResourcesFunc

The input buffer must contain a TXT_BIOS_DEBUG structure where the Data field is empty.

Upon return, the buffer will contain a list of resource structures terminated by STM_RSC_END structure.

```
UINT8 Data[] = RESOURCE List[];
```

In the case of the SMM_BUFFER_TOO_SHORT error, the BufferSize field will be updated with the minimal buffer size needed to read all resources.

Return values (EAX):

```
SMM_SUCCESS             // No errors

SMM_INVALID_BUFFER_SIZE // BufferSize value exceeds 16KB

SMM_BUFFER_TOO_SHORT    // Insufficient buffer size.

ERROR_SMM_UNSPECIFIED   // An unspecified error occurred
```

F.2.1.3 ReplaceBiosResourcesFunc

The semantics of this function are identical to the semantics of the AddRuntimeResourcesFunc function.

F.2.2 AccessResourcesCmd

This command doesn't have sub-functions – the value of APM_STS port will be ignored.

AccessResourcesCmd instructs the SMI handler debug function to access all the resources in the resource list pointed to by ECX:EBX. Only the following resource types are supported: MEM_RANGE, IO_RANGE, MMIO_RANGE, MACHINE_SPECIFIC_REG, PCI_CFG_RANGE. The SMI handler debug function fails if it encounters a resource record that is not one of these. As this is only a debug function, the SMI handler is allowed latitude in how it handles this. It may process and execute each resource record then fail if it encounters an invalid one, or it may scan all resource records first before processing.

Unless the SMI handler debug function encounters an invalid resource type, it must process every resource record. (i.e., it must continue even if the access results in an exception) Upon processing each resource record, the SMI handler debug function will set the record's ReturnStatus field.

This function will interpret the ReturnStatus and IgnoreResource fields of each of resource header in the following way:

- IgnoreResource (note the repurposing of this field) will indicate the requested operation : 1 = write; 0 = read
- ReturnStatus will indicate result of operation : 0 = SMI exception handler was not invoked as a result of operation; 1 = SMI exception handler was invoked.

For each resource, the SMI handler debug function will perform the appropriate access for each element of the resource record. (I.e., if the resource is a memory type and write operation is requested, then the SMI handler debug function will write to every address starting at address pointed to by the Base field and writing Length bytes).

The input buffer must contain a TXT_BIOS_DEBUG structure where its Data field must be a list of resource structures ended by `STM_RSC_END` structure.

```
UINT8 Data[] = RESOURCE List[];
```

Upon return ReturnStatus filed is updated according to result of operation.

Return values (EAX):

```
SMM_SUCCESS             // No errors

SMM_INVALID_RSC         // Invalid resource structure detected

SMM_INVALID_BUFFER_SIZE // BufferSize value exceeds 16KB

SMM_INVALID_LIST        // Resource list is not terminated

ERROR_SMM_UNSPECIFIED   // An unspecified error occurred
```

F.2.3 LoadStmCmd

This command doesn't have sub-functions – the value of the APM_STS port will be ignored.

LoadStmCmd is used to update STM image in MSEG at run time. Its semantic is identical to the semantic of UpdateStmCmd. In addition to the errors returned by UpdateStmCmd, this command will return an error if InitializeProtectionVMCALL() has been successfully called.

Return values (EAX):

```
SMM_AFTER_INIT          // Illegal after InitializeProtectionVMCALL

ERROR_SMM_UNSPECIFIED   // An unspecified error occurred
```

12.1.1.1.3 Related Definitions

```
#define AddRuntimeResourcesFunc   0

#define ReadBiosResourcesFunc     1

#define ReplaceBiosResourcesFunc  2

typedef union {
    STM_RSC_MEM_DESC           ResMem;
    STM_RSC_IO_DESC            IoMem;
    STM_RSC_MMIO_DESC          MmioMem;
    STM_RSC_MSR_DESC           MsrMem;
    STM_RSC_PCI_CFG_DESC       PciCfgMem;
    STM_RSC_TRAPPED_IO_DESC    TrappedIoMem;
    STM_RSC_END                RscEnd;
}   RESOURCE;

typedef struct _{
    UINT32    BufferSize;
    UINT32    Reserved;
    UINT8     Data[];
} TXT_BIOS_DEBUG;

#define SMM_SUCCESS               0x0

#define SMM_INVALID_RSC           0x80020004

#define SMM_INVALID_BUFFER_SIZE   0x80020005

#define SMM_BUFFER_TOO_SHORT      0x80020006

#define SMM_INVALID_LIST          0x80020007

#define SMM_OUT_OF_MEMORY         0x80020008

#define SMM_AFTER_INIT            0x80020009
```