Intel® Trust Domain Extensions (Intel® TDX) Module Base Architecture Specification

348549-001US

September 2021

Notices and Disclaimers

Intel Corporation ("Intel") provides these materials as-is, with no express or implied warranties.

All products, dates, and figures specified are preliminary, based on current expectations, and are subject to change without notice. Intel does not guarantee the availability of these interfaces in any future product. Contact your Intel representative to obtain the latest Intel product specifications and roadmaps.

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

Intel technologies might require enabled hardware, software, or service activation. Some results have been estimated or simulated. Your costs and results might vary.

No product or component can be absolutely secure.

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

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

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice.

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

Table of Contents

SECTION 1: INTRODUCTION AND OVERVIEW 10

1. About this Document 11

1.1. Scope of this Document 11

1.2. Document Organization 11

1.3. Glossary 11

1.4. Notation 14

1.4.1. Requirement and Definition Commitment Levels 15

1.5. References 15

1.5.1. Intel Public Documents 15

1.5.2. Intel TDX Public Documents 15

2. Overview of Intel® Trust Domain Extensions 17

2.1. Intel TDX Module Lifecycle 17

2.1.1. Boot-Time Configuration and Intel TDX Module Loading 17

2.1.2. UPDATED: Intel TDX Module Initialization, Enumeration and Configuration 17

2.2. Guest TD Life Cycle Overview 18

2.2.1. Guest TD Build 18

2.2.2. Guest TD Execution 18

2.2.3. Guest TD Management during its Run-Time 18

2.3. Intel TDX Operation Modes and Transitions 19

2.4. Guest TD Private Memory Protection 20

2.4.1.1. Memory Encryption 20

2.4.2. Address Translation 20

2.5. Guest TD State Protection 21

2.6. Intel TDX I/O Model 21

2.7. Measurement and Attestation 22

2.8. Intel TDX Managed Control Structures 22

2.9. Intel TDX Interface Functions 23

2.9.1. Host-Side (SEAMCALL Leaf) Interface Functions 23

2.9.2. Guest-Side (TDCALL Leaf) Interface Functions 25

3. Software Use Cases 26

3.1. Intel TDX Module Lifecycle 26

3.1.1. UPDATED: Intel TDX Module Platform-Scope Initialization 26

3.1.2. Intel TDX Module Reload and Update 26

3.2. TD Build 27

3.3. TD Run Time 29

3.3.1. Private Memory Management 29

3.3.1.1. Dynamic Page Addition (Shared to Private Conversion) 29

3.3.1.2. Dynamic Page Removal (Private to Shared Conversion) 30

3.3.1.3. Page Promotion (Mapping Merge) 31

3.3.1.4. Page Demotion (Mapping Split) 32

3.3.1.5. GPA Range Unblock 33

3.3.2. Guest TD Execution 33

3.3.2.1. TD VCPU First-Time Invocation 33

3.3.2.2. TD VCPU Entry, Exit on TDG.VP.VMCALL and Re-Entry 34

3.3.2.3. TD VCPU Entry, Exit on Asynchronous Event and Re-Entry 34

3.3.2.4. Guest-Side Functions 35

3.3.2.5. TD VCPU Rescheduling (Migration to Another LP) ... 36

3.4. TD Destruction ... 37

SECTION 2: INTEL TDX MODULE ARCHITECTURE SPECIFICATION ... 39

4. UPDATED: Intel TDX Module Lifecycle: Enumeration, Initialization and Shutdown ... 40

4.1. UPDATED: Overview ... 40

4.1.1. UPDATED: Initialization and Configuration Flow ... 40

4.1.2. UPDATED: Intel TDX Module Lifecycle State Machine ... 41

4.1.3. Platform Compatibility and Configuration Checking ... 41

4.1.3.1. Overview ... 41

4.1.3.2. MSR Sampling and Checks ... 41

4.1.3.3. UPDATED: CPUID Sampling, Checks and Enumeration ... 41

4.1.4. Physical Memory Configuration Overview ... 42

4.1.4.1. Intel TDX ISA Background: Convertible Memory Ranges (CMRs) ... 42

4.1.4.2. TDMRs and PAMT Arrays Configuration ... 42

4.2. Intel TDX Module Initialization Interface ... 44

4.2.1. Global Initialization: TDH.SYS.INIT ... 44

4.2.2. LP-Scope Initialization: TDH.SYS.LP.INIT ... 44

4.2.3. UPDATED: TDX Module Enumeration: TDH.SYS.RD/RDALL and TDH.SYS.INFO ... 44

4.2.4. Global Configuration: TDH.SYS.CONFIG ... 44

4.2.5. Package-Scope Key Configuration: TDH.SYS.KEY.CONFIG ... 45

4.3. TDMR and PAMT Initialization ... 45

4.4. Intel TDX Module Shutdown ... 45

4.4.1. Shutdown Initiated by the Host VMM (as Part of Module Update) ... 45

4.4.2. Shutdown Initiated by a Fatal Error ... 45

5. Key Management ... 46

5.1. Objectives ... 46

5.2. Background: HKID Space Partitioning ... 46

5.3. Key Management Tables ... 47

5.4. Combined Key Management State ... 48

5.5. Key Management Sequences ... 49

5.5.1. Intel TDX Module Initialization: Setting an Ephemeral Key and Reserving an HKID for Intel TDX Data ... 49

5.5.2. TD Creation, Keys Assignment and Configuration ... 49

5.5.3. TD Keys Reclamation, TLB and Cache Flush ... 50

6. UPDATED: TD Non-Memory State (Metadata) and Control Structures ... 51

6.1. Overview ... 51

6.1.1. Opaque vs. Shared Control Structures ... 51

6.1.2. Scope of Control Structures ... 51

6.2. TD-Scope Control Structures ... 51

6.2.1. TDR (Trust Domain Root) ... 52

6.2.2. UPDATED: TDCS (Trust Domain Control Structure) ... 52

6.3. TD VCPU-Scope Control Structures ... 52

6.3.1. Trust Domain Virtual Processor State (TDVPS) ... 52

6.3.1.1. UPDATED: Physical View of TDVPS: TDVPR/TDCX ... 53

6.3.1.2. Logical View of TDVPS ... 53

6.3.2. Non-Protected Control Structures: Shared EPT and VMCS Auxiliary Control Structures ... 54

6.4. UPDATED: TD Non-Memory State (Metadata) Access Functions ... 54

6.5. Concurrency Restrictions and Enforcement ... 54

Section 1: Introduction and Overview

7. UPDATED: TD Life Cycle Management 55
7.1. TD Life Cycle State Machine 55
7.2. UPDATED: OP_STATE: TD Operation Secondary-Level State Machine 55
7.3. TD Creation Sequence 56
7.4. VCPU Creation and Initialization Sequence 57
7.5. TD Teardown Sequence 58

8. Physical Memory Management 59
8.1. UPDATED: Trust Domain Memory Regions (TDMRs) and Physical Address Metadata Tables (PAMTs) 59
8.2. UPDATED: TDMR Details 59
8.3. PAMT Details 59
8.3.1. PAMT Entry 59
8.3.2. PAMT Blocks and PAMT Arrays 60
8.3.3. PAMT Page Types 61
8.3.4. PAMT Hierarchy 61
8.4. Adding Physical Pages 62
8.4.1. Preventing Cache Line Aliasing 62
8.4.2. Adding Pages not Mapped to the Guest TD 62
8.4.3. Adding Pages and Mapping to the Guest TD’s GPA 62
8.5. Reclaiming Physical Pages 62
8.5.1. Reclaiming Pages not Mapped to the Guest TD 62
8.5.2. Reclaiming TD Pages in TD_TEARDOWN State 62
8.5.3. Reclaiming Physical Pages as Part of TD Private Memory Management 62

9. TD Private Memory Management 64
9.1. Overview 64
9.2. Secure EPT Entry 65
9.2.1. UPDATED: Overview 65
9.2.2. UPDATED: SEPT Entry State Diagrams 65
9.3. Secure EPT Walk 66
9.4. Secure EPT Induced TD Exits 67
9.5. Secure EPT Induced Exceptions 67
9.6. UPDATED: Secure EPT Concurrency 67
9.7. Introduction to TLB Tracking 67
9.8. Secure EPT Build and Update: TDH.MEM.SEPT.ADD 68
9.9. Adding TD Private Pages during TD Build Time: TDH.MEM.PAGE.ADD 69
9.10. Dynamically Adding TD Private Pages 70
9.10.1. Overview 70
9.10.2. Page Addition by the Host VMM: TDH.MEM.PAGE.AUG 70
9.10.3. Page Acceptance by the Guest TD: TDG.MEM.PAGE.ACCEPT 71
9.10.3.1. Description 71
9.10.3.2. TDG.MEM.PAGE.ACCEPT Concurrency 73
9.11. Page Merge: TDH.MEM.PAGE.PROMOTE 73
9.12. Page Split: TDH.MEM.PAGE.DEMOTE 75
9.13. Relocating TD Private Pages: TDH.MEM.PAGE.RELOCATE 76
9.14. Removing TD Private Pages: TDH.MEM.PAGE.REMOVE 77
9.15. Removing a Secure EPT Page: TDH.MEM.SEPT.REMOVE 77

9.16. Unblocking a GPA Range: TDH.MEM.RANGE.UNBLOCK 78

10. TD VCPU 79

10.1. VCPU Transitions 79
10.1.1. Initial TD Entry, Asynchronous TD Exit and Subsequent TD Entry 79
10.1.2. Synchronous TD Exit and Subsequent TD Entry 80
10.1.3. UPDATED: VCPU Activity State Machine 80

10.2. TD VCPU TLB Address Space Identifier (ASID) 82
10.2.1. TD ASID Components 82
10.2.2. INVEPT by the Host VMM for Managing the Shared EPT 82

10.3. VCPU-to-LP Association 82
10.3.1. Non-Coherent Caching 82
10.3.2. Intel TDX Functions for VCPU-LP Association and Dis-Association 83
10.3.3. Performance Considerations 83

11. UPDATED: CPU Virtualization (Non-Root Mode Operation) 84

11.1. Initial State 84
11.1.1. Overview 84
11.1.2. Initial State of Guest TD GPRs 84
11.1.3. Initial State of CRs 84
11.1.4. Initial State of Segment Registers 85
11.1.5. Initial State of MSRs 85

11.2. UPDATED: Guest TD Run Time Environment Enumeration 85

11.3. CPU Mode Restrictions 86

11.4. Instructions Restrictions 86
11.4.1. Instructions that Cause a #UD Unconditionally 86
11.4.2. Instructions that Cause a #VE Unconditionally 86
11.4.3. Instructions that Cause a #UD or #VE Depending on Feature Enabling 87
11.4.4. Other Cases 87

11.5. Extended Feature Set 87
11.5.1. Allowed Extended Features Control 87
11.5.2. Extended State Isolation 87
11.5.3. Extended Features Execution Control 87

11.6. UPDATED: CR Handling 89
11.6.1. CR0 89
11.6.2. UPDATED: CR4 90

11.7. MSR Handling 90
11.7.1. Overview 90

11.8. UPDATED: CPUID Virtualization 90
11.8.1. UPDATED: CPUID Configuration by the Host VMM 90
11.8.2. Guest TD Control of CPUID Virtualization 91
11.8.2.1. Unconditional #VE for all CPUID Leaves and Sub-Leaves 91
11.8.2.2. NEW: Leaf/Sub-leaf Specific Control 91
11.8.3. NEW: CPUID configuration & Checks at Guest TD Migration 92

11.9. Interrupt Handling and APIC Virtualization 92
11.9.1. Virtual APIC Mode 92
11.9.2. Virtual APIC Access by Guest TD 92
11.9.3. UPDATED: Posted Interrupts 93
11.9.4. Pending Virtual Interrupt Delivery Indication 94
11.9.5. Cross-TD-VCPU IPI 94
11.9.6. Virtual NMI Injection 94

11.10. Virtualization Exception (#VE) 94
11.10.1. Virtualization Exception Information 95

11.10.2. #VE Injection by the CPU due to EPT Violations 95
11.10.3. #VE Injected by the Intel TDX Module 96

11.11. Secure and Shared Extended Page Tables (EPTs) 96
11.11.1. GPAW-Relate EPT Violations 96
11.11.2. EPT Violation Mutated into #VE 96

11.12. Prevention of TD-Induced Denial of Service 96
11.12.1. Bus Lock Detection by the TD OS 96
11.12.2. Impact of MSR_TEST_CTRL (MSR 0x33) 97
11.12.3. Bus Lock TD Exit 97
11.12.4. Notification TD Exit 97

11.13. Time Stamp Counter (TSC) 97
11.13.1. TSC Virtualization 97

11.14. Supervisor Protection Keys (PKS) 98

11.15. Intel® Total Memory Encryption (Intel® TME) and Multi-Key Total Memory Encryption (MKTME) 98
11.15.1. TME Virtualization 98
11.15.2. MKTME Virtualization 98

11.16. Other Changes in TDX Non-Root Mode 98
11.16.1. Tasking 98
11.16.2. PAUSE-Loop Exiting 99

12. Measurement and Attestation 100

12.1. TD Measurement 100
12.1.1. MRTD: Build-Time Measurement Register 100
12.1.2. RTMR: Run-Time Measurement Registers 100

12.2. UPDATED: TD Measurement Reporting 100

12.3. TD Measurement Quoting 101
12.3.1. Intel SGX-Based Attestation 101

12.4. Quote Signing Key 102

12.5. TCB Recovery 102

13. New: Service TDs 104

13.1. Overview 104

13.2. Service TD Binding 104
13.2.1. Service TD Binding Table in the Target TD's TDCS 104
13.2.2. SERVTD_BINDING_STATE: Service TD Binding State 105
13.2.3. SERVTD_TYPE: Service TD Binding Type 105
13.2.4. SERVTD_ATTR: Service TD Binding Attributes 105
13.2.4.1. INSTANCE_BINDING: Class vs. Instance Binding 105
13.2.4.2. MIGRATABLE_BINDING: Binding Migratability 105
13.2.4.3. IGNORE_TDINFO: TDINFO Component Filtering 106
13.2.5. SERVTD_UUID: Service TD Instance Identifier 106
13.2.6. Service TD's Binding SERVTD_INFO_HASH Calculation 106
13.2.7. Target TD's SERVTD_HASH Calculation 106
13.2.8. TDH.SERVTD.PREBIND: Pre-Binding a Service TD 107
13.2.9. TDH.SERVTD.BIND: Binding a Service TD 107
13.2.10. Binding Handle 108

13.3. Target TD Metadata Access by a Service TD 108
13.3.1. TDG.SERVTD.RD/WR*: Metadata Read/Write Interface Functions 108
13.3.2. Metadata Access Error Handling 108
13.3.3. Cross-TD Concurrency Handling: Maintaining Host-Side Priority 109
13.3.3.1. Problem Description 109
13.3.3.2. Solution 109

14. I/O Support 110

14.1. Overview 110

14.2. Paravirtualized I/O 110

14.3. MMIO Emulation and Emulated Devices 110

14.4. Direct Device Assignment (DDA) and SRIOV 110

14.5. IOMMU – DMA Remapping 110

14.6. Shared Virtual Memory (SVM) 111

15. Debug and Profiling Architecture 112

15.1. On-TD Debug 112

15.1.1. Overview 112

15.1.2. Generic Debug Handling 112

15.1.2.1. Context Switch 112

15.1.2.2. IA32_DEBUGCTL MSR Virtualization 112

15.1.3. Debug Feature-Specific Handling 113

15.2. On-TD Performance Monitoring 114

15.2.1. Overview 114

15.2.2. Performance Monitoring MSRs 114

15.2.3. Performance Monitoring Interrupts (PMIs) 115

15.3. Off-TD Debug 115

15.3.1. Modifying Debuggable TD's State, Controls and Memory 116

15.3.2. Preventing Guest TD Corruption of DRs 116

15.4. Uncore Performance Monitoring Interrupts (Uncore PMIs) 116

16. Machine Check Handling 117

16.1. Machine Check Architecture Background 117

16.2. Security Objectives for Machine Check Handling 117

16.3. Corrected Machine Check Interrupt (CMCI) 118

16.4. Handling #MC during Guest TD Operation 118

16.5. Virtualization of Machine Check Capabilities and Controls to the Guest TD 118

16.6. Handling #MC during Intel TDX Module Operation 118

16.7. Reclaiming Memory after a Machine Check Event 118

17. Side Channel Attack Mitigation Mechanisms 120

17.1. Checking CPU Vulnerabilities to Known Attacks 120

17.2. Branch Prediction Side Channel Attacks Mitigation Mechanisms 120

17.3. Single-Step and Zero-Step Attacks Mitigation Mechanisms 120

17.3.1. Description 120

17.3.2. Host VMM Expected Behavior 120

17.3.3. Guest TD Interface and Expected Guest TD Operation 121

18. UPDATED: General Aspects of the Intel TDX Interface Functions 122

18.1. Concurrency Restrictions and Enforcement 122

18.1.1. Explicit Concurrency Restrictions 122

18.1.2. Implicit Concurrency Restrictions 122

18.1.3. Transactions 123

18.1.4. NEW: Concurrency Restrictions with Host Priority 123

18.1.4.1. Host-Side (Host VMM) Operation 123

18.1.4.2. Guest-Side (Service TD) Operation 123

18.2. Memory and Resource Operands Access ... 123
18.2.1. Overview ... 123
18.2.1.1. Access Semantics ... 124
18.2.1.2. Explicit vs. Implicit Access ... 124
18.2.1.3. Memory Operand Address Specification ... 124
18.2.1.4. Memory Type ... 125
18.2.1.5. Actual Memory Access vs. Memory Reference ... 125
18.2.1.6. Summary Table ... 125
18.3. UPDATED: Register Operands and CPU State Convention ... 126
18.3.1. Overview: Regular vs. Transition Leaf Functions ... 126
18.3.2. NEW: Interface Function Leaf and Version Numbers ... 126
18.3.3. UPDATED: Interface Function Completion Status ... 126
18.3.3.1. Least Detailed Level: Success/Warning/Error ... 126
18.3.3.2. UPDATED: Medium Detailed Level: Class, Recoverability and Fatality ... 127
18.3.3.3. UPDATED: Most Detailed Level ... 127
18.3.4. CPU State Preservation Convention ... 128
18.3.4.1. TDH.VP.ENTER ... 128
18.3.4.2. Other Interface Functions ... 128
18.3.5. Transition Cases: TD Entry and Exit ... 128
18.3.5.1. TD Entry: TDH.VP.ENTER ... 128
18.3.5.2. TD Synchronous Exit: TDG.VP.VMCALL ... 128
18.4. UPDATED: Metadata Access Interface ... 129
18.4.1. Introduction ... 129
18.4.2. Metadata Fields and Elements ... 129
18.4.3. Arrays of Metadata Fields ... 130
18.4.4. NEW: Metadata Field Sequences ... 130
18.4.5. NEW: Metadata Lists ... 132
18.5. Latency of the Intel TDX Interface Functions ... 132

SECTION 1:
INTRODUCTION AND OVERVIEW

1. About this Document

1.1. Scope of this Document

This document describes the architecture of the Intel® Trust Domain Extensions (Intel® TDX) module, implemented using the Intel TDX Instruction Set Architecture (ISA) extensions, for confidential execution of Trust Domains in an untrusted hosted cloud environment.

This document is part of the **TDX Module Architecture Specification Set**, which includes the following documents:

Table 1.1: TDX Module Architecture Specification Set

Document Name	Reference	Description
TDX Module **Base Architecture Specification**	[TDX Module Spec]	Base TDX module architecture overview and specification, covering key management, TD lifecycle management, memory management, virtualization, measurement and attestation, service TDs, debug aspects etc.
TDX Module **TD Migration Architecture Specification**	[TD Migration Spec]	Architecture overview and specification for TD migration
TDX Module **ABI Reference Specification**	[TDX Module ABI]	Detailed TDX module Application Binary Interface (ABI) reference specification, covering the entire TDX module architecture

This document is a work in progress and is subject to change based on customer feedback and internal analysis. This document does not imply any product commitment from Intel to anything in terms of features and/or behaviors.

Note: The contents of this document are accurate to the best of Intel's knowledge as of the date of publication, though Intel does not represent that such information will remain as described indefinitely in light of future research and design implementations. Intel does not commit to update this document in real time when such changes occur.

1.2. Document Organization

The document has the following main sections:

- Section 1 contains an introduction to the document and an overview of the Intel TDX module.
- Section 2 contains the Intel TDX module architecture specification.

1.3. Glossary

Table 1.2: Intel TDX Glossary

Acronym	Full Name	New for TDX	Description
ABI	**Application Binary Interface**	No	A programming interface defined at the binary level (i.e., instruction opcode and CPU registers). The Intel TDX module interface is specified as an ABI.
ACM	**Authenticated Code Module**	No	A code module that is designed to be loaded, verified and executed by the CPU in on-chip memory (CRAM).
N/A	**Accessible (Memory)**	No	Memory whose content is readable and/or writeable (e.g., TD private memory is accessible to the guest TD).

Acronym	Full Name	New for TDX	Description
N/A	**Addressable (Memory)**	No	Memory that can be referred to by its address. The content of addressable memory might not necessarily be accessible (e.g., TDCS is not accessible to the host VMM).
CMR	**Convertible Memory Range**	Yes	A range of physical memory configured by BIOS and verified by MCHECK. MCHECK verification is intended to help ensure that a CMR may be used to hold TDX memory pages encrypted with a private HKID.
N/A	**Enlightened OS**	No	A TD OS is considered enlightened if it is aware that it is running as a TD (see Paravirtualization).
EPxE	**Extended Paging Structures Cache**	No	The CPU's cache of EPT intermediate translations (as opposed to TLB, which caches full LA or GPA to HPA translations).
GPA	**Guest Physical Address**	No	An address viewed as a physical address, from a guest VM's point of view. A GPA is subject to further translation (by EPT) to produce an HPA.
N/A	**Hidden**	No	A resource or a data structure that is not directly addressable by software (except the Intel TDX module).
HKID	**Host Key ID**	Yes	When MKTME is activated, HKID is a key identifier for an encryption key used by one or more memory controllers on the platform.
N/A	**Host VMM**	Yes	The VMM that serves as a host to guest TDs. The term "host" is used to differentiate between the "host VMM" and future VMMs that may be nested within TDs.
HPA	**Host Physical Address**	No	A physical address at the host VMM level. This is the actual physical address used by the hardware (e.g., caches). See also PA.
KET	**Key Encryption Table**	Yes	A table held by each MKTME encryption engine, intended for holding encryption key information, indexed by HKID.
KOT	**Key Ownership Table**	Yes	An internal, hidden table held by the Intel TDX module, intended for controlling the assignment of HKIDs to TDs.
MBZ	**Must Be Zero**	No	Normally used to indicate that reserved fields must contain 0.
MKTME	**Multi-Key TME**	No	This SoC capability adds support to the TME to allow software to use one or more separate keys for encryption of volatile or persistent memory encryption. When used with TDX, it can provide confidentiality via separate keys for memory used by TDs. MKTME can be used with and without TDX extensions.[1]
MRTD	**Measurement of Trust Domain**	Yes	The SHA-384 measurement of a TD accumulated during TD build.
NP-SEAMLDR	**Non-Persistent SEAM Loader**	Yes	An ACM intended to load an Intel P-SEAMLDR module into the SEAM range.
P-SEAMLDR	**Persistent SEAM Loader**	Yes	A SEAM module intended to install (load or update) Intel TDX modules into SEAM range.

[1] In this document, the term "MK-TME" is used to mean both the feature and the encryption engine itself.

Acronym	Full Name	New for TDX	Description
PA	**Physical Address**	No	The physical address used by the hardware (e.g., caches). See also HPA.
PAMT	**Physical Address Metadata Table**	Yes	An internal, hidden data structure used by the Intel TDX module, which is intended to hold the metadata of physical pages.
PV	**Para-Virtualization**	No	A virtualization technique where the VM can be aware it is being virtualized (as opposed to running directly on hardware).
RTMR	**Run-Time Measurement Register**	Yes	A SHA-384 measurement register that can be updated during TD run-time.
SEAM	**Secure Arbitration Mode**	Yes	See TDX ISA.
SEAMRR	**SEAM Range Register**	Yes	A range register used by the BIOS to help configure the SEAM memory range, where the Intel TDX module is loaded and executed.
Service TD	**Service TD**	Yes	A Trust Domain (TD) VM used to provide a dedicated service/utility. Extends the TCB of the tenant TD it provides the service to. Migration TD (MigTD) is an example Service TD.
SEPT	**Secure EPT**	Yes	An Extended Page Table for GPA-to-HPA translation of TD private HPA. A Secure EPT is designed to be encrypted with the TD's ephemeral private key. SEPT pages are allocated by the host VMM via Intel TDX functions, but their content is intended to be hidden and is not architectural.
Intel® SGX	**Intel® Software Guard Extensions**	No	An Intel CPU mode and ISA extensions that support operation and management of Intel® SGX enclaves.
SoC	**System on Chip**	No	A whole system, including cores, uncore, interconnects etc., packaged as a single device.
SPA	**System Physical Address**	No	The physical address used by the hardware (e.g., caches). See also HPA.
TD	**Trust Domain**	Yes	Trust Domains (TDs) are designed to be hardware isolated Virtual Machines (VMs) deployed using Intel® Trust Domain Extensions (Intel® TDX).
TD OS	**Trust Domain Operating System**	Yes	The guest operating system that runs in a TD.
TD VM	**TD Virtual Machine**	Yes	Same as TD
N/A	**TD Private Memory (Access)**	Yes	TD Private Memory is designed to hold TD private content, encrypted by the CPU using the TD ephemeral key.

Acronym	Full Name	New for TDX	Description
N/A	**TD Shared Memory (Access)**	Yes	TD Shared memory is designed to hold content accessible to the TD and the host software (and/or other TDs). TD shared memory may be encrypted using MKTME keys managed by the VMM.
TDCS	**Trust Domain Control Structure**	Yes	Multi-page control structure for a TD. TDCS pages are allocated by the host VMM via Intel TDX functions, but their content is intended to be non-architectural and not directly accessible to software.
TDCX	**Trust Domain Control Extension**	Yes	4KB physical pages that are intended to hold parts of a multi-page control structure.
TDR	**Trust Domain Root**	Yes	The root control structure for a TD. The TDR page is allocated by the host VMM via Intel TDX functions, but its content is intended to be non-architectural and not directly accessible to software.
TDMR	**Trust Domain Memory Range**	Yes	A range of memory, configured by the host VMM, that is covered by PAMT and is intended to hold TD private memory and TD control structures.
TDVPS	**Trust Domain Virtual Processor State**	Yes	A multi-page structure for holding a TD Virtual CPU (VCPU) state. TDVPS pages are allocated by the host VMM via Intel TDX functions, but their content is intended to be non-architectural and not directly accessible to software.
TDVPR	**Trust Domain Virtual Processor Root**	Yes	A 4KB physical page that is intended to be the root (first) page of a TDVPS.
Intel® TDX	**Intel® Trust Domain Extensions**	Yes	An architecture, based on the TDX Instruction Set Architecture (ISA) extensions and the Intel TDX module, which supports operation and management of Trust Domains.
TDX ISA	**Intel® TDX Instruction Set Architecture**	Yes	Intel CPU Instruction Set Architecture (ISA) extensions that support the Intel TDX module: an isolated software module that facilitates the operation and management of Trust Domains.
TME	**Intel® Total Memory Encryption**	No	A memory encryption/decryption engine using an ephemeral platform key designed to encrypt memory contents exposed externally from the SoC.
XFAM	**Extended Features Allowed Mask**	Yes	A mask of CPU extended features (in XCR0 format) that the TD is allowed to use.

1.4. Notation

This section describes the notation used in this document.

1.4.1. Requirement and Definition Commitment Levels

When specifying requirements or definitions, the level of commitment is specified following the convention of RFC 2119: Key words for use in RFCs to indicate Requirement Levels, as described in the following table:

Table 1.3: Requirement and Definition Commitment Levels

Keyword	Description
Must	“**Must**”, "**Required**" or "**Shall**" means that the definition is an absolute requirement of the specification.
Must Not	“**Must Not**” or "**Shall Not**" means that the definition is an absolute prohibition of the specification.
Should	“**Should**”, or the adjective "**Recommended**", means that there may exist valid reasons in particular circumstances to ignore a particular item, but the full implications must be understood and carefully weighed before choosing a different course.
Should Not	“**Should Not**”, or the phrase "**Not Recommended**" means that there may exist valid reasons in particular circumstances when the particular behavior is acceptable or even useful, but the full implications should be understood, and the case must be carefully weighed before implementing any behavior described with this label.
May	“**May**”, or the adjective "**Optional**", means that an item is discretionary. An implementation may choose to include the item, while another may omit the same item, because of various reasons.

1.5. References

1.5.1. Intel Public Documents

Table 1.4: Intel Public Documents

Reference	Document	Version & Date
Intel SDM	Intel® 64 and IA-32 Architectures Software Developer’s Manual	325462-072US, May 2020
ISA Extensions	Intel® Architecture Instruction Set Extensions and Future Features Programming Reference	319433-040, June 2020

1.5.2. Intel TDX Public Documents

Table 1.5: Intel TDX Public Documents

Reference	Document	Version & Date
TDX Whitepaper	Intel Trust Domain Extensions Whitepaper	August 2020
Intel TDX Spec	Intel® Architecture Trust Domain Extensions (TDX) Specification	Rev. 1.0, August 2020
MKTMEi Spec	Intel® Architecture Memory Integrity Specification	Rev. 1.0, March 2020
TDX Module Spec	Intel TDX Module 1.5 Base Architecture Specification	348549-001US September 2021

Reference	Document	Version & Date
TD Migration Spec	Intel TDX Module 1.5 TD Migration Architecture Specification	348550-001US September 2021
TDX Module ABI	Intel TDX Module 1.5 ABI Reference Specification	348551-001US September 2021
GHCI Spec	Intel TDX Guest-Hypervisor Communication Interface Version 1.5	348552-001US September 2021

2. Overview of Intel® Trust Domain Extensions

Intel® Trust Domain Extensions (Intel® TDX) refers to an Intel technology that extends Virtual Machines Extensions (VMX) and Multi-Key Total Memory Encryption (MKTME) with a new kind of virtual machine guest called a **Trust Domain (TD)**. A TD runs in a CPU mode that is designed to protect the confidentiality of its memory contents and its CPU state from any other software, including the hosting Virtual Machine Monitor (VMM), unless explicitly shared by the TD itself.

The TDX solution is built using a combination of Intel® Virtual Machine Extensions (VMX) and Multi-Key Total Memory Encryption (MK-TME), as extended by the **Intel® Trust Domain Extensions Instruction Set Architecture (Intel TDX ISA)**. An attested software module called the **Intel TDX module** is designed to implement the TDX architecture.

The platform is managed by a TDX-aware **host VMM**. As shown in Figure 2.1 below, a host VMM can launch and manage both guest TDs and legacy guest VMs. The host VMM maintains all legacy functionality from the legacy VMs' perspective; it is restricted only with regard to the TDs it manages.

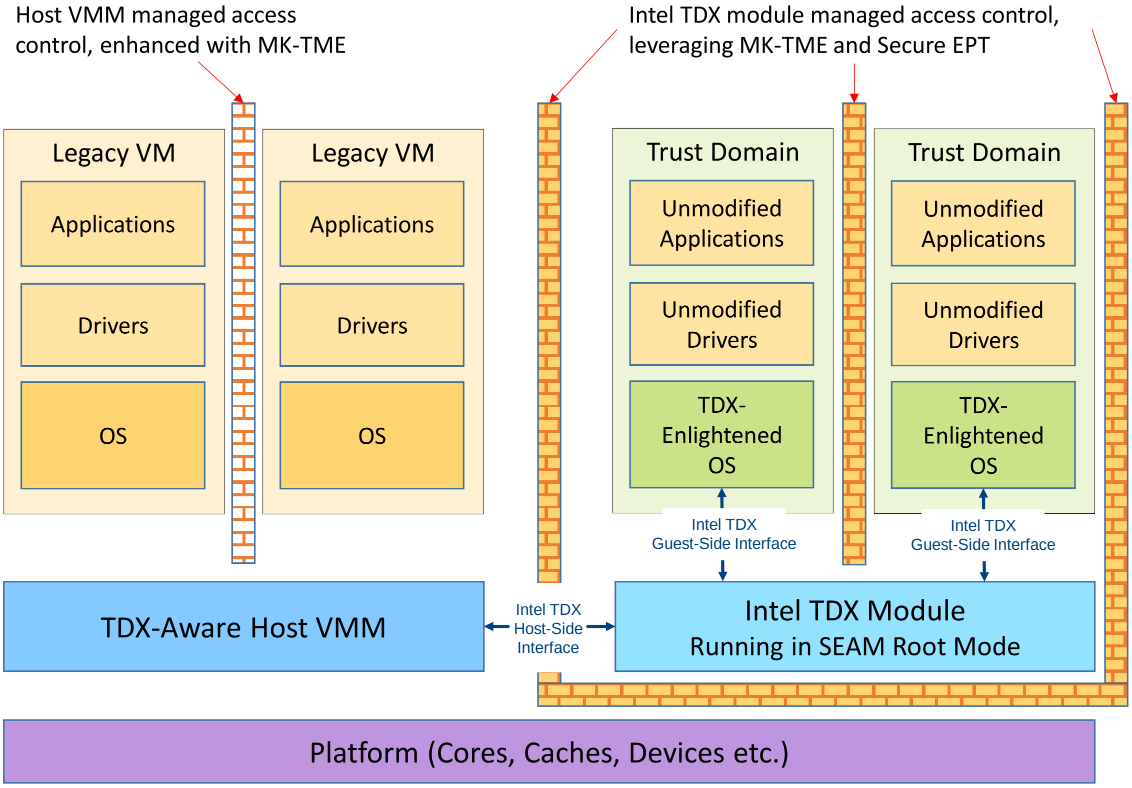

Host VMM managed access control, enhanced with MK-TME
Intel TDX module managed access control, leveraging MK-TME and Secure EPT
Legacy VM
Applications
Drivers
OS
Legacy VM
Applications
Drivers
OS
Trust Domain
Unmodified Applications
Unmodified Drivers
TDX-Enlightened OS
Trust Domain
Unmodified Applications
Unmodified Drivers
TDX-Enlightened OS
Intel TDX Guest-Side Interface
Intel TDX Guest-Side Interface
TDX-Aware Host VMM
Intel TDX Host-Side Interface
Intel TDX Module Running in SEAM Root Mode
Platform (Cores, Caches, Devices etc.)

Figure 2.1: Intel® Trust Domain Extension Components Overview

2.1. Intel TDX Module Lifecycle

2.1.1. Boot-Time Configuration and Intel TDX Module Loading

1. BIOS should configure the SEAMRR registers and prepares a table of **Convertible Memory Regions (CMRs)** – memory regions that can hold TD-private memory pages.
2. BIOS should then initiate MCHECK (as part of a µCode patch load) by WRMSR(0x79). MCHECK is designed to check the correct configuration of SEAMRR and CMRs and store the information in a well-known location in SEAMRR.
3. The host VMM can then load the Intel TDX module using the Persistent SEAMLDR module.

2.1.2. UPDATED: Intel TDX Module Initialization, Enumeration and Configuration

1. After loading the Intel TDX module, the host VMM should call the TDH.SYS.INIT function to globally initialize the module.
2. The host VMM should then call the TDH.SYS.LP.INIT function on each logical processor. TDH.SYS.LP.INIT is intended to initialize the module within the scope of the Logical Processor (LP).
3. The host VMM should then call the TDH.SYS.RD/RDALL or TDH.SYS.INFO function to enumerate the Intel TDX module functionality and parameters, and retrieve the trusted platform topology and CMR information as previously checked by MCHECK.

4. Based on the above, the host VMM should then decide on a set of **Trust Domain Memory Regions (TDMRs)**. TDMR is a region of convertible memory that may contain some reserved sub-regions.
5. The host VMM should then call the TDH.SYS.CONFIG function and pass TDMR information with other configuration information. TDH.SYS.CONFIG is intended to check the configuration information vs. the Intel TDX module's trusted internal data.
6. The host VMM should then call the TDH.SYS.KEY.CONFIG function per package. TDH.SYS.KEY.CONFIG is intended to configure a CPU-generated random key that is used as the Intel TDX module's global private key.
7. The host VMM should then use the TDH.SYS.TDMR.INIT function to initialize the TDMRs and their associated control structures.

The Intel TDX module lifecycle is detailed in Chapter 4.

2.2. Guest TD Life Cycle Overview

2.2.1. Guest TD Build

The host VMM can create a new guest TD by allocating and initializing a TD Root (TDR) control structure using the TDH.MNG.CREATE function. As an input to TDH.MNG.CREATE, the host VMM assigns the TD with a memory protection key identifier, also known as a Host Key ID (HKID). The HKID can be used by the CPU to tag memory accesses done by the TD and by the multi-key total memory encryption engines (MKTMEs) to select the encryption/decryption keys – the keys themselves are designed to not be exposed to the host VMM. The VMM should then program the HKID and encryption key into the MKTME encryption engines using the TDH.MNG.KEY.CONFIG function on each package.

Once the TD is assigned a key, the host VMM can build the TD Control Structure (TDCS) by adding control structure pages, using the TDH.MNG.ADDCX function, and initialize using the TDH.MNG.INIT function. It can then build the Secure EPT tree using the TDH.MEM.SEPT.ADD function and add the initial set of TD-private pages using the TDH.MEM.PAGE.ADD function. These pages typically contain Virtual BIOS code and data along with some clear pages for stacks and heap. Most of the guest TD code and data is dynamically loaded at a later stage. The guest TD can extend run-time measurement registers, designed to be securely maintained by the Intel TDX module, for the added contents using the TDH.MR.EXTEND function.

The host VMM can then create and initialize TD Virtual CPUs (VCPUs). After creating each VCPU using the TDH.VP.CREATE function, the VMM allocates a set of pages to hold the VCPU state (in a structure called TDVPS) using the TDH.VP.ADDCX function. The host VMM can then initialize the VCPU using the TDH.VP.INIT function.

After the initial set of pages is added and extended, the VMM can finalize the TD measurement using the TDH.MR.FINALIZE function.

2.2.2. Guest TD Execution

The host VMM may enter the TD (launch the TD for the first time, or resume a previously intercepted TD execution) using the TDH.VP.ENTER function. The Intel TDX module is designed to load CPU state from the TDVPS structure and perform VM entry to go into TDX non-root mode.

When TD exit is triggered, the Intel TDX module is designed to save CPU state into the TDVPS structure, load the CPU state saved on TD entry, and switch back to TDX root mode (SEAMRET) at the instruction following SEAMCALL. The VMM can then inspect the TD exit information in General Purpose Registers (GPRs).

2.2.3. Guest TD Management during its Run-Time

During TD lifetime, the VMM might need to dynamically control the TD and manage the resources assigned to it. The Intel TDX module provides the VMM with functions to support scenarios such as:

- Adding and removing TD pages.
- Changing page mapping sizes.
- Reclaiming the HKIDs from a TD, and assigning them to another TD.
- Destroying an existing TD.

2.3. Intel TDX Operation Modes and Transitions

The Intel TDX module is designed to provide two main new **logical** modes of operation built upon the new SEAM root and non-root CPU modes added to the Intel VMX architecture: TDX Root Mode, and TDX Non-Root Mode. Figure 2.2 below shows the Intel TDX logical modes and transitions (in red) on top of the CPU architectural modes.

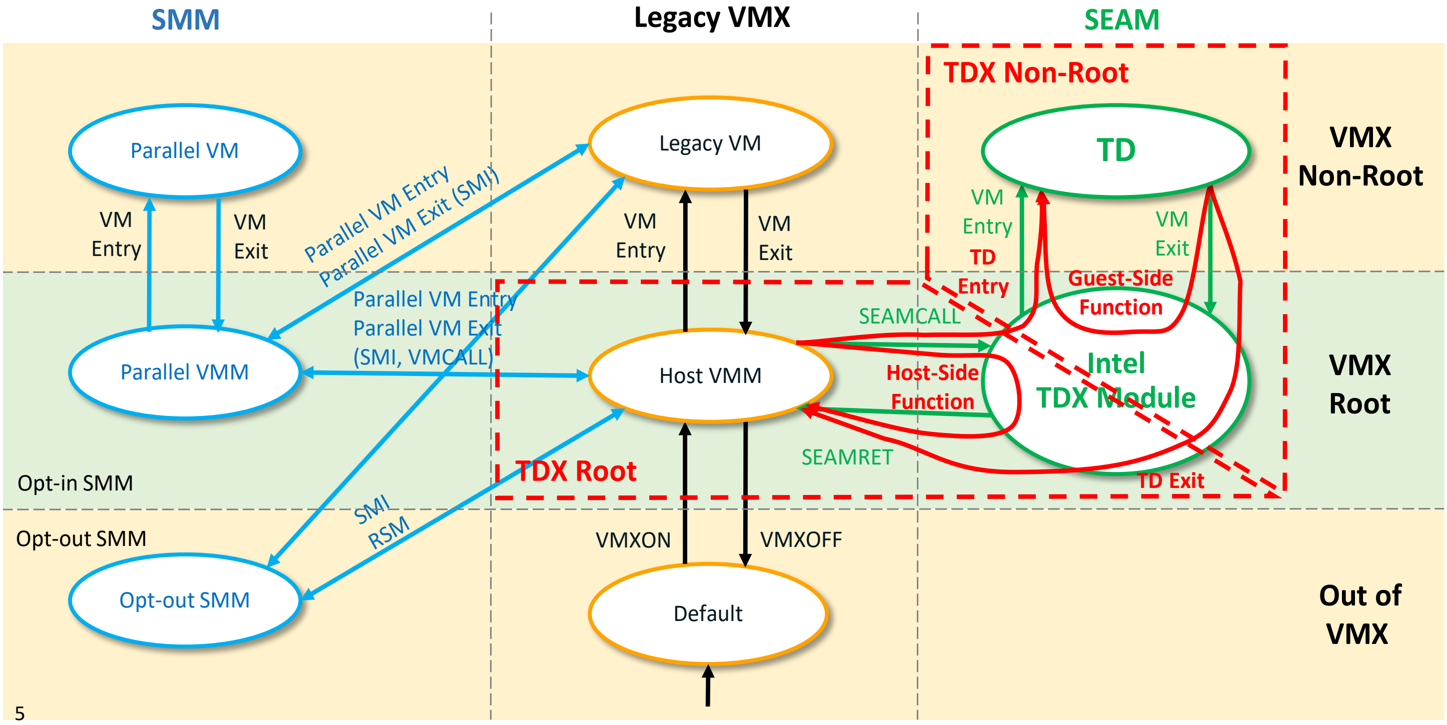


Figure 2.2: Overview of Intel TDX Modes & Transitions based on VMX and SEAM Modes and Transitions

The following table adds more details.

Table 2.1: Overview of Intel TDX Modes

Intel TDX Logical Mode	Intel VMX Mode	SEAM Mode	Description
TDX Root	VMX Root	Non-SEAM (mostly), SEAM (during host-side Intel TDX functions execution)	TDX root mode is mostly identical to the legacy VMX root operation mode. It is generally used for host VMM operation. Host-side Intel TDX functions, triggered by SEAMCALL, are provided by the Intel TDX module. Logically, host-side functions run in TDX root mode, though the CPU's SEAM mode is on.

Intel TDX Logical Mode	Intel VMX Mode	SEAM Mode	Description
TDX Non-Root	VMX Non-Root (mostly), VMX Root (during guest-side Intel TDX flows execution)	SEAM	TDX non-root mode is used for TD guest operation. TDX non-root operation is similar to legacy VMX non-root operation, with changes and restrictions to better assure that no other software or hardware has direct visibility of the TD memory and state. The changes in TDX non-root mode vs. legacy VMX non-root operation are implemented by: • The CPU running in SEAM non-root mode. This modifies the address translation to support Secure EPT and usage of private HKIDs, and it also modifies the VMX operation (entry, exit, etc.). • The Intel TDX module, acting as the root VMM for the guest TD, using VMX and SEAM to virtualize the CPU behavior and emulate the required TDX non-root behavior. Guest-side Intel TDX flows, triggered by a VM Exit, are provided by the Intel TDX module. Logically, guest-side functions run in TDX non-root mode, though the CPU runs VMX root mode. TDX non-root operation is described in Chapter 11.

Intel TDX transitions between TDX root operation and TDX non-root operation include **TD Entries,** from TDX root to TDX non-root mode, and **TD Exits** from TDX non-root to TDX root mode. A TD Exit might be **asynchronous**, triggered by some external event (e.g., external interrupt or SMI) or an exception, or it might be **synchronous**, triggered by a TDCALL(TDG.VP.VMCALL) function.

2.4. Guest TD Private Memory Protection

2.4.1.1. Memory Encryption

The Intel TDX module helps protect guest TD private memory using memory encryption and integrity protection as enabled by the CPU's MKTME and TDX ISA features. The Intel TDX module adds **key management functionality** to help enforce its security objectives.

Memory encryption is designed to be performed by encryption engines that reside at each memory controller. An encryption engine holds a table of encryption keys, known as the Key Encryption Table (KET). An encryption key is selected for each memory transaction based on a **Host Key Identifier (HKID)** that should be provided with the transaction.

In the first generation of MKTME, HKID is "stolen" from the physical address by allocating a configurable number of bits from the top of the physical address. TDX ISA is designed to further partition the HKID space into **shared HKIDs** for legacy MKTME accesses and **private HKIDs** for SEAM-mode-only accesses. Future generations might choose to express HKID differently.

During TDX non-root operation, memory accesses can be qualified as either shared or private, based on the value of a new SHARED bit in the Guest Physical Address (GPA). Shared accesses are intended to behave as legacy memory accesses and use the upper bits of the host physical address as an HKID, which must be from the range allocated to legacy MKTME. Private accesses use the guest TD's private HKID.

The host-side Intel TDX functions help provide the means for the host VMM to manage HKID assignment to guest TDs, configure the memory encryption engines, etc., while better assuring proper operation to help maintain the TDX's security objectives. By design, the host VMM does not have access to the encryption keys.

Encryption-based memory protection is described in the [MKTME PAS] and [SEAM PAS]. Key management is described in Chapter 4.

2.4.2. Address Translation

Guest Physical Address (GPA) space is divided into private and shared sub-spaces, determined by the SHARED bit of GPA.

As designed, the CPU translates shared GPAs using the Shared EPT, which resides in host VMM memory. The Shared EPT is directly managed by the host VMM – the same as with legacy VMX.

As designed, the CPU translates private GPAs using a separate Secure EPT. The Secure EPT pages are encrypted and integrity-protected with the TD's ephemeral private key. The Secure EPT is not intended to be directly accessible by any software other than the Intel TDX module, nor by any devices. Secure EPT can be managed indirectly by the host VMM, using Intel TDX functions. The Intel TDX module helps ensure that the Secure EPT security properties are kept. At the end of translation, the CPU sets the HKID bits in the HPA to the TD's assigned HKID.

TD private memory management is described in Chapter 9.

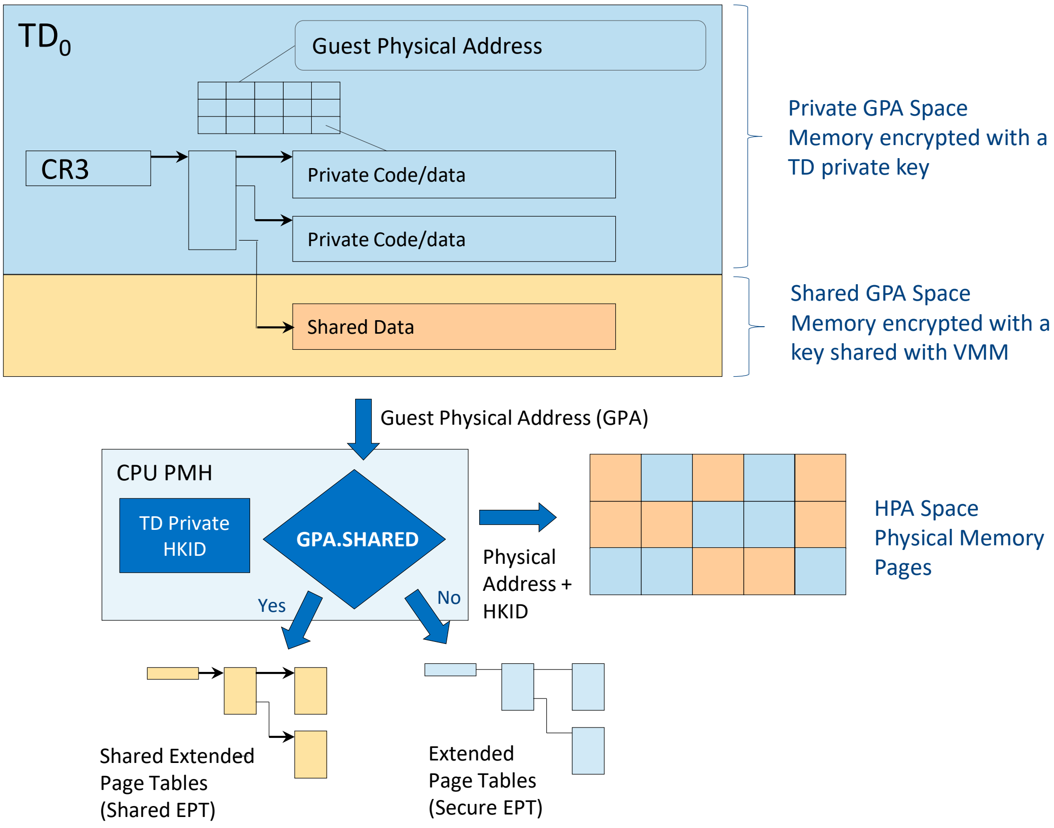


Figure 2.3: Secure EPT Concept

2.5. Guest TD State Protection

Intel TDX helps protect the confidentiality and integrity of a guest TD and the state of its Virtual CPUs (VCPUs) with the following mechanisms:

Protected Control Structures
TD-scope and TD VCPU-scope control structures, which hold guest TD metadata and TD VCPU state, are not directly accessible to any software (besides the Intel TDX module) or devices. As designed, the control structures are encrypted and integrity-protected with a private key, and managed by Intel TDX functions. TD control structures are described in Chapter 6.

VCPU State on TD Transitions
On asynchronous TD exits, which happen due to exceptions or external events, the CPU state is saved to the VCPU control structures, and a synthetic state is loaded into the CPU registers. On the following TD Entry, the CPU state is restored from the protected control structures.

On synchronous TD-initiated exit, using the TDCALL(TDG.VP.VMCALL) function, selected GPR and XMM state can be passed as-is to the host VMM. On the following TD entry, that state can be passed back as-is to the guest TD.

2.6. Intel TDX I/O Model

The TD guest can use the following I/O models:

- Paravirtualized devices
- Paravirtualized devices with MMIO emulation
- Direct assignment of devices to a TD

The Intel TDX architecture does not provide specific mechanisms for trusted I/O. Any integrity or confidentiality protection of data submitted to or received from physical or emulated devices must be done by the guest software using cryptography.

Intel TDX I/O is detailed in Chapter 13.

2.7. Measurement and Attestation

As designed, during TD launch, the initial contents and configuration of the TD are recorded by the Intel TDX module. In addition, run-time measurement registers can be used by the guest TD software, e.g., to measure a boot process. At run-time, the Intel TDX module reuses the Intel® Software Guard Extensions (Intel® SGX) attestation infrastructure to provide support for attesting to these measurements as described below.

Intel TDX attestation is intended to be used in two phases:

1. Software within the guest TD can use the TDCALL(TDG.MR.REPORT) function to request the Intel TDX module to generate an integrity-protected TDREPORT structure. The Intel TDX ISA provides support for enabling the Intel TDX module to create this structure that includes the TD's measurements, the Intel TDX module's measurements, and a value provided by the guest TD software. This will typically be an asymmetric key that the attestation verifier can use to establish a secure channel or protect sensitive data to be sent to the TD software.
2. An Intel SGX Quoting Enclave, written specifically to support quoting Intel TDX TDs, uses a new ENCLU instruction leaf, EVERIFYREPORT2, to help check the integrity of the TDG.MR.REPORT. If it passes, the Quoting Enclave can use a certified quote signing key to sign a quote containing the guest TD's measurements and the additional data being quoted.

The Quoting Enclave can run anywhere on the platform where Intel SGX is supported.

Note: Running Intel SGX enclaves within a guest TD is not supported.

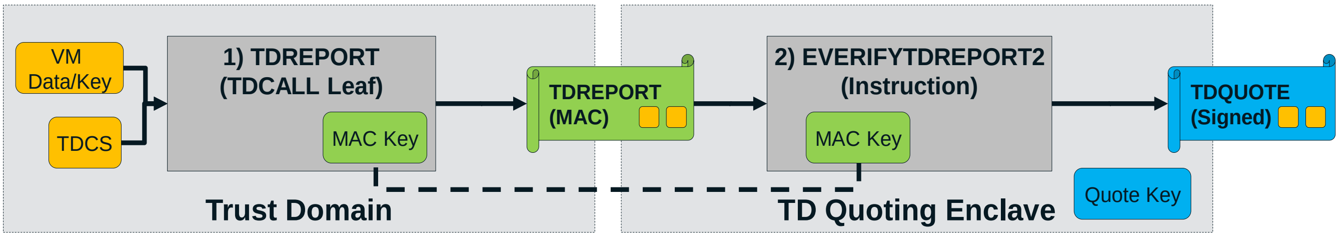


Figure 2.4: TD Attestation

TD measurement and attestation is described in Chapter 12.

2.8. Intel TDX Managed Control Structures

As designed, the Intel TDX module holds and manages a set of control structures that are not directly accessible to software (except the Intel TDX module itself). The controls structures are encrypted with private keys and HKIDs, and their content is only accessible in SEAM mode. Most control structures are addressable by the host VMM, which is responsible for allocating the memory to hold them.

The Intel TDX module uses control structures to help manage TD-private memory, transitions into and out of TDX non-root operation (TD entries and TD exits), as well as processor behavior in TDX non-root operation.

Table 2.2: TDX-Managed Control Structures Overview

Scope	Name	Meaning	Description
Platform	KOT	Key Ownership Table	Designed to control private HKID assignment. KOT is internal to the Intel TDX module, intended not to be directly accessible to any other software.
	PAMT	Physical Address Metadata Table	The PAMT is designed to hold metadata of each page in a Trust Domain Memory Range (TDMR). It controls assignment of physical pages to guest TDs, etc. The PAMT is intended not to be directly accessible to software. It resides in memory allocated by the host VMM on TDX initialization.

Scope	Name	Meaning	Description
Guest TD	TDR	Trust Domain Root	The TDR is intended to be the root control structure of a guest TD. It controls the key management and build/teardown process. The TDR is not intended to be directly accessible to software. It resides in memory allocated by the host VMM, via Intel TDX interface functions.
	TDCS	Trust Domain Control Structure	The TDCS is intended to control the operation of a guest TD as a whole. The TDCS is not intended to be directly accessible to software. It resides in memory allocated by the host VMM, via Intel TDX interface functions.
	SEPT	Secure EPT	The TDX-managed Extended Page Table (EPT) tree, used to help securely manage address translation for the TD private pages. The SEPT is not intended to be directly accessible to software. SEPT pages reside in memory allocated by the host VMM via Intel TDX interface functions.
Guest TD VCPU	TDVPS	Trust Domain Virtual Processor State	The TDVPS helps control the operation and hold the state of a guest TD virtual processor. It holds the TD VMCS and its auxiliary structures as well as other non-VMX control and state fields. The TDVPS is not intended to be directly accessible to software. It resides in memory allocated by the host VMM, via Intel TDX interface functions.

Intel TDX control structures are described in Chapter 6.

2.9. Intel TDX Interface Functions

The Intel TDX module implements functions that are triggered by executing two TDX instructions:

SEAMCALL The instruction used by the host VMM to invoke **host-side TDX interface functions**. The desired interface function is selected by an input operand (**leaf number**, in RAX). Host-side interface function names start with TDH (Trust Domain Host).

TDCALL The instruction used by the guest TD software (in TDX non-root mode) to invoke **guest-side TDX functions**. The desired interface function is selected by an input operand (**leaf number**, in RAX). Guest-side interface function names start with TDG (Trust Domain Guest).

Intel TDX interface function details are described in the [TDX Module ABI].

2.9.1. Host-Side (SEAMCALL Leaf) Interface Functions

Table 2.3: Host-Side (SEAMCALL Leaf) Interface Functions

Class	Interface Function Name	Leaf #	Description
Intel TDX Module Management	TDH.SYS.CONFIG	45	Globally configure the Intel TDX module
Intel TDX Module Management	TDH.SYS.INFO	32	Get Intel TDX module information
Intel TDX Module Management	TDH.SYS.INIT	33	Globally initialize the Intel TDX module
Intel TDX Module Management	TDH.SYS.KEY.CONFIG	31	Configure the Intel TDX global private key on the current package
Intel TDX Module Management	TDH.SYS.LP.INIT	35	Initialize the Intel TDX module per logical processor
Intel TDX Module Management	TDH.SYS.LP.SHUTDOWN	44	Shutdown the Intel TDX module on the current LP
Intel TDX Module Management	TDH.SYS.RD	34	Read a TDX Module global-scope metadata field
Intel TDX Module Management	TDH.SYS.RDALL	37	Read all host-readable TDX Module global-scope metadata fields
Intel TDX Module Management	TDH.SYS.TDMR.INIT	36	Partially initialize a Trust Domain Memory Region (TDMR)

Class	Interface Function Name	Leaf #	Description
TD Management	TDH.MNG.ADDCX	1	Add a control structure page to a TD
TD Management	TDH.MNG.CREATE	9	Create a guest TD and its TDR root page
TD Management	TDH.MNG.INIT	21	Initialize per-TD control structures
TD Management	TDH.MNG.KEY.CONFIG	8	Configure the TD private key on a single package
TD Management	TDH.MNG.KEY.FREEID	20	Mark the guest TD's HKID as free
TD Management	TDH.MNG.KEY.RECLAIMID	27	Does nothing; provided for backward compatibility
TD Management	TDH.MNG.RD	11	Read TD metadata
TD Management	TDH.MNG.VPFLUSHDONE	19	Check all of a guest TD's VCPUs have been flushed by TDH.VP.FLUSH
TD Management	TDH.MNG.WR	13	Write TD metadata
VCPU Scope	TDH.VP.ADDCX	4	Add a control structure page to a TD VCPU
VCPU Scope	TDH.VP.CREATE	10	Create a guest TD VCPU and its TDVPR root page
VCPU Scope	TDH.VP.ENTER	0	Enter TDX non-root operation
VCPU Scope	TDH.VP.FLUSH	18	Flush the address translation caches and cached TD VMCS associated with a TD VCPU
VCPU Scope	TDH.VP.INIT	22	Initialize the per-VCPU control structures
VCPU Scope	TDH.VP.RD	26	Read VCPU metadata
VCPU Scope	TDH.VP.WR	43	Write VCPU metadata
Physical Memory Management	TDH.PHYMEM.CACHE.WB	40	Write back the contents of the cache on a package
Physical Memory Management	TDH.PHYMEM.PAGE.RDMD	24	Read the metadata of a page in a TDMR
Physical Memory Management	TDH.PHYMEM.PAGE.RECLAIM	28	Reclaim a physical memory page owned by a TD (i.e., TD private page, Secure EPT page or a control structure page)
Physical Memory Management	TDH.PHYMEM.PAGE.WBINVD	41	Write back and invalidate all cache lines associated with the specified memory page and HKID
Private Memory Management	TDH.MEM.PAGE.ADD	2	Add a 4KB private page to a TD during TD build time
Private Memory Management	TDH.MEM.PAGE.AUG	6	Dynamically add a 4KB private page to an initialized TD
Private Memory Management	TDH.MEM.PAGE.DEMOTE	15	Split a 2MB or a 1GB private TD page mapping into 512 4KB or 2MB page mappings respectively
Private Memory Management	TDH.MEM.PAGE.PROMOTE	23	Merge 512 consecutive 4KB or 2MB private TD page mappings into one 2MB or 1GB page mapping respectively
Private Memory Management	TDH.MEM.PAGE.RELOCATE	5	Relocate a 4KB mapped page from its HPA to another
Private Memory Management	TDH.MEM.PAGE.REMOVE	29	Remove a private page from a guest TD
Private Memory Management	TDH.MEM.RANGE.BLOCK	7	Block a TD private GPA range
Private Memory Management	TDH.MEM.RANGE.UNBLOCK	39	Remove the blocking of a TD private GPA range
Private Memory Management	TDH.MEM.RD	12	Read from private memory of a debuggable guest TD
Private Memory Management	TDH.MEM.SEPT.ADD	3	Add and map a 4KB Secure EPT page to a TD
Private Memory Management	TDH.MEM.SEPT.RD	25	Read a Secure EPT entry
Private Memory Management	TDH.MEM.SEPT.REMOVE	30	Remove a Secure EPT page from a TD
Private Memory Management	TDH.MEM.TRACK	38	Increment the TD's TLB tracking counter
Private Memory Management	TDH.MEM.WR	14	Write to private memory of a debuggable guest TD
Measurement and Attestation	TDH.MR.EXTEND	16	Extend the guest TD measurement register during TD build
Measurement and Attestation	TDH.MR.FINALIZE	17	Finalize the guest TD measurement register
Service TD	TDH.SERVTD.BIND	48	Bind a service TD to a target TD
Service TD	TDH.SERVTD.MSG	49	Process a service TD request message
Migration	TDH.MIG.STREAM.CREATE	96	Create a migration stream
Migration Export	TDH.EXPORT.ABORT	64	Abort an export session
Migration Export	TDH.EXPORT.BLOCKW	65	Block a TD private page for writing
Migration Export	TDH.EXPORT.RESTORE	66	Cancel the export of a previously exported TD private page
Migration Export	TDH.EXPORT.MEM	68	Export a TD private page
Migration Export	TDH.EXPORT.PAUSE	70	Pause the exported TD

Class	Interface Function Name	Leaf #	Description
Migration Export	TDH.EXPORT.TRACK	71	End an in-order export phase epoch and start a new epoch or the out-of-order phase
Migration Export	TDH.EXPORT.STATE.IMMUTABLE	72	Start an export session and export the TD's immutable state
Migration Export	TDH.EXPORT.STATE.TD	73	Export the TD's mutable state
Migration Export	TDH.EXPORT.STATE.VP	74	Export a VCPU mutable state
Migration Export	TDH.EXPORT.UNBLOCKW	75	Unblock a page that has been blocked for writing
Migration Import	TDH.IMPORT.ABORT	80	Abort an import session
Migration Import	TDH.IMPORT.END	81	End an import session
Migration Import	TDH.IMPORT.COMMIT	82	Commit the import session and allow the imported TD to run
Migration Import	TDH.IMPORT.MEM	83	Import a TD private page
Migration Import	TDH.IMPORT.TRACK	84	Process a start token and end the in-order import phase
Migration Import	TDH.IMPORT.STATE.IMMUTABLE	85	Start an import session and import the TD's immutable state
Migration Import	TDH.IMPORT.STATE.TD	86	Import the TD's mutable state
Migration Import	TDH.IMPORT.STATE.VP	87	Import a VCPU mutable state

2.9.2. Guest-Side (TDCALL Leaf) Interface Functions

Table 2.4: Guest-Side (TDCALL Leaf) Interface Functions

Class	Interface Function Name	Leaf #	Description
Global Scope	TDG.SYS.RD	11	Read a TDX Module global-scope metadata field
Global Scope	TDG.SYS.RDALL	12	Read all gust-readable TDX Module global-scope metadata fields
VM Scope	TDG.VM.RD	7	Read a TD-scope metadata field
VM Scope	TDG.VM.WR	8	Write a TD-scope metadata field
VCPU Scope	TDG.VP.CPUIDVE.SET	5	Control delivery of #VE on CPUID instruction execution
VCPU Scope	TDG.VP.INFO	1	Get TD execution environment information
VCPU Scope	TDG.VP.RD	9	Read a VCPU-scope metadata field
VCPU Scope	TDG.VP.VEINFO.GET	3	Get Virtualization Exception Information for the recent #VE exception
VCPU Scope	TDG.VP.VMCALL	0	Call a host VM service
VCPU Scope	TDG.VP.WR	10	Write a VCPU-scope metadata field
Private Memory Management	TDG.MEM.PAGE.ACCEPT	6	Accept a pending private page into the TD
Measurement and Attestation	TDG.MR.REPORT	4	Creates a cryptographic report of the TD
Measurement and Attestation	TDG.MR.RTMR.EXTEND	2	Extend a TD run-time measurement register

3. Software Use Cases

This chapter summarizes the software use cases (also known as software flows) used with the Intel TDX module.

3.1. Intel TDX Module Lifecycle

3.1.1. UPDATED: Intel TDX Module Platform-Scope Initialization

This sequence is intended to be used by the host VMM to initialize the Intel TDX module at the platform scope.

Table 3.1: Typical Intel TDX Module Platform-Scope Initialization Sequence

Phase		Intel TDX Function	Scope	Execute On	Description
Boot	1	N/A	Platform	Each core	BIOS configures Convertible Memory Regions (CMRs); MCHECK checks them and securely stores the information.
P-SEAMLDR Loading	2	N/A	Platform	One of the BSPs	OS launches the NP-SEAMLDR ACM, which loads the Intel P-SEAMLDR module.
Intel TDX Module Loading	3	SEAMLDR.INSTALL	Platform	All LPs	VMM calls the P-SEAMLDR with “load” scenario to install the TDX module.
Intel TDX Module Initialization	4	TDH.SYS.INIT	Platform	Any one LP	Perform global initialization of the Intel TDX module.
	5	TDH.SYS.LP.INIT	LP	Each LP	Perform LP-scope, core-scope and package-scope initialization, checking and configuration of the platform and the Intel TDX module.
Enumeration and Configuration	6	TDH.SYS.RD* or TDH.SYS.INFO	Platform	Any initialized LP	Retrieve Intel TDX module information and convertible memory (CMR) information.
	7	TDH.SYS.CONFIG	Platform	Any one LP	Configure the Intel TDX module with TDMR and PAMT setup.
	8	N/A	Package	Each Package	If any MODIFIED cache lines may exist for the PAMT ranges, flush them to memory using, e.g., WBINVD.
	9	TDH.SYS.KEY.CONFIG	Package	Each Package	Configure the Intel TDX global private key used for encrypting PAMT and TDR on the hardware (other TD-scope control structures are encrypted with their respective TD’s ephemeral private keys).
	At this point any Intel TDX function may be executed on any LP.				
Memory Initialization	10	TDH.SYS.TDMR.INIT (multiple)	Platform	One or more LPs	Called multiple times to gradually initialize the PAMT structure for each TDMR.
	Once each 1GB block of TDMR has been initialized by TDH.SYS.TDMR.INIT, it can be used to hold TD-private pages.				

3.1.2. Intel TDX Module Reload and Update

In both reload and update scenarios, the previous TDX module in SEAM range is erased when the next TDX module is installed. Since the previous module can’t pass any information to the next TDX module, the next TDX module starts afresh, and all guest TDs’ context and memory out of SEAM range becomes effectively inaccessible.

Table 3.2: TDX Module Reload Sequence

Phase		Intel TDX Function	Scope	Execute On	Description
P-SEAMLDR	1	SEAMLDR.INSTALL with “load” or “update” scenario	Platform	All LPs	Installs the next TDX module. If the scenario is “load” then the installation proceeds regardless of the previous TDX module. If the scenario is “update” then the installation fails is the previous TDX module had higher SEAM SVN.
Next TDX Module	The initialization sequence continues in the same way as described in 3.1.1 above, steps 3 to 8.				

3.2. TD Build

The following sequence is intended to be used by the host VMM to build a TD.

Table 3.3: Typical TD Build Sequence

	Step		Description	SEAMCALL Leaf Functions
A	**TD Creation and Key Resource Assignment**	1	The host VMM finds/allocates a free HKID for the new TD.	TDH.MNG.CREATE TDH.MNG.KEY.CONFIG
		2	The host VMM allocates a 4K page for the TDR in TDMR. If any MODIFIED cache lines may exist for this page, the host VMM flushes them to memory using, e.g., CLFLUSHOPT or TDH.PHYMEM.PAGE.WBINVD.	
		3	The host VMM creates the new TD by calling the TDH.MNG.CREATE function (passing HPA of the TDR page). This initializes the target TDR page.	
		4	The TD host VMM configures the MKTME hardware with the TD’s private key by calling the TDH.MNG.KEY.CONFIG function on each package.	
		5	At this point, the TD private memory is accessible. The VMM can use Intel TDX interface functions to create control structures and TD private pages as described below.	
B	**TDCS Memory Allocation and TD Initialization**	1	The host VMM allocates multiple 4KB TDCX pages for TDCS. The number of required TDCX pages is enumerated by TDH.SYS.RD* or TDH.SYS.INFO. If any MODIFIED cache lines may exist for these pages, the host VMM flushes them to memory using, e.g., CLFLUSHOPT or TDH.PHYMEM.PAGE.WBINVD.	TDH.MNG.ADDCX TDH.MNG.INIT
		2	For each TDCX page, the host VMM calls the TDH.MNG.ADDCX function (passing HPA of TDCX) to add the page to the TD.	
		3	The host VMM builds a TD_PARAMS structure. For example, the TD configuration parameters can be obtained from a TD manifest supplied by the TD owner.	
		4	The host VMM calls the TDH.MNG.INIT function (passing the TD_PARAMS structure) to initialize the TD.	

	Step		Description	SEAMCALL Leaf Functions
C	**Virtual Processor Creation and Configuration (Executed per each VCPU)**	1	The host VMM allocates target pages for the VCPU's TDVPR and TDCX pages in TDMR in the context of a TD. The number of required TDCX pages is enumerated by TDH.SYS.RD* or TDH.SYS.INFO. If any MODIFIED cache lines may exist for these pages, the host VMM flushes them to memory using, e.g., CLFLUSHOPT or TDH.PHYMEM.PAGE.WBINVD.	TDH.VP.CREATE TDH.VP.ADDCX TDH.VP.INIT TDH.VP.WR
		2	The host VMM creates a new TD virtual CPU by calling the TDH.VP.CREATE function (passing the HPA of the new TDVPR page and its owner TDR page).	
		3	For each TDCX page, the TDRM calls the TDH.VP.ADDCX function (passing the HPA of the new TDCX page and its parent TDVPR page).	
		4	The host VMM initializes the TD VCPU by calling the TDH.VP.INIT function (passing the HPA of its TDVPR page). It also passes a single 64b parameter that is later passed to the VBIOS in the initial value of RCX. This parameter can be used as a pointer to a configuration structure in shared memory.	
		5	The host VMM allocates Shared EPT for each VP.	
		6	The host VMM uses the TDH.VP.WR function to write to the TD VMCS Shared EPTP field.	
		7	The host VMM may modify a few TD VMCS execution control fields using TDH.VP.WR.	
D	**TD Boot Memory Setup**	1	The host VMM loads the TD boot image to its memory. The boot image contains code and data pages that typically include a virtual BIOS, OS boot loader, configuration, etc.	TDH.MEM.SEPT.ADD TDH.MEM.PAGE.ADD TDH.MR.EXTEND
		2	The host VMM builds the TD Secure EPT by allocating physical pages and calling the TDH.MEM.SEPT.ADD function multiple times. If any MODIFIED cache lines may exist for these pages, the host VMM flushes them to memory using, e.g., CLFLUSHOPT or TDH.PHYMEM.PAGE.WBINVD.	
		3	The host VMM allocates the initial set of physical pages for the TD boot image and maps them into host address space. If any MODIFIED cache lines may exist for these pages, the host VMM flushes them to memory using, e.g., CLFLUSHOPT or TDH.PHYMEM.PAGE.WBINVD.	
		4	For each TD page: 1. The host VMM specifies a TDR as a parameter and calls the TDH.MEM.PAGE.ADD function. It copies the contents from the TD image page into the target TD page which is encrypted with the TD ephemeral key. TDH.MEM.PAGE.ADD also extends the TD measurement with the page GPA. 2. The host VMM extends the TD measurement with the contents of the new page by calling the TDH.MR.EXTEND function on each 256-byte chunk of the new TD page.	
E	**TD Measurement Finalization**	1	The host VMM calls the TDH.MR.FINALIZE function, which finalizes the TD measurement.	TDH.MR.FINALIZE
		2	At this point, the TD is finalized. • Its measurement cannot be modified anymore (except the run-time measurement registers). • TD VCPUs can be entered using SEAMCALL(TDH.VP.ENTER).	

3.3. TD Run Time

3.3.1. Private Memory Management

3.3.1.1. Dynamic Page Addition (Shared to Private Conversion)

5 The following sequence is intended to be used by the host VMM to dynamically add a page to a guest TD.

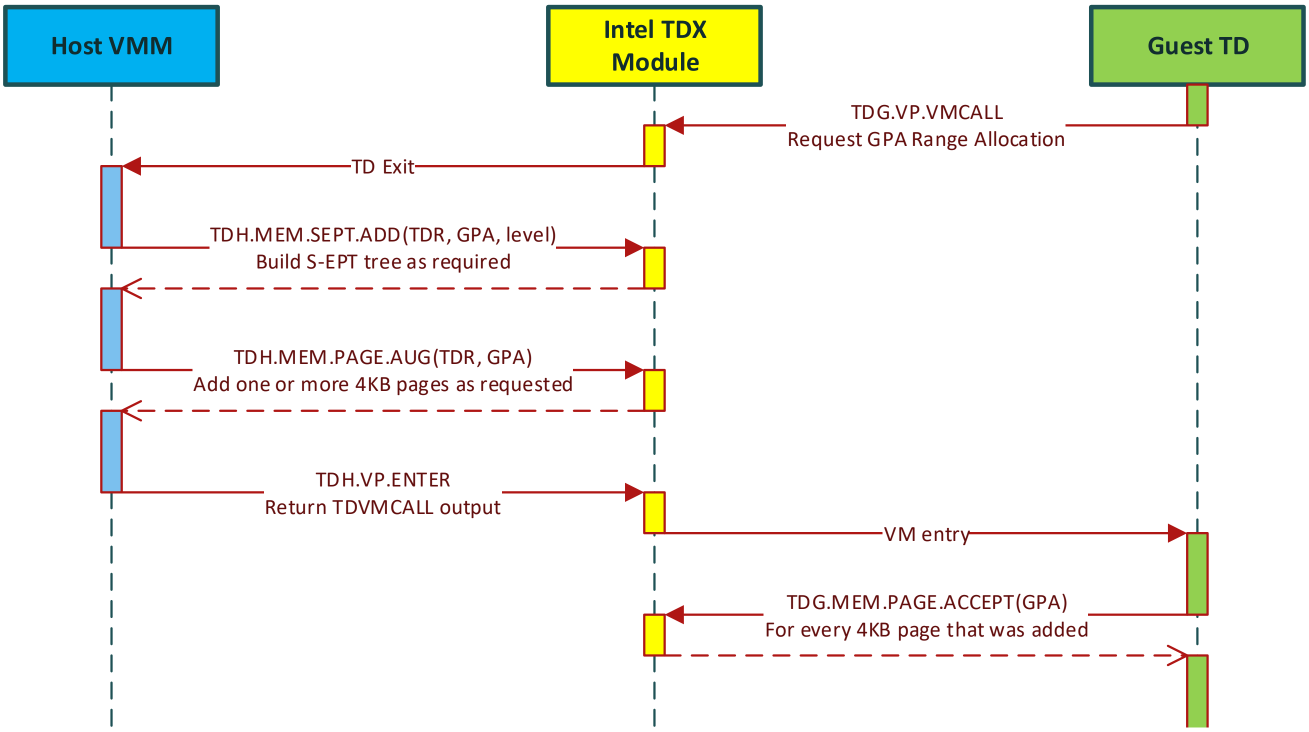

Host VMM
Intel TDX Module
Guest TD
TDG.VP.VMCALL
Request GPA Range Allocation
TD Exit
TDH.MEM.SEPT.ADD(TDR, GPA, level)
Build S-EPT tree as required
TDH.MEM.PAGE.AUG(TDR, GPA)
Add one or more 4KB pages as requested
TDH.VP.ENTER
Return TDVMCALL output
VM entry
TDG.MEM.PAGE.ACCEPT(GPA)
For every 4KB page that was added

Figure 3.1: Typical Dynamic Page Addition Sequence

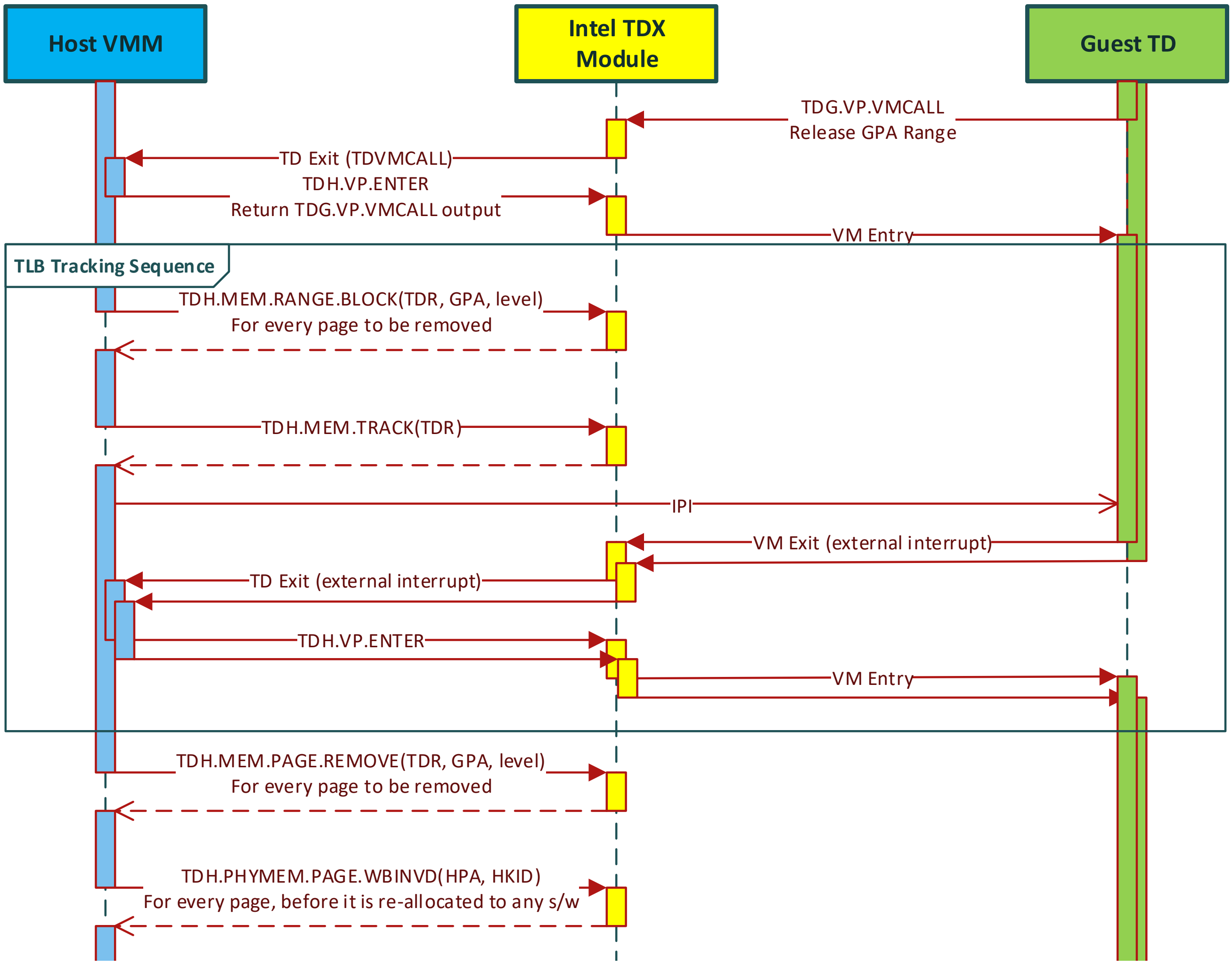
Table 3.4: Typical Dynamic Page Addition (Shared to Private Conversion) Sequence

Phase		Side	Intel TDX Function	Scope	Execute On	Description
Allocation Request	1	TD	TDG.VP.VMCALL	TD	Any LP	Optional software protocol: Request GPA range allocation.
Page Addition	2	VMM	TDH.MEM.SEPT.ADD	TD	Any LP	If required, update the Secure EPT.
	3	VMM	TDH.MEM.PAGE.AUG (multiple)	TD	Any LP	Add one or more new 4KB or 2MB private pages.
	At this point, the new page is pending acceptance by the guest TD and cannot be accessed by it yet.					
	4	VMM	TDH.VP.ENTER	TD	Any LP	Optional software protocol: Return TDG.VP.VMCALL result.
Page Acceptance	5	TD	TDG.MEM.PAGE.ACCEPT (multiple)	TD	Any LP	Accept the new pending page(s). Content of each page is zeroed out.
	At this point, the new page can be accessed by the guest TD.					

10

3.3.1.2. *Dynamic Page Removal (Private to Shared Conversion)*

The following sequence is intended to be used by the host VMM to dynamically remove a page from a guest TD. Dynamic page removal is detailed in 9.14.


Host VMM
Intel TDX Module
Guest TD
TDG.VP.VMCALL
Release GPA Range
TD Exit (TDVMCALL)
TDH.VP.ENTER
Return TDG.VP.VMCALL output
VM Entry
TLB Tracking Sequence
TDH.MEM.RANGE.BLOCK(TDR, GPA, level)
For every page to be removed
TDH.MEM.TRACK(TDR)
IPI
VM Exit (external interrupt)
TD Exit (external interrupt)
TDH.VP.ENTER
VM Entry
TDH.MEM.PAGE.REMOVE(TDR, GPA, level)
For every page to be removed
TDH.PHYMEM.PAGE.WBINVD(HPA, HKID)
For every page, before it is re-allocated to any s/w

5 Figure 3.2: Typical Dynamic Page Removal Sequence

Table 3.5: Typical Dynamic Page Removal (Private to Shared Conversion) Sequence

Phase		Side	Intel TDX Function	Scope	Execute On	Description
Ballooning Notification	1	TD	TDG.VP.VMCALL	TD	Any LP	Optional software protocol: Release GPA range.
	2	VMM	TDH.VP.ENTER	TD	Any LP	Optional software protocol: Return TDG.VP.VMCALL result.
TLB Tracking Sequence	3	VMM	TDH.MEM.RANGE.BLOCK (multiple)	TD	Any LP	Block private pages from further address translation.
	4	VMM	TDH.MEM.TRACK	TD	Any one LP	Increment the TD's TLB epoch.
	5	VMM	N/A	TD	Multiple LPs	Send an IPI, causing TD exit on any remote LP associated with a VCPU. Subsequent TDH.VP.ENTER will flush TLB.

Phase		Side	Intel TDX Function	Scope	Execute On	Description
Page Removal	6	VMM	TDH.MEM.PAGE.REMOVE (multiple)	TD	Any LP	Clear Secure EPT entry, and mark the physical page as free.
Cache Flushing	Before re-allocating any of the removed pages to any use, the host VMM must assure none of the cache lines of the removed pages are in the MODIFIED state to avoid corruption due to cache line aliasing. This is done using one of the following methods:					
	7a	VMM	TDH.PHYMEM.PAGE.WBINVD (multiple)	TD	Any one LP	Flush the cache lines of the removed page(s).
	7b	VMM	WBNOINVD	Platform	One LP per package[2]	Globally write back all caches.
	7c	VMM	WBINVD	Platform	One LP per package[3]	Globally write back and invalidate all caches.

3.3.1.3. *Page Promotion (Mapping Merge)*

Page size promotion is intended to be used by the host VMM to merge 512 pages mapped as 4KB or 2MB into a single page mapped as 2MB or 1GB, respectively. It is detailed in 9.11.

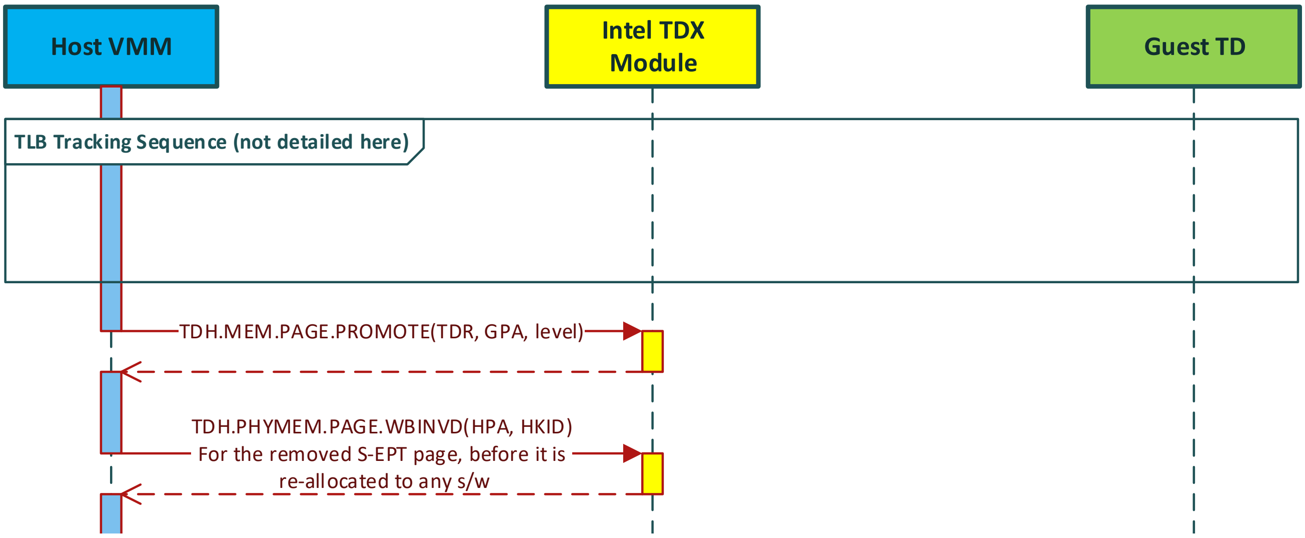

Host VMM
Intel TDX Module
Guest TD
TLB Tracking Sequence (not detailed here)
TDH.MEM.PAGE.PROMOTE(TDR, GPA, level)
TDH.PHYMEM.PAGE.WBINVD(HPA, HKID)
For the removed S-EPT page, before it is
re-allocated to any s/w

Figure 3.3: Typical Page Promotion Sequence

[2] Some CPUs may require running WBNOINVD per core.

[3] Some CPUs may require running WBINVD per core.

Table 3.6: Typical Page Promotion (Mapping Merge) Sequence

Phase		Intel TDX Function	Scope	Execute On	Description
TLB Tracking Sequence	1	TDH.MEM.RANGE.BLOCK	TD	Any LP	Block the GPA range to be merged from further address translation.
	2	TDH.MEM.TRACK	TD	Any one LP	Increment the TD's TLB epoch.
	3	N/A	TD	Multiple LPs	Send an IPI, causing TD exit on any remote LP associated with a VCPU. Subsequent TDH.VP.ENTER will flush TLB.
Promotion	4	TDH.MEM.PAGE.PROMOTE	TD	Any LP	Merge small pages in the GPA range into a large page.
Cache Flushing	5	TDH.PHYMEM.PAGE.WBINVD	TD	Any LP	Flush the removed Secure EPT page's cache lines.

3.3.1.4. *Page Demotion (Mapping Split)*

Page size demotion is intended to be used by the host VMM to split a page mapped as 1GB or 2MB into 512 pages mapped as 2MB or 4KB, respectively. It is detailed in 9.12.

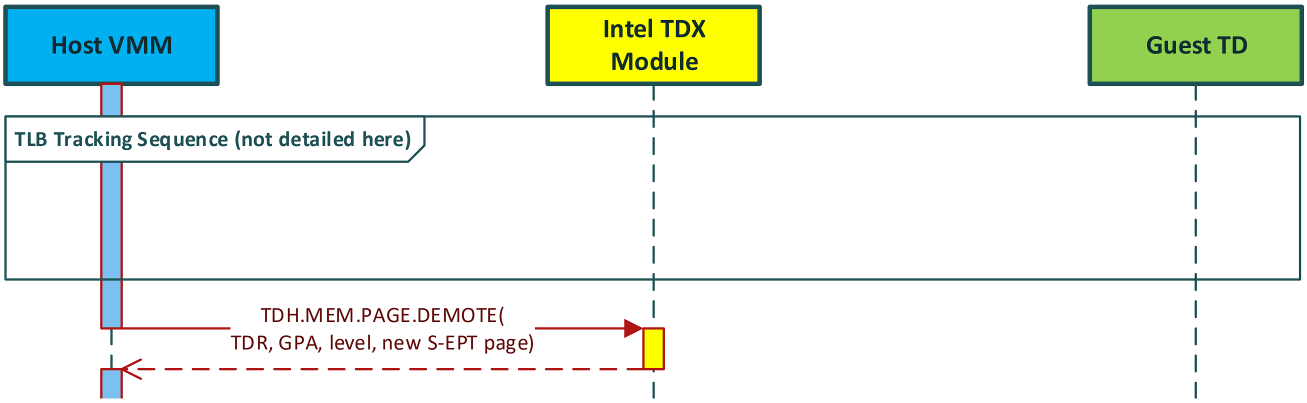

Host VMM
Intel TDX Module
Guest TD
TLB Tracking Sequence (not detailed here)
TDH.MEM.PAGE.DEMOTE(
TDR, GPA, level, new S-EPT page)

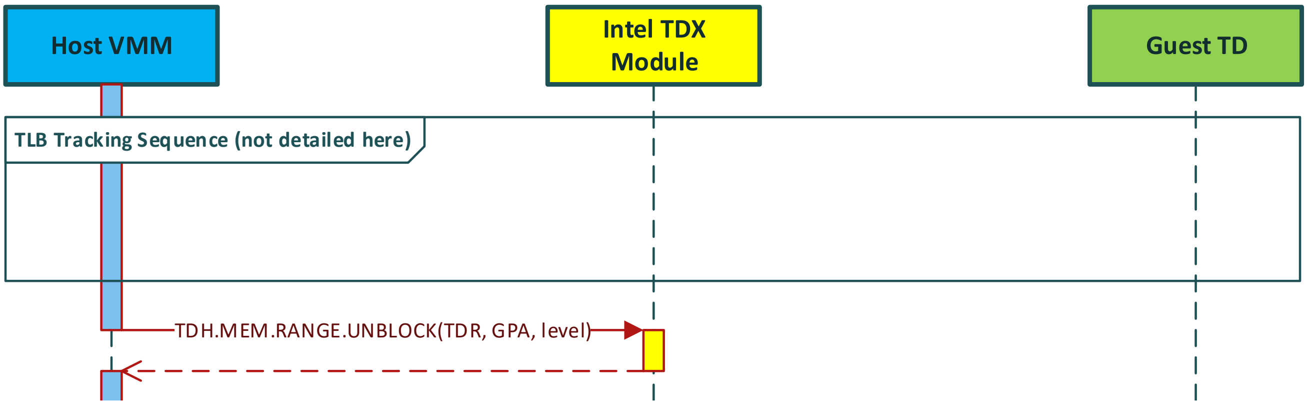
Figure 3.4: Typical Page Demotion Sequence

Table 3.7: Typical Page Demotion (Mapping Split) Sequence

Phase		Intel TDX Function	Scope	Execute On	Description
TLB Tracking Sequence	1	TDH.MEM.RANGE.BLOCK	TD	Any LP	Block private large page from further address translation.
	2	TDH.MEM.TRACK	TD	Any one LP	Increment the TD's TLB epoch.
	3	N/A	TD	Multiple LPs	Send an IPI, causing TD exit on any remote LP associated with a VCPU. Subsequent TDH.VP.ENTER will flush TLB.
Demotion	4	TDH.MEM.PAGE.DEMOTE	TD	Any LP	Split the large page into multiple small pages.

3.3.1.5. *GPA Range Unblock*

GPA range unblock is intended to be used when a range has been blocked, for example, for page removal, but the host VMM decides to cancel the operation. Unblock is detailed in 9.16.


Host VMM
Intel TDX Module
Guest TD
TLB Tracking Sequence (not detailed here)
TDH.MEM.RANGE.UNBLOCK(TDR, GPA, level)

5 **Figure 3.5: Typical GPA Range Unblock Sequence**

Table 3.8: Typical GPA Range Unblock Sequence

Phase		Intel TDX Function	Scope	Execute On	Description
TLB Tracking Sequence	1	TDH.MEM.RANGE.BLOCK (multiple)	TD	Any LP	Block private GPA range from further address translation.
	2	TDH.MEM.TRACK	TD	Any one LP	Increment the TD's TLB epoch.
	3	N/A	TD	Multiple LPs	Send an IPI, causing TD exit on any remote LP associated with a VCPU. Subsequent TDH.VP.ENTER will flush TLB.
Unblocking	4	TDH.MEM.RANGE.UNBLOCK	TD	Any LP	Remove the private GPA range blocking.

3.3.2. Guest TD Execution

10 3.3.2.1. *TD VCPU First-Time Invocation*

Table 3.9: Typical TD VCPU First-Time Invocation Sequence

Phase		Side	Intel TDX Function	Scope	Execute On	Description
Entering TD VCPU (First Time)	1	VMM	N/A	LP	LP x	Save VMM LP state not preserved across TD Entry to TD exit.
	2	VMM	TDH.VP.ENTER	VCPU/LP	LP x	Restore initial LP state, as set by TDH.VP.INIT, from TDVPS and enter TDX non-root mode.
TD VCPU Initial Execution	TD software (VBIOS) starts execution in 32-bit protected mode with no paging.					
	3	TD	N/A	VCPU/LP	LP x	TD software parses initial information in GPR, builds page tables and switches to 64-bit mode.
	TD software (VBIOS) now executes in 64-bit mode.					
Enumeration	4	TD	TDG.VP.INFO	VCPU/LP	LP x	TD software retrieves basic TD and execution environment information.

Phase		Side	Intel TDX Function	Scope	Execute On	Description
	5	TD	TDG.MR.REPORT	VCPU/LP	LP x	TD software retrieves additional TD information.
	TD continues execution in TDX non-root mode.					

3.3.2.2. *TD VCPU Entry, Exit on TDG.VP.VMCALL and Re-Entry*

Table 3.10: Typical TD Entry, Exit on TDG.VP.VMCALL and Re-Entry Sequence

Phase		Side	Intel TDX Function	Scope	Execute On	Description
TD Entry	1	VMM	N/A	LP	LP x	Save VMM LP state not preserved across TD Entry to TD exit.
	2	VMM	TDH.VP.ENTER	VCPU/LP	LP x	Restore LP state from TDVPS and enter TDX non-root mode.
	TD executes in TDX non-root mode.					
Software Protocol over TDG.VP.VMCALL	3	TD	TDG.VP.VMCALL	VCPU/LP	LP x	Exit TDX non-root mode, save LP state to TDVPS, and set synthetic state (except most GPRs and all XMMs).
	4	VMM	N/A	LP	LP x	Optionally: Restore VMM LP state saved before TDH.VP.ENTER.
	5	VMM	N/A	LP	LP x	Perform TDG.VP.VMCALL function, as determined by the TD-VMM software contract (out of the scope for this document).
	6	VMM	N/A	LP	LP x	Save VMM LP state not preserved across TD Entry to TD exit.
	7	VMM	TDH.VP.ENTER	VCPU/LP	LP x	Restore LP state from TDVPS (except most GPRs and all XMMs), and enter TDX non-root mode.
	8	TD	N/A	VCPU/LP	LP x	Parse TDG.VP.VMCALL output operands as determined by TD – VMM software contract.
TD Execution	TD continues execution in TDX non-root mode.					

3.3.2.3. *TD VCPU Entry, Exit on Asynchronous Event and Re-Entry*

Table 3.11: Typical TD Entry, Exit on Asynchronous Event and Re-Entry Sequence

Phase		Side	Intel TDX Function	Scope	Execute On	Description
TD Entry	1	VMM	N/A	LP	LP x	Save LP state not preserved across TD Entry to TD exit.
	2	VMM	TDH.VP.ENTER	VCPU/LP	LP x	Restore LP state from TDVPS, and enter TDX non-root mode.
	TD executes in TDX non-root mode.					

Phase		Side	Intel TDX Function	Scope	Execute On	Description
Async. TD Exit and Re-Entry	3	TD	N/A	VCPU/LP	LP x	Asynchronous event (interrupt, exception, EPT violation, etc.) causes TD exit. Save LP state to TDVPS, and set synthetic state.
	4	VMM	N/A	LP	LP x	Restore any required LP state saved by the VMM before TDH.VP.ENTER.
	5	VMM	N/A	LP	LP x	Handle the asynchronous event.
	6	VMM	N/A	LP	LP x	Save VMM LP state not preserved across TD Entry to TD exit.
	7	VMM	TDH.VP.ENTER	VCPU/LP	LP x	Restore LP state from TDVPS and enter TDX non-root mode.
TD Execution	TD continues execution in TDX non-root mode.					

3.3.2.4. *Guest-Side Functions*

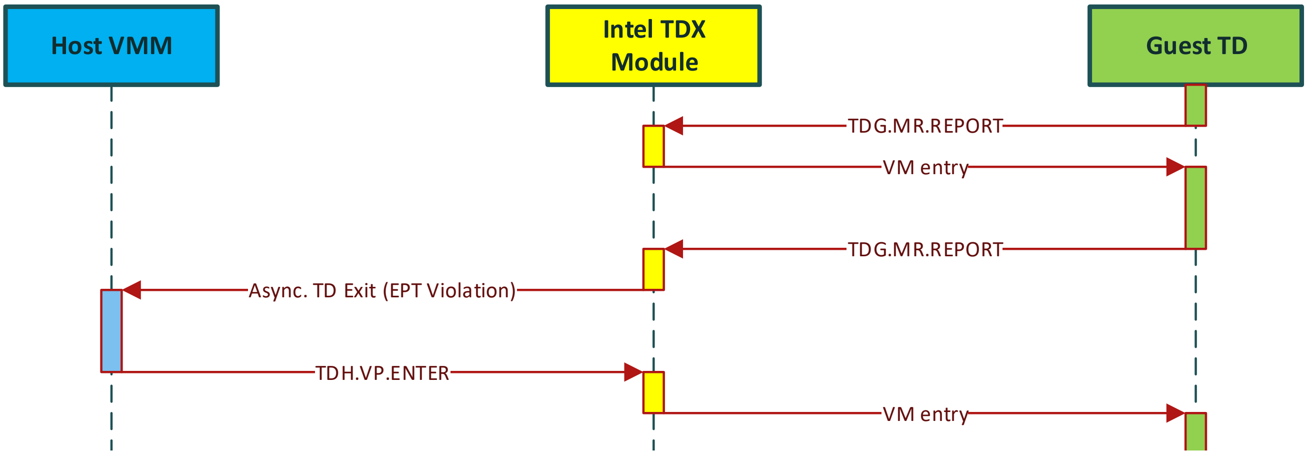


Figure 3.6: Typical Guest-Side Function Sequences

5

Table 3.12: Typical Guest-Side Functions Sequences

Case		Side	Intel TDX Function	Scope	Execute On	Description
Guest-Side Function Returns to Guest TD	TD executes in TDX non-root mode					
	1	TD	TDG.MR.REPORT	VCPU/LP	LP x	The guest TD VM exits to the Intel TDX module, which handles the guest-side function and re-enters the TD.
	TD continues execution in TDX non-root mode					
Guest-Side Function Causes Async. TD Exit	2	TD	TDG.MR.REPORT	VCPU/LP	LP x	The guest TD exits to the Intel TDX module, which handles the guest-side function, but an asynchronous event (e.g., EPT violation, etc.) causes TD exit.
	3	VMM	N/A	LP	LP x	Optional: The host VMM restores the VMM LP state saved before TDH.VP.ENTER.
	4	VMM	N/A	LP	LP x	The host VMM handles the asynchronous event.

Case		Side	Intel TDX Function	Scope	Execute On	Description
	5	VMM	N/A	LP	LP x	The host VMM saves any VMM LP state not preserved across TD Entry to TD exit.
	6	VMM	TDH.VP.ENTER	VCPU/LP	LP x	The Intel TDX module restores LP state from TDVPS and enters TDX non-root mode.
	TD continues execution in TDX non-root mode.					

3.3.2.5. *TD VCPU Rescheduling (Migration to Another LP)*

The Intel TDX module is designed to allow a TD VCPU to be associated with at most one LP at any time. The host VMM must explicitly break this association in order to migrate the VCPU to another LP.

Table 3.13: Typical VCPU Migration to Another LP Sequence

Phase		Intel TDX Function	Scope	Execute On	Description
Old VCPU→LP Association	1	Any VCPU-specific SEAMCALL leaf	VCPU	Old LP	Any VCPU-specific SEAMCALL leaf (e.g., TDH.VP.INIT, TDH.VP.ENTER, TDH.VP.RD, etc.) creates an association between the current LP and the VCPU.
Breaking Old VCPU→LP Association	2	TDH.VP.FLUSH	VCPU	Old LP	Break the VCPU-LP association: flush the VCPU's TD VMCS to TDVPS memory and flush the VCPU's TLB ASID.
	At this point the VCPU is not associated with any LP.				
New VCPU→LP Association	3	Any VCPU-specific SEAMCALL leaf	VCPU	New LP	Create a new VCPU-LP association.

3.4. TD Destruction

The following sequence is intended to be used by the host VMM to destroy a TD and reclaim all its resources.

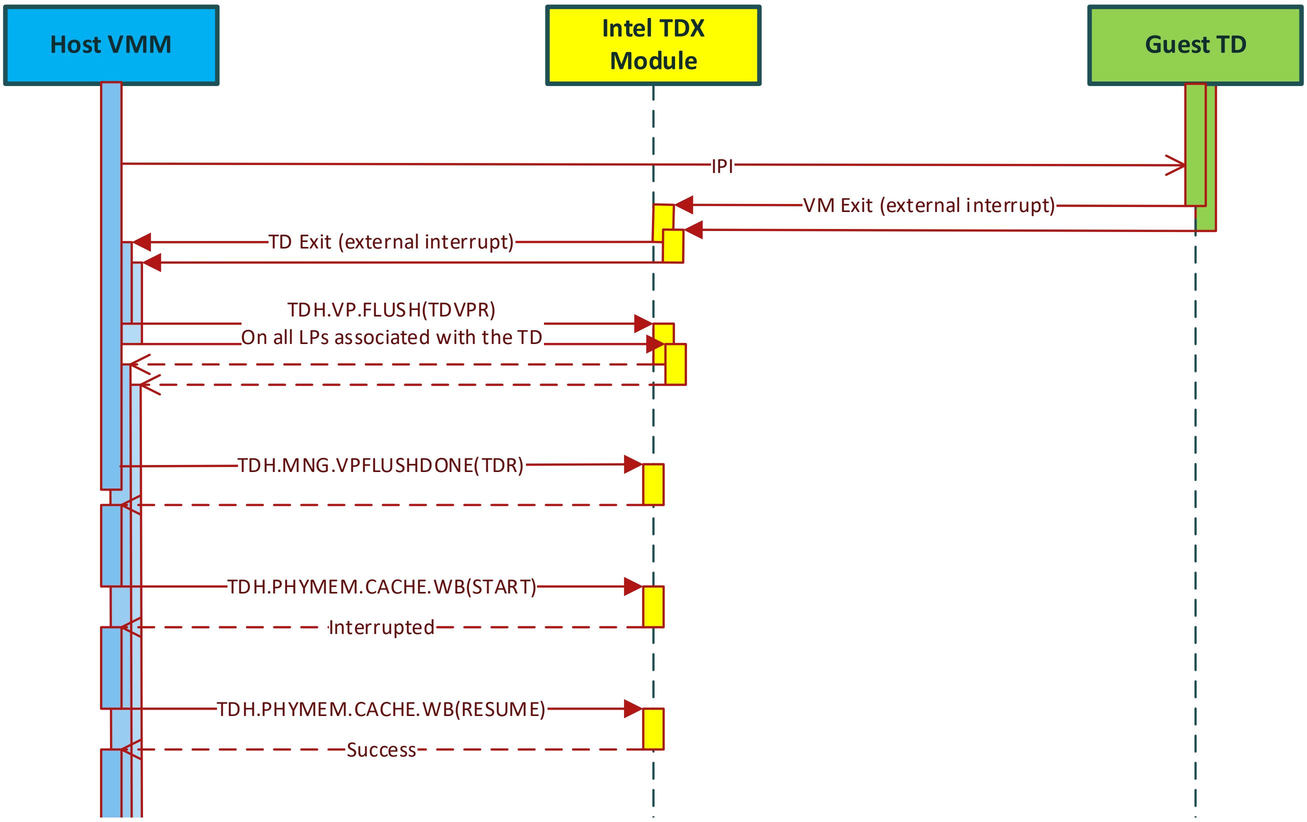

Host VMM
Intel TDX Module
Guest TD
IPI
VM Exit (external interrupt)
TD Exit (external interrupt)
TDH.VP.FLUSH(TDVPR)
On all LPs associated with the TD
TDH.MNG.VPFLUSHDONE(TDR)
TDH.PHYMEM.CACHE.WB(START)
Interrupted
TDH.PHYMEM.CACHE.WB(RESUME)
Success

Figure 3.7: Typical TD Destruction Sequence Step A: Stopping and Flushing Out

5

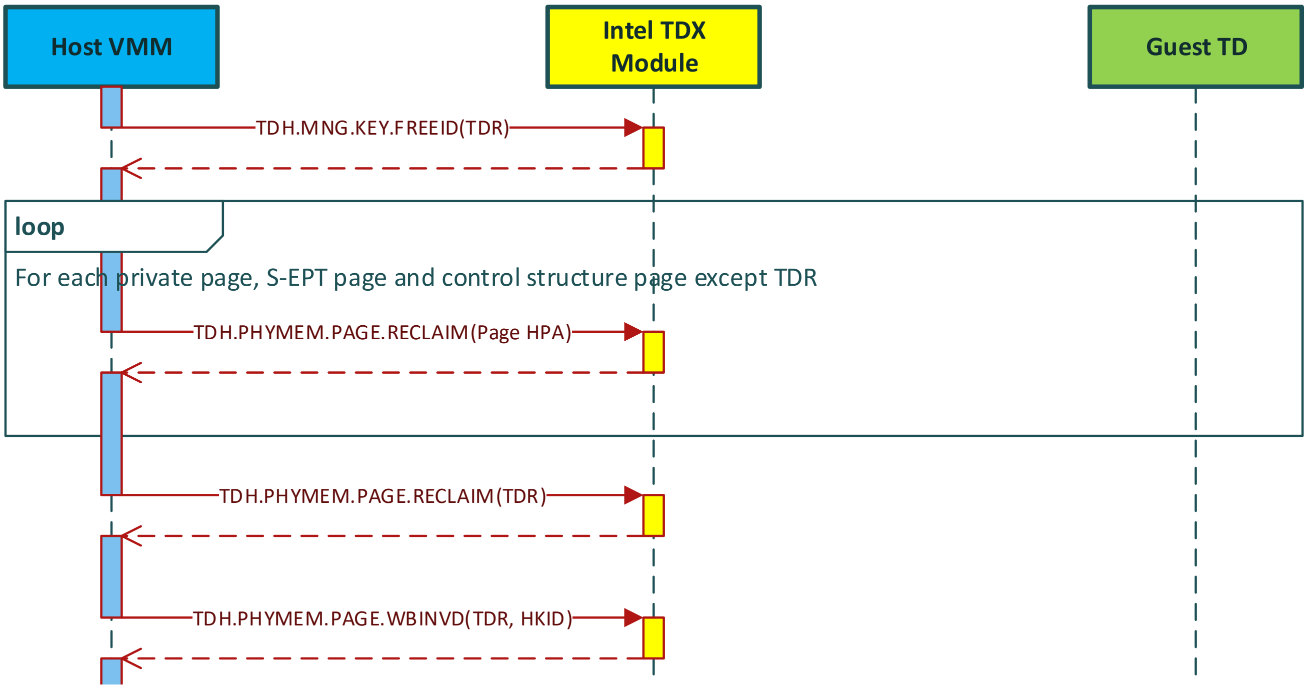

Host VMM
Intel TDX Module
Guest TD
TDH.MNG.KEY.FREEID(TDR)
loop
For each private page, S-EPT page and control structure page except TDR
TDH.PHYMEM.PAGE.RECLAIM(Page HPA)
TDH.PHYMEM.PAGE.RECLAIM(TDR)
TDH.PHYMEM.PAGE.WBINVD(TDR, HKID)

Figure 3.8: Typical TD Destruction Sequence Step B: Resource Reclamation

Table 3.14: Typical TD Destruction Sequence

	Step		Description	SEAMCALL Leaf Functions
A	**TD Stopping and Flushing Out**	1	The host VMM selects a TD to destroy. It sends a virtual interrupt to the TD to shut down gracefully.	TDH.VP.FLUSH TDH.MNG.VPFLUSHDONE TDH.PHYMEM.CACHE.WB
		2	The host VMM broadcasts inter-processor interrupts (IPIs) and must ensure TD exit on all logical processors.	
		3	The host VMM calls the TDH.VP.FLUSH function on all LPs associated with a TD VCPU to flush the TLBs and cached TD VMCS associated with a TD VCPU on those LPs.	
		4	The host VMM calls the TDH.MNG.VPFLUSHDONE function. It checks that above step executed for all the TD's VCPUs are associated with an LP.	
		5	The host VMM calls the TDH.PHYMEM.CACHE.WB function on each package to write back to memory the TD contents from all caches. TDH.PHYMEM.CACHE.WB is interruptible by external events. The host VMM should restart it if it indicates it was interrupted, until successfully completed.	
		6	At this point, no address translations or cache lines may exist for this TD except for the TDR page.	
B	**Resource Reclamation**	1	The host VMM calls the TDH.MNG.KEY.FREEID function. It marks the HKID used by the TD as available for other TDs.	TDH.MNG.KEY.FREEID TDH.PHYMEM.PAGE.RECLAIM TDH.PHYMEM.PAGE.WBINVD
		2	For each physical page in TDMR allocated to the TD (TD private pages, Secure EPT pages, and control structures except TDR), the host VMM calls the TDH.PHYMEM.PAGE.RECLAIM function to mark the page as free.	
		3	The host VMM calls the TDH.PHYMEM.PAGE.RECLAIM function to mark the TDR page as free. The function checks that all other TD physical pages have been reclaimed before.	
		4	Before allocating the reclaimed TDR physical page to any use, the host VMM calls TDH.PHYMEM.PAGE.WBINVD to flush its cache lines.	

SECTION 2:
INTEL TDX MODULE ARCHITECTURE SPECIFICATION

4. UPDATED: Intel TDX Module Lifecycle: Enumeration, Initialization and Shutdown

This chapter discusses the design of the Intel TDX module life cycle: how its capabilities are enumerated by the host VMM, how it is initialized, how it is configured and how it is shut down.

4.1. UPDATED: Overview

4.1.1. UPDATED: Initialization and Configuration Flow

The Intel TDX module initialization and configuration typically happens as described below:

Table 4.1: UPDATED: Typical Intel TDX Module Enumeration, Initialization and Configuration Sequence

	Step	Intel TDX Function	Description
1	CMR Configuration & Checking	N/A	BIOS configures convertible memory regions (CMRs); MCHECK checks them and securely stores the information.
2	Intel TDX Module Loading	N/A	OS/VMM launches the NP-SEAMLDR ACM to load the P-SEAMLDR, and then calls the P-SEAMLDR module to install the Intel TDX module.
2	Intel TDX Module Loading	N/A	OS/VMM launches the SEAMLDR ACM which loads the Intel TDX module.
3	Global Initialization	TDH.SYS.INIT	The host VMM calls TDH.SYS.INIT once. This function performs global initialization of the Intel TDX module.
4	LP Initialization	TDH.SYS.LP.INIT (each LP)	The host VMM calls TDH.SYS.LP.INIT once on each logical processor. This function performs LP-scope, core-scope and package-scope initialization, checking and configuration of the platform and the Intel TDX module.
5	Enumeration	TDH.SYS.RD/RDALL or TDH.SYS.INFO	The host VMM calls TDH.SYS.RD/RDALL or TDH.SYS.INFO to retrieve Intel TDX module information and convertible memory (CMR) information.
6	Global Configuration	TDH.SYS.CONFIG	The host VMM calls TDH.SYS.CONFIG once, providing a set of configuration parameters including a table of TDMRs. This function performs global configuration of the Intel TDX module.
7	Cache Flush	N/A	The host VMM flushes any MODIFIED cache lines that may exist for the PAMT ranges, using, e.g., WBINVD on each package.
8	Key Configuration	TDH.SYS.KEY.CONFIG (each package)	The host VMM calls TDH.SYS.KEY.CONFIG once on each package. This function configures the Intel TDX global private key on the hardware.
9	Intel TDX module is available	Any	Once TDH.SYS.KEY.CONFIG has executed successfully on all packages, any Intel TDX function may be executed on any LP.
10	TDMR and PAMT Initialization	TDH.SYS.TDMR.INIT (multiple)	The host VMM calls TDH.SYS.TDMR.INIT in a loop, gradually initializing the PAMT structure for each TDMR.
11	Memory is available	Any	Once each 1GB block of TDMR has been initialized by TDH.SYS.TDMR.INIT, it can be used to hold TD-private pages.

4.1.2. UPDATED: Intel TDX Module Lifecycle State Machine

The Intel TDX lifecycle state machine helps track the module's life cycle through the initialization sequence and shutdown.

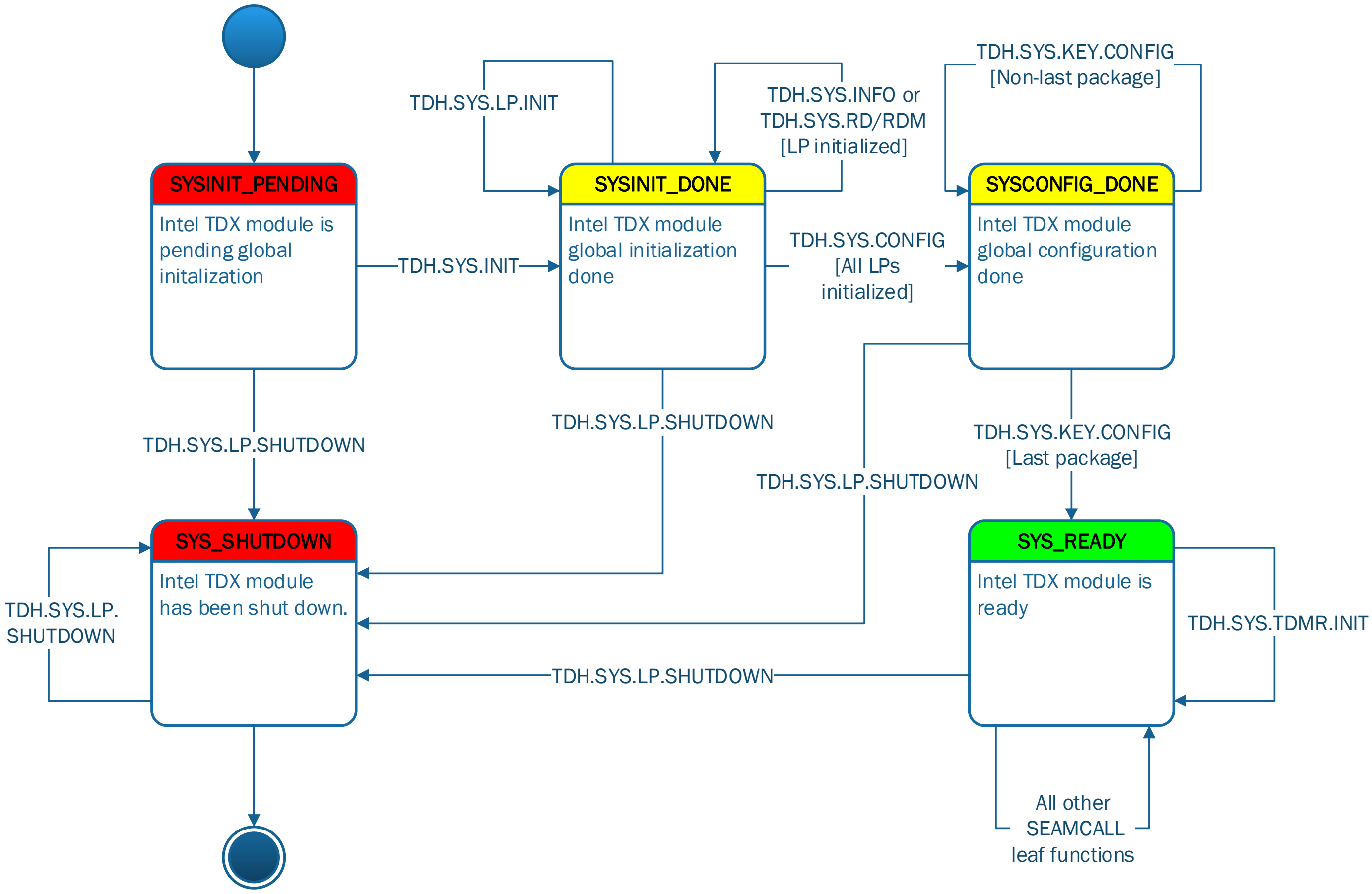

SYSINIT_PENDING
Intel TDX module is pending global initalization
TDH.SYS.INIT
TDH.SYS.LP.INIT
SYSINIT_DONE
Intel TDX module global initialization done
TDH.SYS.INFO or TDH.SYS.RD/RDM [LP initialized]
TDH.SYS.CONFIG [All LPs initialized]
TDH.SYS.KEY.CONFIG [Non-last package]
SYSCONFIG_DONE
Intel TDX module global configuration done
TDH.SYS.LP.SHUTDOWN
TDH.SYS.LP.SHUTDOWN
TDH.SYS.KEY.CONFIG [Last package]
TDH.SYS.LP.SHUTDOWN
SYS_SHUTDOWN
Intel TDX module has been shut down.
TDH.SYS.LP. SHUTDOWN
SYS_READY
Intel TDX module is ready
TDH.SYS.TDMR.INIT
TDH.SYS.LP.SHUTDOWN
All other SEAMCALL leaf functions

Figure 4.1: Intel TDX Module Lifecycle State Machine

4.1.3. Platform Compatibility and Configuration Checking

4.1.3.1. *Overview*

The Intel TDX module is built assuming a certain set of core and platform features. Most platform configuration required to support the Intel TDX module is checked by MCHECK. However, some configuration is designed to be checked by the Intel TDX module. During the initialization process, the Intel TDX module is designed to check that the platform on which it is running is compatible with this core and platform feature set and/or that the same set of features is provided across the platform. Some of the checks are done per core, and some are done per package. Most of the details are part of the Intel TDX module detailed design.

4.1.3.2. *MSR Sampling and Checks*

TDH.SYS.INIT reads and checks the contents of some MSRs. In many cases, the MSR value read by TDH.SYS.INIT is also checked for consistency (i.e., having the same values) by TDH.SYS.LP.INIT. In other cases, TDH.SYS.LP.INIT may perform additional checks.

4.1.3.3. *UPDATED: CPUID Sampling, Checks and Enumeration*

Note: CPUID virtualization is described in 11.8.

The TDH.SYS.INIT and TDH.SYS.LP.INIT functions sample CPUID leaf and sub-leaf return values. This is intended to check compatibility with the Intel TDX module and with any guest TD operation. If any of these checks fail, Intel TDX module initialization is designed to fail.

The TDH.SYS.RD, TDH.SYS.RDALL and TDH.SYS.INFO functions may be called by the host VMM to enumerate the directly configurable and allowable CPUID fields.

4.1.4. Physical Memory Configuration Overview

Configuration of the physical memory available to the Intel TDX module (TDMRs) and its associated metadata (PAMT arrays) is done using the TDH.SYS.CONFIG function.

4.1.4.1. *Intel TDX ISA Background: Convertible Memory Ranges (CMRs)*

A 4KB memory page is defined as **convertible** if it can be used to hold an Intel TDX private memory page or any Intel TDX control structure pages while helping guarantee Intel TDX security properties (i.e., if it can be **converted** from a Shared page to a Private page).

Convertible Memory Ranges (**CMRs**) are defined as physical address ranges that are declared by BIOS, and checked by MCHECK, to hold only convertible memory pages.

CMRs have the following characteristics:

- CMR configuration is "soft" – no hardware range registers are used.
- Each CMR defines a single contiguous physical address range.
- All the memory within each CMR is convertible, and it must comply with the rules checked by MCHECK.
- Each CMR has its own size. CMR size is a multiple of 4KB, and it is **not** required to be a power of two.
- CMRs cannot overlap with each other.
- CMRs must reside within the effective physical address range of the platform (after taking into account the most significant PA bits stolen for Key IDs).
- CMRs are configured at platform scope (no separate configuration per package).
- The maximum number of CMRs is implementation specific. It is not explicitly enumerated; it is deduced from Family/Model/Stepping information provided by CPUID.
    - The maximum number of CMRs is 32.
- CMRs are available on systems with TDX ISA capabilities as enumerated by the IA32_MTRRCAP.SEAMRR bit.
- CMR configuration is checked by MCHECK and cannot be modified afterwards.

MCHECK stores the CMR table in a pre-defined location in SEAMRR's SEAMCFG region so it can be read later and trusted by the Intel TDX module.

4.1.4.2. *TDMRs and PAMT Arrays Configuration*

TDMRs and PAMTs are described in 8.1. This section provides an overview of their configuration and their relationships to CMRs.

4.1.4.2.1. Background: Reserved Areas within TDMRs

As described in 8.1, the Intel TDX module physical memory management is done using PAMT Blocks – each holding the metadata of a 1GB block of TDMR. This implies that TDMR granularity must be 1GB.

However, there is a requirement for the host VMM to be able to allocate memory at granularities smaller than 1GB. This is especially important in systems that have a relatively small amount of memory.

To support the two requirements above, the Intel TDX module's design allows arbitrary reserved areas within TDMRs. Reserved areas are still covered by PAMT. However, during initialization their respective PAMT entries are marked with a PT_RSVD page type, so pages in reserved areas are not used by the Intel TDX module for allocating privately encrypted memory pages (but they can be used for PAMT areas, see below).

Only the non-reserved parts of a TDMR are required to be inside CMRs.

4.1.4.2.2. Background: Three PAMT Areas

As described in 8.1, a logical PAMT Block is composed of 1 PAMT_1G entry, 512 PAMT_2M entries and 512^2 PAMT_4K entries. Thus, the overall size of a PAMT Block, and as a result of the whole PAMT, is not a power of 2.

However, the host VMM may only be able to allocate memory buffers for PAMT in sizes that are a power of 2.

To enable this, buffers for PAMT_1G entries, PAMT_2M entries and PAMT_4K entries are allocated separately. As a result, if the host VMM allocates a TDMR whose size is a power of 2, its three respective PAMT areas will also have sizes that are a power of 2.

PAMT areas are required to be inside CMRs because PAMT is encrypted with a private HKID.

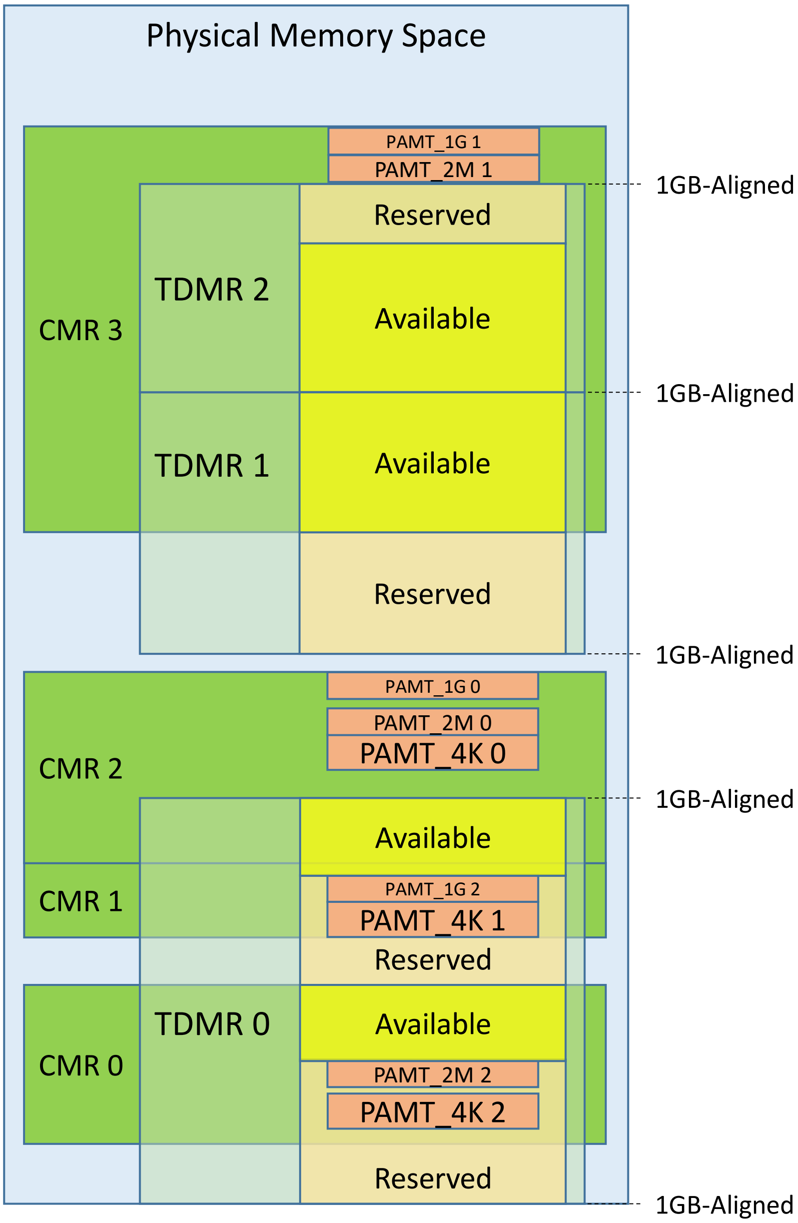


Figure 4.2: Example of Convertible Memory Ranges (CMRs) vs. Trust Domain Memory Regions (TDMRs)

4.1.4.2.3. UPDATED: Configuration Rules

In addition to the rules described in 8.1, the following rules apply to TDMR configuration as related to CMRs:

- Any non-reserved 4KB page within a TDMR must be convertible – i.e., it must be within a CMR.
- Reserved areas within a TDMR need not be within a CMR.

Three PAMT areas must be configured for each TMDR – one for each physical page size controlled by PAMT:

- Area for PAMT_4K entries
- Area for PAMT_2M entries
- Area for PAMT_1G entries

PAMT areas have the following attributes:

- A PAMT area size is directly proportional to the TDMR with which it is associated. The size ratio is enumerated by TDH.SYS.RD/RDALL or TDH.SYS.INFO. Note that the size ratio may be different for each of the 3 PAMT array types.
- A PAMT area must reside in convertible memory – i.e., each PAMT area page must be a CMR page.
- PAMT areas must not overlap with TDRM non-reserved areas; however, they may reside within TDMR reserved areas (as long as these are convertible).
- PAMT areas must not overlap with each other.

4.2. Intel TDX Module Initialization Interface

4.2.1. Global Initialization: TDH.SYS.INIT

TDH.SYS.INIT is intended to globally initialize the Intel TDX module. It works as follows:

1. Initialize Intel TDX module global data.
2. Sample and check platform features that need to be checked for platform-wide compatibility – i.e., the Intel TDX module supports several options, but they must be the same across platform. These are later checked on each LP.
3. Sample and check the platform configuration on the current LP. For example, TDH.SYS.INIT samples SMRR and SMRR2, checks they are locked and do not overlap any CMR, and stores their values to be checked later on each LP.
4. Set the system state to SYSINIT_DONE.

For a detailed description of TDH.SYS.INIT, see the [TDX Module ABI].

4.2.2. LP-Scope Initialization: TDH.SYS.LP.INIT

TDH.SYS.LP.INIT is intended to perform LP-scope, core-scope and package-scope initialization of the Intel TDX module. It can be called only after TDH.SYS.INIT completes successfully, and it can run concurrently on multiple LPs. At a high level, TDH.SYS.LP.INIT works as follows:

1. Do a global EPT flush (INVEPT type 2).
2. Initialize Intel TDX module LP-scope data.
3. Check features and configuration compatibility and uniformity – once per LP, core or package, depending on the scope of the checked feature or configuration:
   3.1. Check features compatibility with the Intel TDX module.
   3.2. Check configuration uniformity.

For a detailed description of TDH.SYS.LP.INIT, see the [TDX Module ABI].

4.2.3. UPDATED: TDX Module Enumeration: TDH.SYS.RD/RDALL and TDH.SYS.INFO

Once an LP has been initialized, the host VMM can call TDH.SYS.RD, TDH.SYS.RDALL or TDH.SYS.INFO on that LP to help enumerate the Intel TDX module capabilities and platform configuration.

TDH.SYS.RD and TDH.SYS.RDALL are the recommended enumeration methods. They enable the host VMM to read the values of TDX module global metadata fields, enumerating the TDX module capabilities. The list of fields is described in the [TDX Module ABI].

To read all host readable TDX Module fields, the host VMM can invoke TDH.SYS.RDALL. This function returns the information as a metadata list.

To read a single TDX Module field, TDH.SYS.RD can be invoked. It returns the next host-readable field identifier, thus it can also be used to enumerate the TDX Module by calling it in a loop, starting from field identifier value of 0, until it returns a next field identifier value of 0.

TDH.SYS.INFO is provided for backward compatibility with previous TDX module versions:

- Intel TDX module capabilities are enumerated in the returned TDSYSINFO_STRUCT (see the [TDX Module ABI]).
- Convertible Memory Ranges (CMRs), as previously set by BIOS and checked by MCHECK, are enumerated in the returned CMR_INFO table.

For a detailed description of interface functions and metadata fields, see the [TDX Module ABI].

4.2.4. Global Configuration: TDH.SYS.CONFIG

After performing global and LP-scope initialization, the host VMM can call TDH.SYS.CONFIG to globally configure the Intel TDX module, providing the following information:

- **TDMR and PAMT Table,** where each entry contains a TDMR base address, size and corresponding PAMT reserved area base address and size. Refer to 8.1 for definition of TDMRs.
- The **HKID** to be used by the Intel TDX module for its global private key, used for encrypting PAMT and TDRs.

For a detailed description of the table format (TDMR_INFO) and TDH.SYS.CONFIG, see the [TDX Module ABI].

4.2.5. Package-Scope Key Configuration: TDH.SYS.KEY.CONFIG

After performing global configuration, the host VMM calls TDH.SYS.KEY.CONFIG to perform package-scope configuration of the Intel TDX module’s global private key on the hardware.

For a detailed description of TDH.SYS.KEY.CONFIG, see the [TDX Module ABI].

4.3. TDMR and PAMT Initialization

TDMR and PAMT initialization procedure is designed to be performed **during VMM run-time**, after VMM boot. The host VMM should be able to work normally while initialization takes place, at any time using memory that has already been initialized. At a high level, TDMR initialization has the following characteristics:

- Initialization is performed gradually.
- Initialization function TDH.SYS.TDMR.INIT adheres to the latency rules of most Intel TDX functions – i.e., they take no more than a predefined number of clock cycles.
- Initialization function TDH.SYS.TDMR.INIT **can run concurrently on multiple LPs** if each concurrent flow initializes a **different TDMR**.
- After each 1GB page of a TDMR has been initialized, that 1GB page becomes available for use by any Intel TDX function that creates a private TD page or a control structure page – e.g., TDH.MEM.PAGE.ADD, TDH.VP.ADDCX, etc.

For each TDMR, the VMM should execute a loop of **TDH.SYS.TDMR.INIT** providing the TDMR start address (at 1GB granularity) as an input.

TDH.SYS.TDMR.INIT initializes an (implementation-defined) number of PAMT entries. The maximum number of PAMT entries to be initialized is designed to avoid latency issues. Initialization uses direct writes (MOVDIR64B).

Once the PAMT for each 1GB block of TDMR has been fully initialized, TDH.SYS.TDMR.INIT marks that 1GB block as ready for use; that means 4KB pages in this 1GB block may be converted to private pages – e.g., by TDH.MEM.PAGE.ADD. This can be done concurrently with adding and initializing other TDMRs.

For a detailed description of TDH.SYS.TDMR.INIT, see the [TDX Module ABI].

4.4. Intel TDX Module Shutdown

4.4.1. Shutdown Initiated by the Host VMM (as Part of Module Update)

The host VMM can initiate Intel TDX module shutdown at any time by calling the TDH.SYS.LP.SHUTDOWN function. This is intended for use as part of reloading the Intel TDX module without going through a warm or cold reset sequence. TDH.SYS.LP.SHUTDOWN is designed to set state variables to block all SEAMCALLs on the current LP and all SEAMCALL leaf functions except TDH.SYS.LP.SHUTDOWN on the other LPs. SEAMLDR, when instructed to reload a new Intel TDX module image, can check that TDH.SYS.SHUTDOWN has been executed on all LPs.

4.4.2. Shutdown Initiated by a Fatal Error

By design, fatal errors during Intel TDX module execution cause an immediate SEAM shutdown. Subsequent SEAMCALLs on any LP fail with a VMfailInvalid indication (RFLAGS.CF set to 1). This situation can only be recovered by a platform reset.

5. Key Management

5.1. Objectives

The main goal of Intel TDX key management is to enable the VMM to perform the following:

- Manage HKID space as a limited platform resource, assigning HKIDs to TDs and reclaiming them as required.
- Enable the Intel TDX module to use a global ephemeral key for encrypting its data (e.g., PAMT).
- Enable each TD to use its own ephemeral key.

The Intel TDX interface functions are designed to provide the required building blocks and help assure that software cannot perform operations that are not compliant with TDX security objectives, as follows:

1. Help assure that only HKID values that have been configured for TDX private memory encryption keys can be assigned to TDs, and that those HKID values cannot be used by non-TD software or devices.
2. Prevent assignment of the same HKID to more than one TD.
3. At the time an HKID is assigned to a TD, there must be no modified cache lines – at any level, for any core, on any package – for that HKID. All such cache lines that may have held modified data have been written to memory (if required). Note that this requirement applies only to TDX private HKID and not to legacy MKTME HKIDs.
4. TD memory may be accessed, and the TD may run, only when the following conditions are met:
   4.1. An HKID has been assigned for the TD's ephemeral key.
   4.2. The encryption key has been configured for all the TD's ephemeral HKID, on all crypto engines, on all packages.

5.2. Background: HKID Space Partitioning

Since the same MKTME encryption engines and the same set of encryption keys are used for legacy MKTME operation and for TDX operation, TDX ISA enables the enumeration and partitioning of the activated HKID space between the two technologies. As designed, the encryption keys and their associated HKIDs are divided into three ranges, as shown in Table 5.1 below. The values of NUM_MKID_KEYS and NUM_TDX_PRIV_KEYS are read from the IA32_MKTME_KEYID_PARTITIONING MSR (0x87).

Private HKIDs and private keys are designed to be fully controlled by the Intel TDX module and are the subject of this chapter.

Table 5.1: HKID Space Partitioning

	HKID	**Key**
Shared HKIDs	0	Legacy TME key, shared
	1	Legacy MKTME key #1
	2	Legacy MKTME key #2
	…	…
	NUM_MKID_KEYS	Last legacy MKTME key
Private HKIDs	NUM_MKID_KEYS + 1	Private key of a specific TD
	NUM_MKID_KEYS + 2	Private key of a specific TD
	NUM_MKID_KEYS + 3	Private key of a specific TD
	…	…
	NUM_MKID_KEYS + NUM_TDX_PRIV_KIDS	Private key of a specific TD

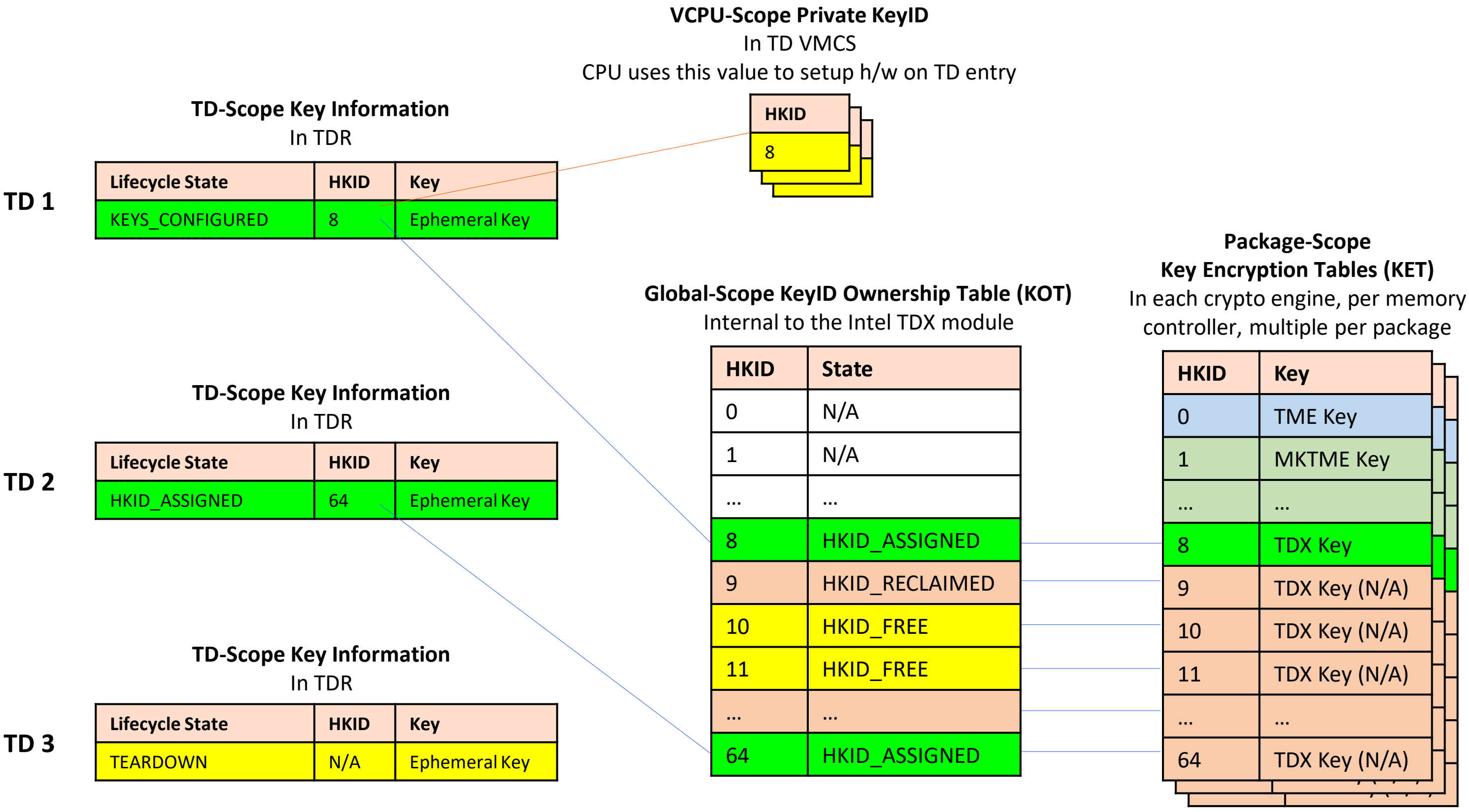
5.3. Key Management Tables

The CPU and the Intel TDX module maintain several tables for key management. No table is intended to be directly accessible by software; the tables are used by the Intel TDX functions. The tables help the Intel TDX module track the proper operation of the software and help achieve the Intel TDX security objectives.

Table 5.2: Key Management Tables

Table	Scope	Description
Key Encryption Table (KET)	Package	KET is an abstraction of the CPU micro-architectural hardware table for configuring the memory encryption engines. The KET is indexed by HKID. All crypto engines on a package are configured the same way. KET is part of the legacy MKTME architecture. Intel TDX ISA partitions KET to shared and private ranges, as described in 5.2 above. • A KET entry in private HKIDs range is configured per package by the host VMM using the SEAMCALL(TDH.MNG.KEY.CONFIG) function. • A KET entry in the shared HKID range is configured by software per package directly, using the PCONFIG instruction.
KeyID Ownership Table (KOT)	Platform	KOT is an Intel TDX module hidden table for managing the TDX HKIDs inventory. It is used for assigning HKIDs to TDs, revoking HKIDs from TDs and controlling cache flush. KOT is indexed by HKID.
TD Key Management Fields	TD	TD-scope key management fields are held in TDR. They include the key state, ephemeral private HKID and key information, and a bitmap for tracking key configuration.

Figure 5.1 below provides an abstract, high-level picture of how the tables are related. Detailed discussion is provided in the following sections.


VCPU-Scope Private KeyID
In TD VMCS
CPU uses this value to setup h/w on TD entry
TD-Scope Key Information
In TDR
TD 1
TD 2
TD 3
Global-Scope KeyID Ownership Table (KOT)
Internal to the Intel TDX module
Package-Scope
Key Encryption Tables (KET)
In each crypto engine, per memory controller, multiple per package

HKID
8

TD 1

Lifecycle State	HKID	Key
KEYS_CONFIGURED	8	Ephemeral Key

TD 2

Lifecycle State	HKID	Key
HKID_ASSIGNED	64	Ephemeral Key

TD 3

Lifecycle State	HKID	Key
TEARDOWN	N/A	Ephemeral Key

HKID	State
0	N/A
1	N/A
…	…
8	HKID_ASSIGNED
9	HKID_RECLAIMED
10	HKID_FREE
11	HKID_FREE
…	…
64	HKID_ASSIGNED

HKID	Key
0	TME Key
1	MKTME Key
…	…
8	TDX Key
9	TDX Key (N/A)
10	TDX Key (N/A)
11	TDX Key (N/A)
…	…
64	TDX Key (N/A)

Figure 5.1: Overview of the Key Management State at TD-Scope, LP-Scope, Package-Scope and Global-Scope

5.4. Combined Key Management State

Key management state is composed of two state variables:

- **Per-HKID KOT Entry State** is designed to control how the inventory of private HKIDs is managed using the KOT.
- **Per-TD Life Cycle State** is designed, among other things, to control how TD keys are configured on the hardware and the process of shutting down a TD.

The combined key management state is intended to affect whether the TD private memory is accessible, whether its contents may be cached, whether private GPA-to-HPA address translations are allowed and whether such translations may be cached.

Table 5.3 below lists the designed combined key management state values and their meaning. Figure 5.2 below shows a simplified diagram of the combined key state. Refer also to the key management sequences described in 5.5.

Table 5.3: Combined TD Key Management States

TD Life Cycle State	KOT Entry (HKID) State	Private Memory Access		S-EPT Translations		Comments
		New	Cached	New	Cached	
N/A	HKID_FREE	No	No	No	No	HKID not assigned to TD
TD_HKID_ASSIGNED	HKID_ASSIGNED	No	No	No	No	TD private key not configured
TD_KEYS_CONFIGURED		TD	TD	TD	TD	TD build and execution
TD_BLOCKED	HKID_FLUSHED	No	TD	No	No	TD private memory access is blocked, TD may not run
TD_TEARDOWN	N/A (HKID_FREE)	No	No	No	No	TD has no HKID
N/A	HKID_RESERVED	Global	Global	N/A	N/A	HKID for Intel TDX global data

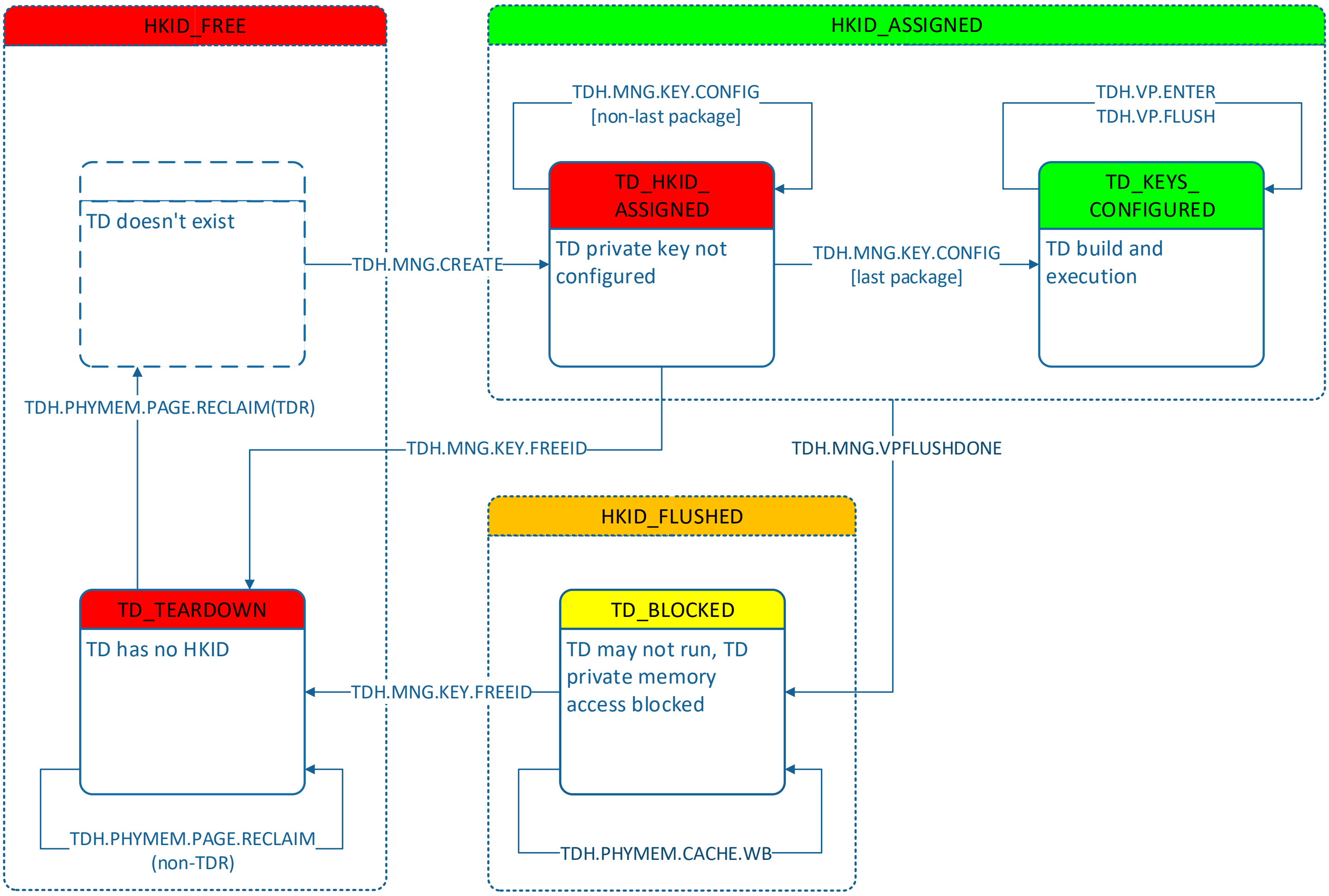


Figure 5.2: Simplified Combined TD Key Management State Diagram

Chapter 7 discusses TD life cycle management and zooms-in into the TD_KEYS_CONFIGURED state, detailing its secondary sub-states that control TD operation and TD migration.

5.5. Key Management Sequences

5.5.1. Intel TDX Module Initialization: Setting an Ephemeral Key and Reserving an HKID for Intel TDX Data

This sequence is described as part of the Intel TDX module initialization sequence in 4.1.1.

5.5.2. TD Creation, Keys Assignment and Configuration

This sequence is intended to be used by the host VMM to create a new TD, select HKIDs from the global pool in KOT and assign them to the TD, and configure the TD keys on the hardware.

Refer also to the software flow discussion in 3.2.

Table 5.4: Typical TD Creation, Keys Assignment and Configuration (TD-Scope and KOT-Scope) Sequence

	Intel TDX Function	Scope	Execute On	Description
1	TDH.MNG.CREATE	TD	One LP	Assign the TD's private HKID.
2	TDH.MNG.KEY.CONFIG	TD	Each package and each TD key	Configure the TD's random ephemeral key on the package.

5.5.3. TD Keys Reclamation, TLB and Cache Flush

This sequence is intended to be used by the host VMM to reclaim the HKIDs assigned to a TD and return them to the global pool in KOT. At the end of this sequence, the HKIDs should be free to be assigned to another TD.

The cache flush operation is long. Since it is designed to run at global scope and is decoupled from any TD, the host VMM may choose to implement it in a lazy fashion, i.e., wait until a certain number of HKIDs in the KOT pool become RECLAIMED. This is especially important since TDH.PHYMEM.CACHE.WB operates on all cache lines regardless of HKID.

To avoid long latencies, TDH.PHYMEM.CACHE.WB is designed to be interruptible. The host VMM is expected to repeat the execution of this instruction until it returns a success indication.

Refer also to the software flow discussion in 3.4.

Table 5.5: Typical TLB and Cache Flush (TD-Scope and KOT-Scope) Sequence

	Intel TDX Function	Scope	Execute On	Description
As a preparation, the host VMM avoids any VCPU-specific SEAMCALL function (i.e., TDH.VP.ENTER, TDH.VP.INIT, TDH.VP.RD and TDH.VP.WR) and waits until no VCPU is running.				
1	TDH.VP.FLUSH	TD VCPU	One each LP associated with a TD VCPU	Flush the VCPU's TD VMCS to TDVPS memory, and flush the VCPU's TLB ASID.
2	TDH.MNG.VPFLUSHDONE	TD, KOT	One LP	Check all the VCPUs have been flushed.
3	TDH.PHYMEM.CACHE.WB	KOT	Each package or core[4]	Write back cache hierarchy, at least for the HKIDs marked as TLB_FLUSHED. The instruction execution time is long; it is interruptible by external events and may be restarted until completed.
4	TDH.MNG.KEY.FREEID	TD, KOT	One LP	Mark TD's HKID as FREE.

[4] Enumerated by CPU during Intel TDX module initialization, see 4.1.3.3.

6. UPDATED: TD Non-Memory State (Metadata) and Control Structures

This chapter discusses the guest TD control structures that hold non-memory state (metadata) and how they are intended to be used during the TD life cycle.

6.1. Overview

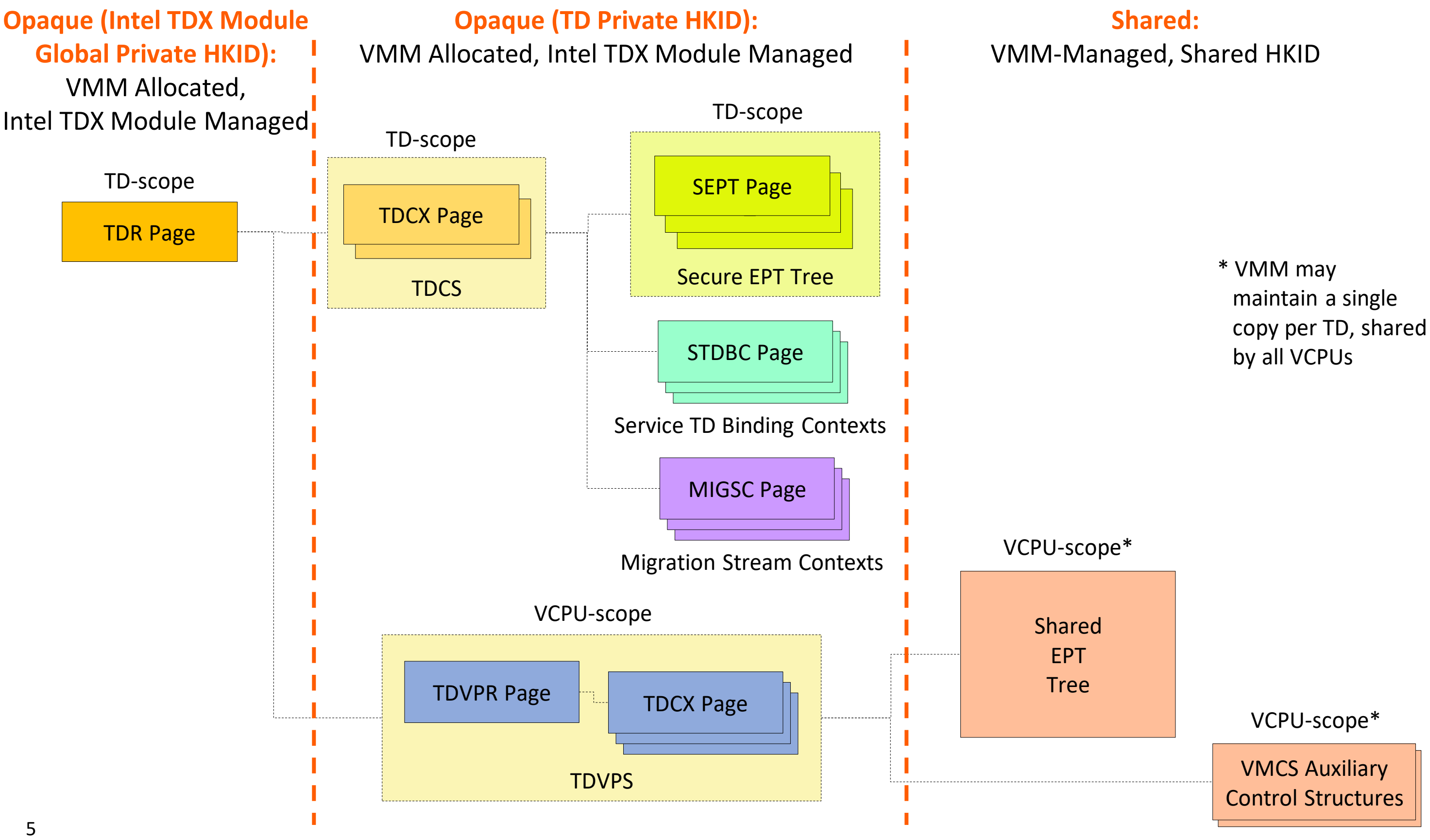

Opaque (Intel TDX Module Global Private HKID):
VMM Allocated,
Intel TDX Module Managed
TD-scope
TDR Page
Opaque (TD Private HKID):
VMM Allocated, Intel TDX Module Managed
TD-scope
TDCX Page
TDCS
TD-scope
SEPT Page
Secure EPT Tree
STDBC Page
Service TD Binding Contexts
MIGSC Page
Migration Stream Contexts
VCPU-scope
TDVPR Page
TDCX Page
TDVPS
Shared:
VMM-Managed, Shared HKID
* VMM may maintain a single copy per TD, shared by all VCPUs
VCPU-scope*
Shared EPT Tree
VCPU-scope*
VMCS Auxiliary Control Structures

Figure 6.1: UPDATED: Guest TD Control Structures Overview

All guest TD control structures reside in memory pages that are allocated by the host VMM from the pre-configured TDMRs. Guest TD control structure pages are addressable by the host VMM.

6.1.1. Opaque vs. Shared Control Structures

Control structures are divided to two classes:

- **Shared control structures** are intended to be directly managed by the host VMM and are encrypted with a shared HKID. The Intel TDX module architecture only describes the shared control structures that might directly impact its operation. The host VMM may hold additional control structures.
- **Opaque control structures** are not intended to be directly accessible to any software (except the Intel TDX module) or DMA. They are intended to be managed via Intel TDX module functions. Generally speaking, the host VMM is not aware of the exact format of opaque control structures. Opaque control structures' memory pages are intended to be encrypted with a private HKID.

6.1.2. Scope of Control Structures

Guest TD control structures have two possible scopes:

- **TD-scope control structures** are intended to apply for a guest TD as a whole.
- **TD VCPU-scope control structures** are intended to apply for a single virtual CPU of a guest TD.

6.2. TD-Scope Control Structures

TD-scope control structures include TDR and TDCS, discussed below, and Secure EPT, discussed in Chapter 9.

6.2.1. TDR (Trust Domain Root)

TDR is the root control structure of a guest TD. As designed, TDR is encrypted using the Intel TDX global private HKID. It holds a minimal set of state variables that enable guest TD control even during times when the TD's private HKID is not known, or when the TD's key management state does not permit access to memory encrypted using the TD's private key.

TDR occupies a single 4KB naturally aligned page of memory. It is designed to be the first TD page to be allocated and the last to be removed. Its physical address serves as a unique identifier of the TD, as long as any TD page or control structure resides in memory.

At a high level, TDR holds the following information:

- Fields designed to control guest TD build and teardown process.
- Fields designed to manage memory encryption keys.

6.2.2. UPDATED: TDCS (Trust Domain Control Structure)

TDCS is the main control structure of a guest TD. As designed, TDCS is encrypted using the guest TD's ephemeral private key. TDCS is a multi-page logical structure composed of multiple TDCX physical pages.

At a high level, TDCS holds the following information:

- Fields designed to control the TD operation as a whole (e.g., a counter of the number of VCPUs currently running).
- Fields designed to control the TD's execution control (debuggability, CPU features available to the TD, etc.).
- Fields related to TD measurement.
- EPTP: as designed, a pointer (HPA) to the TD's secure EPT root page and EPT attributes.
- MSR bitmaps, designed to be used by all the TD's VCPUs.
- As designed, the secure EPT root page.
- A page filled with zeros, designed to be used in cases where the Intel TDX module needs a read-only constant-0 page encrypted with the TD's private key.

TDCS may hold forward links to the following control structures:

- Secure EPT pages.
- Service TD Binding Context (STDBC) pages.
- Migration Stream Context (MIGSC) pages.

6.3. TD VCPU-Scope Control Structures

6.3.1. Trust Domain Virtual Processor State (TDVPS)

Trust Domain Virtual Processor State (TDVPS) is the root control structure of a TD VCPU. It helps the Intel TDX module control the operation of the VCPU, and holds the VCPU state while the VCPU is not running. TDVPS is a single logical control structure composed of multiple physical 4KB pages.

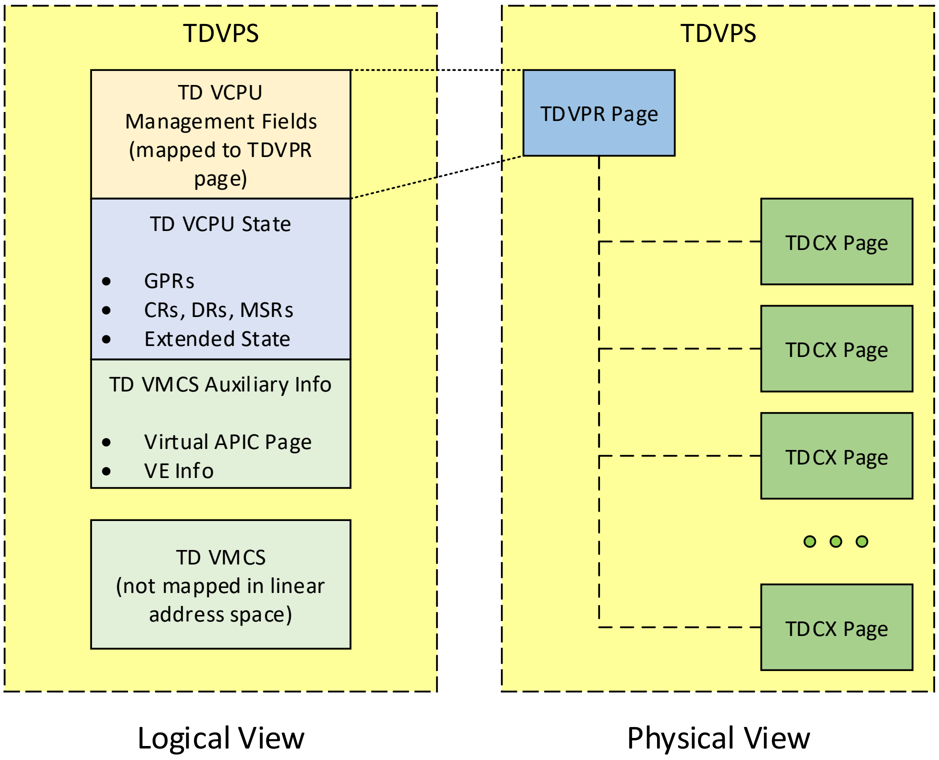


Figure 6.2: High Level Logical and Physical View of TDVPS

6.3.1.1. UPDATED: Physical View of TDVPS: TDVPR/TDCX

TDVPS is designed to be opaque to software and DMA access, accessible only by using the Intel TDX module functions. From the VMM perspective, TDVPS is composed of multiple 4KB pages, where each page may reside in arbitrary locations in convertible memory.

Trust Domain Virtual Processor Root (TDVPR) is the 4KB root page of TDVPS. Its physical address serves as a unique identifier of the VCPU (as long as it resides in memory).

Trust Domain Control structure eXtension (TDCX) 4KB pages extend TDVPR to help provide enough physical space for the logical TDVPS structure.

The TDVPR and TDCX pages are designed to be encrypted with the TD's ephemeral private key. They are addressable by the host VMM, which is responsible for allocating memory to hold them.

The required number of 4KB TDVPR/TDCX pages in TDVPS is enumerated to the VMM by the TDH.SYS.RD* or TDH.SYS.INFO function (see 4.2.3).

6.3.1.2. Logical View of TDVPS

Logically, TDVPS is organized as a single large data structure. At a high level, it is composed of the following parts:

VMX (with TDX ISA Extensions) Standard Control Structures

- TD VMCS
- TD VMCS auxiliary structures, such as virtual APIC page, virtualization exception information, etc. Note that MSR bitmaps are held as part of TDCS because they are meant to have the same value for all VCPUs of the same TD.

The TDX design does not require some of the VMX control structures (notably, the Shared EPT) to be protected. They are described below.

Proprietary Fields

- TD VCPU Management fields designed to manage the operation of the VCPU
- TD VCPU State fields designed to hold most of the VPCU state (except state that is saved to the TD VMCS) when the VCPU is not running

TDVPS organization and format are detailed in the [TDX Module ABI]..

Section 2: Introduction and Overview

6.3.2. Non-Protected Control Structures: Shared EPT and VMCS Auxiliary Control Structures

Several VMX control structures are directly managed and accessed by the host VMM. These control structures are pointed to by fields in the TD VMCS. The Intel TDX module checks that the pointers conform to the shared-access HPA semantics (see 18.2.1.1).

Non-protected control structures include:

- Shared EPT tree
- Posted interrupt descriptor

6.4. *UPDATED: TD Non-Memory State (Metadata) Access Functions*

As set of interface functions is provided to enable host VMM and guest TD access to TD non-memory state (metadata). These functions employ **metadata abstraction**, using field code to abstract the actual control structure format. The generic metadata access interface mechanisms are described in 18.4.

Table 6.1: TD Non-Memory State (Metadata) Single Field Access Functions

Side	Scope	Control Structures	Intel TDX Functions
Host VMM (SEAMCALL)	TD	TDR and TDCS	TDH.MNG.RD, TDH.MNG.WR
	VCPU	TDVPS (including TD VMCS)	TDH.VP.RD, TDH.VP.WR
Guest TD (TDCALL)	TD	TDR and TDCS	TDG.VM.RD, TDG.VM.WR
	VCPU	TDVPS (including TD VMCS)	TDG.VP.RD, TDG.VP.WR

Access to control structure fields using the provided interface functions (down to the bit granularity, if required) depends on whether the TD is debuggable (ATTRIBUTES.DEBUG bit is 1) or not.

In many cases, control structure field access means more than just reading or writing the field content. For example:

- When a field that contains an HPA is written, its value is checked not to overlap the SEAMRR range.
- In some cases, there may be inter-dependency between fields. When such fields are written, multiple checks may need to be done and some actions may need to be taken.
- For some fields, the internal format and/or value may be different than what is visible externally.

For details about the TDX module's metadata access interface, see 18.4.

6.5. *Concurrency Restrictions and Enforcement*

A general description of concurrency restriction is provided in 18.1.

Normally, exclusive or shared access is acquired, if needed, for the typically short duration of function flows. A TD VCPU execution is an exception case. Shared access to TDCS and TDVPS is acquired on TD Entry and released on TD Exit. This implies that SEAMCALL(TDH.VP.ENTER) function, all TDCALL functions, and asynchronous TD Exit have implicit shared access to TDCS and TDVPS.

This mechanism helps protect running VCPUs against concurrent functions that may try to change their governing control structures.

7. UPDATED: TD Life Cycle Management

This chapter discusses guest TD life cycle management.

7.1. TD Life Cycle State Machine

The TD Life Cycle state machine controls the overall TD build, run-time and destruction process. It operates in conjunction with the HKID state machine, as described in 5.4. Figure 7.1 below shows the TD Life Cycle state diagram.

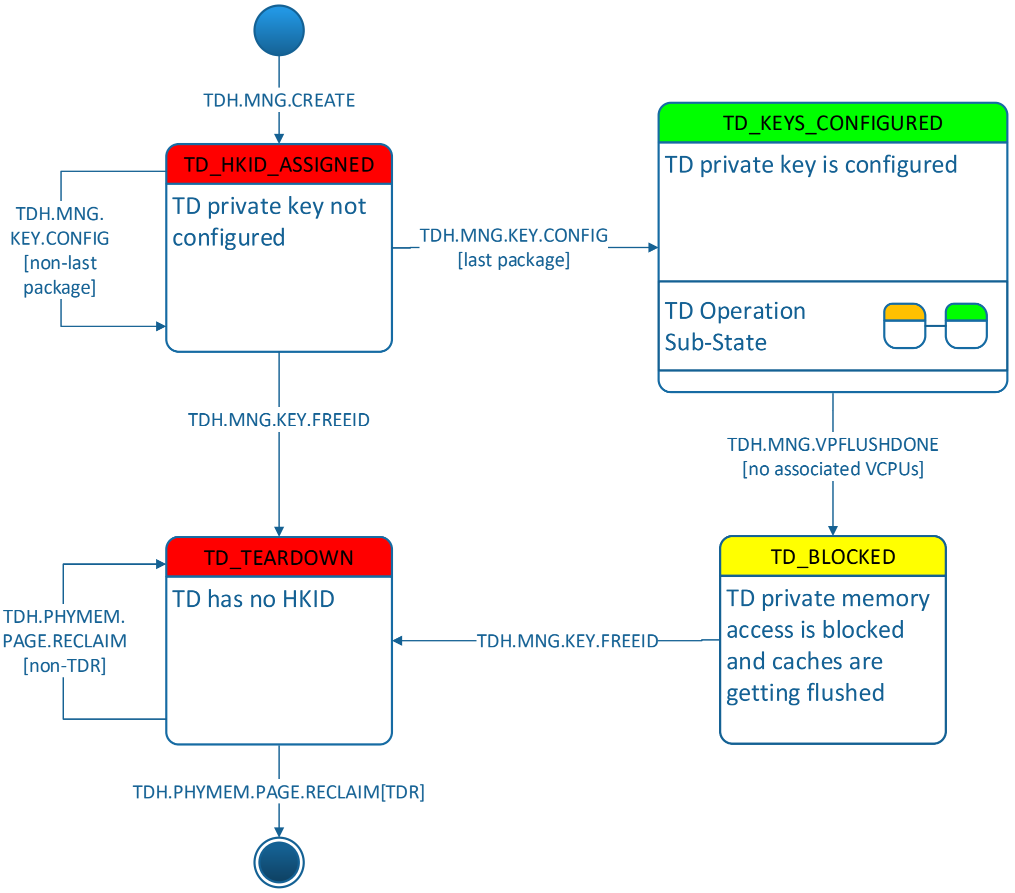

TDH.MNG.CREATE
TD_HKID_ASSIGNED
TD private key not configured
TDH.MNG. KEY.CONFIG [non-last package]
TDH.MNG.KEY.CONFIG [last package]
TD_KEYS_CONFIGURED
TD private key is configured
TD Operation Sub-State
TDH.MNG.KEY.FREEID
TDH.MNG.VPFLUSHDONE [no associated VCPUs]
TD_TEARDOWN
TD has no HKID
TDH.PHYMEM. PAGE.RECLAIM [non-TDR]
TDH.MNG.KEY.FREEID
TD_BLOCKED
TD private memory access is blocked and caches are getting flushed
TDH.PHYMEM.PAGE.RECLAIM[TDR]

Figure 7.1: High-Level TD Life Cycle State Diagram

Most of the TD lifetime is spent in the TD_KEYS_CONFIGURED state. Within that state, a secondary-level state machine controls the overall TD operation and migration.

7.2. UPDATED: OP_STATE: TD Operation Secondary-Level State Machine

The TD Operation state machine controls sub-states of the TD Life Cycle's TD_KEYS_CONFIGURED state. It shown in Figure 7.2 below. This document describes the baseline states: UNALLOCATED, UNINITIALIZED, INITIALIZED and RUNNABLE. Other states and transitions highlighted in red lines support TD migration and are described in the [TD Migration Spec].

Section 2: Introduction and Overview

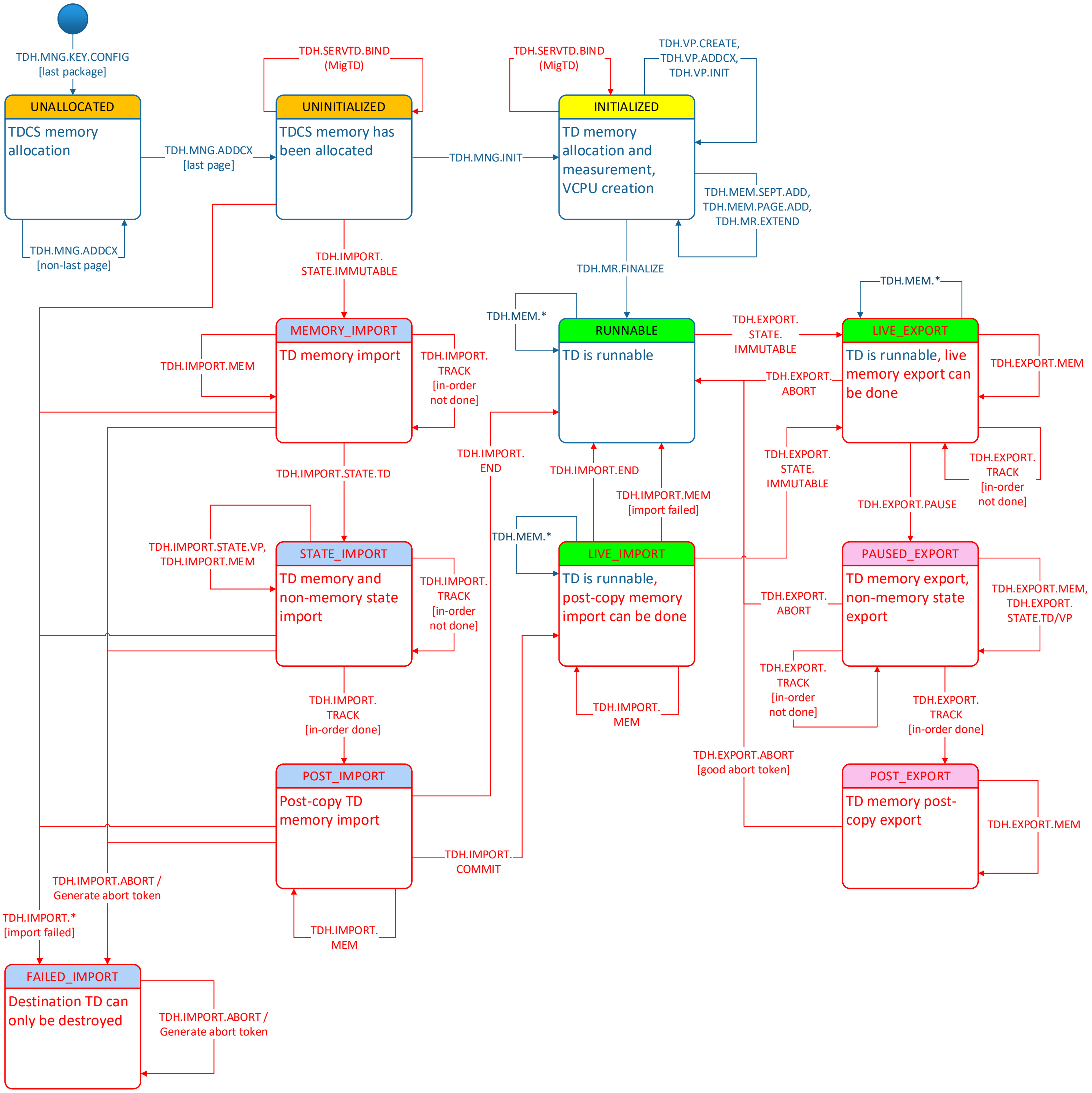


Figure 7.2: TD Operation State Machine (Sub-States of TD_KEYS_CONFIGURED)

7.3. TD Creation Sequence

The following sequence is intended to be used by the host VMM to create a new TD. Note that only the general aspects of TD creation are described here. Other aspects, such as key management, are described in other chapters.

Refer also to the software flow discussion in 3.2.

Table 7.1: Typical TD Creation Sequence

	Intel TDX Function	Inputs	Description
1	N/A	N/A	If any MODIFIED cache lines may exist for the physical pages to be written below (TDR, TDCS, Secure EPT root page), flush them to memory using, e.g., CLFLUSH (possibly on multiple LPs). This is required to avoid corruption due to cache line aliasing.
2	TDH.MNG.CREATE	TDR page PA	Create the TDR and generate the TD's random ephemeral key.
3	Multiple	See 5.5.2	Assign an HKID, and configure the TD's random ephemeral key on all packages, as described in 5.5.2.
4	TDH.MNG.ADDCX (multiple)	• Owner TDR PA • TDCX page PA	Run multiple times to add the required number of TDCX pages.
5	TDH.MNG.INIT	• Owner TDR PA • TD initialization parameters	Initialize the TD state in TDR and TDCS.
At this point the TD is initialized. Private memory pages can be added as described in Chapter 9. VCPUs can be created and initialized as described below.			

7.4. VCPU Creation and Initialization Sequence

VCPU creation and initialization is only allowed during TD build time.

The following sequence is intended to be used by the host VMM to create a new TD VCPU. After this sequence is done, the TD VCPU may be entered on an LP (assuming other conditions are met).

Refer also to the software flow discussion in 3.2.

Table 7.2: Typical TD VCPU Creation and Initialization Sequence

	Intel TDX Function	Inputs	Description
1	N/A	N/A	If any MODIFIED cache lines may exist for the physical pages to be written below (TDVPR, TDCX), flush them to memory (e.g., using CLFLUSH – possibly on multiple LPs). This is required to avoid corruption from cache line aliasing.
2	TDH.VP.CREATE	• TDVPR page PA • Owner TDR PA	Create the VCPU and its TDVPR page.
3	TDH.VP.ADDCX (multiple)	• TDCX page PA • Parent TDVPR PA	Run multiple times to add the required number of TDCX pages as an extension to a parent TDVPR.
4	TDH.VP.INIT	• TDVPR PA • VMM-provided identifier	Initialize the VCPU state.

	Intel TDX Function	Inputs	Description
5	TDH.VP.WR	• TDVPR page PA • Field code • New field value • Write mask	The host VMM typically writes one or more of the following TD VCPU's VMCS controls: • Shared EPTP • Posted-interrupts descriptor address, posted-interrupts notification vector and process posted interrupt • bus-lock detection • notification exiting and notify window For details, see the [TDX Module ABI].

7.5. TD Teardown Sequence

The following sequence is intended to be used by the host VMM to tear down a TD. Note that only the general aspects of TD teardown are described here. Other aspects, such as key management, are described in other chapters. See also the discussion of physical page reclamation in 8.5.

Refer also to the software flow discussion in 3.4.

Table 7.3: Typical TD Teardown Sequence

	Intel TDX Function	Inputs	Description
1	Multiple	See 5.5.3	Reclaim the HKID, and flush TLB and cache, as described in 5.5.3.
2	TDH.PHYMEM.PAGE.RECLAIM (multiple)	TD page or control structure PA	Remove all TD private pages and control structure pages, and mark them as PT_NDA in the PAMT.
3	TDH.PHYMEM.PAGE.RECLAIM	TDR PA	Remove the TDR page, and mark it as PT_NDA in the PAMT.
4	TDH.PHYMEM.PAGE.WBINVD	TDR PA	Flush MODIFIED cache lines: this is required to avoid corruption due to cache line aliasing. Note that all cache lines for all other TD pages must have been flushed before the TDR page was reclaimed.

8. Physical Memory Management

This chapter describes how the Intel TDX module manages memory as a set of physical pages.

8.1. UPDATED: Trust Domain Memory Regions (TDMRs) and Physical Address Metadata Tables (PAMTs)

Trust Domain Memory Region (TDMR) is defined as a range of convertible memory pages. TDMRs are set by the host VMM, based on the CMR information previously checked by MCHECK.

Each TDMR is defined as controlled by a (logically) single **Physical Address Metadata Table (PAMT)**. The PAMT structure is discussed in 8.3 below. PAMT tables reside in VMM-allocated memory, and they are designed to be encrypted with the Intel TDX global private HKID. The required size of PAMT memory, as a function of TDMR size, is enumerated to the VMM by TDH.SYS.RD/RDALL or TDH.SYS.INFO.

Typically, after the host VMM initializes the Intel TDX module (TDH.SYS.INIT and TDH.SYS.LP.INIT), it configures the TDMRs and their respective PAMTs using TDH.SYS.CONFIG. It then would gradually initialize the TDMRs using TDH.SYS.TDMR.INIT. For a detailed description of the typical Intel TDX module initialization and configuration sequence, see Chapter 4.

8.2. UPDATED: TDMR Details

The following list includes definitions of the characteristics of a TDMR:

- TDMR configuration is "soft" – no hardware range registers are used.
- Each TDMR defines a single physical address range.
- Each TDMR has its own size which must be a multiple of 1GB. TDMR size is **not** required to be a power of two.
- A TDMR must be aligned on 1GB.
- TDMRs cannot overlap with each other.
- TDMRs may contain reserved areas. This effectively allows the host VMM to flexibly configure TDMRs based on the VMM’s own consideration of system memory allocation – without being impacted by the 1GB granularity of the TDMR size.
  - A reserved area must be aligned on 4KB, and its size must be a multiple of 4KB.
  - The number of reserved areas that may be configured per TDMR is enumerated by TDH.SYS.RD/RDALL or TDH.SYS.INFO.
- TDMR memory, except for reserved areas, must be convertible as checked by MCHECK (i.e., every TDMR page must reside within a CMR).
- There is no requirement for TMDRs to cover all CMRs.
- TDMRs are configured at platform scope (no separate configuration per package).
- The maximum number of TDMRs is Intel TDX module implementation specific. It is enumerated to the host VMM using the TDH.SYS.RD/RDALL or TDH.SYS.INFO function, as described below.

8.3. PAMT Details

The Physical Address Metadata Table (PAMT) is designed to track the metadata of every physical page in TDMR. A page metadata includes page type, page size, assignment to a TD, and other attributes.

The PAMT is used by the Intel TDX module to help enforce the following properties:

Page Attributes A physical page in TDMR has a well-defined set of attributes, such as page type and page size.

Single TD Assignment A physical page in TDMR can be assigned to at most one TD.

Secure EPT Consistency The page size of any private TD page, mapped in Secure EPT, matches its page size attribute in PAMT.

8.3.1. PAMT Entry

Note: The description below is provided at a high level. Implementation details may differ.

A PAMT entry is designed to hold metadata for a single physical page. The page size may be 4KB, 2MB or 1GB depending on the PAMT level (see 8.3.2 below).

Section 2: Introduction and Overview

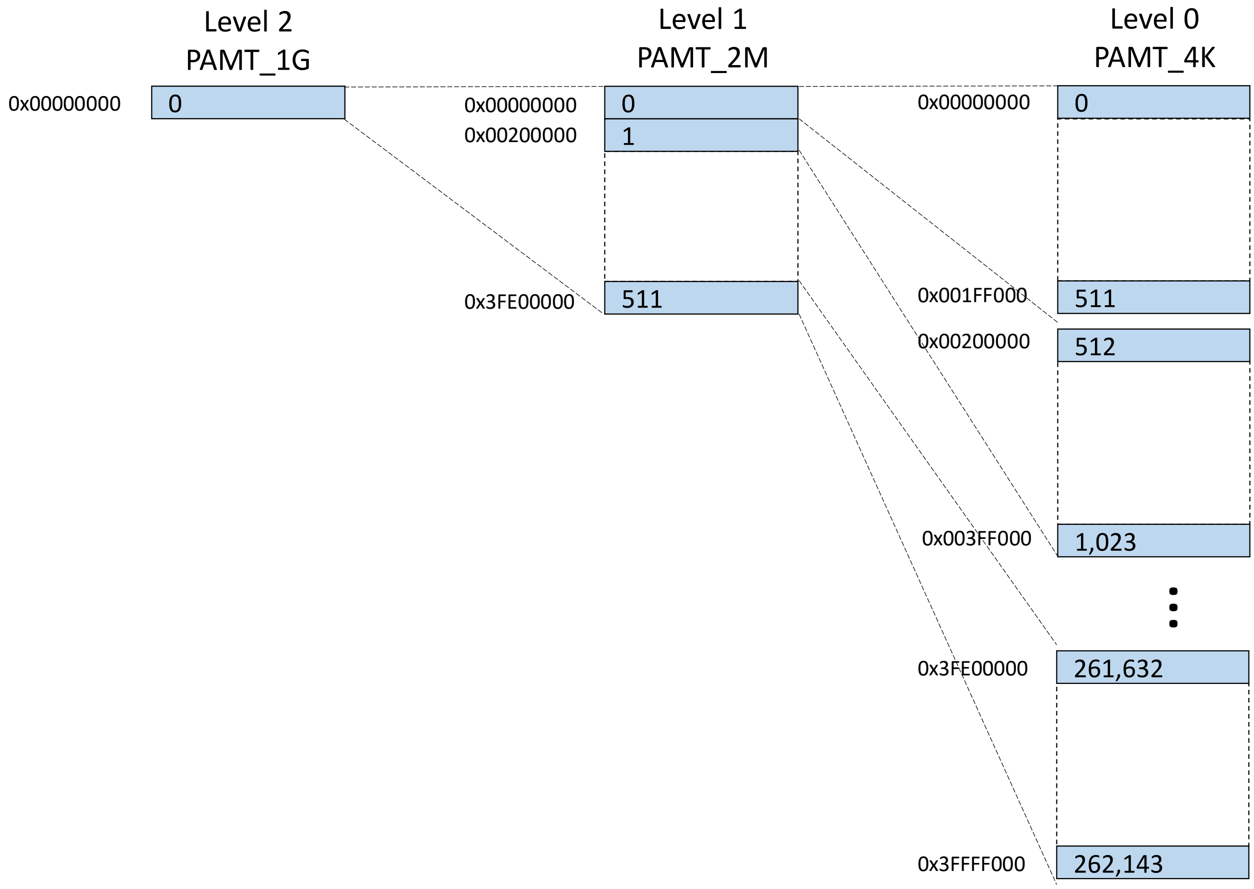
Table 8.1: High-Level View of a PAMT Entry

Field	Description
PT	PT indicates the type of page intended to be associated with this PAMT entry. See Table 8.3 below for details.
OWNER	OWNER is designed to contain bits 51:12 of the physical address of the TD's TDR page. This field can be applicable in all cases when a page is assigned to the Intel TDX module at this PAMT level or at a higher level. See Table 8.3 below for details.
BEPOCH	By design, the value of TDCS.TD_EPOCH as sampled by TDH.MEM.RANGE.BLOCK This field is intended to be applicable only if PT is PT_REG or PT_EPT. See 9.7 for a detailed discussion.

8.3.2. PAMT Blocks and PAMT Arrays

For each 1GB of TDMR physical memory, there is a corresponding **PAMT Block**. A PAMT Block is **logically** arranged in a 5 three-level tree structure of **PAMT Entries**, as shown in Figure 8.1 below. Levels 0 through 2 (PAMT_4K, PAMT_2M and PAMT_1G) correspond to 4KB, 2MB and 1GB physical TDMR pages, respectively.

Physically, for each TDMR the design includes three arrays of PAMT entries, one for each PAMT level. This aims to simplify VMM memory allocation. A logical PAMT Block has one entry from the PAMT_1G array, 512 entries from the PAMT_2M array, and 512^2 entries from the PAMT_4K array.



10 Figure 8.1: Typical Example of a PAMT Block Hierarchy for a 1GB TDMR Block

8.3.3. PAMT Page Types

Table 8.2 below describes the PAMT page types:

Table 8.2: PAMT Page Types

Page Type	PAMT Level	Associated TDX Page	Description
PT_NDA	Any	Depending on PT at higher PAMT level, if any	The physical page is **Not Directly Assigned** to the Intel TDX module at this size (4K, 2M or 1G) and PAMT level. This page may be part of a larger page that is assigned to the Intel TDX module at a higher level, or this page may contain smaller pages that are assigned to the Intel TDX module at lower levels. See Table 8.3 below for details.
PT_RSVD	PAMT_4K	None	The physical page is reserved for non-TDX usage. The Intel TDX module will not allow converting this page to any other page type. The page can be used by the host VMM for any purpose. PT_RSVD is used for implementing reserved areas within TDMRs. See 4.1.4.2.1 for details.
PT_REG	Any	TD private page	The physical page at this PAMT level (4K, 2M or 1G) holds TD private memory and is mapped in the guest TD GPA space by the Secure EPT.
PT_TDR	PAMT_4K	TDR	TDR control structure page
PT_TDCX	PAMT_4K	TDCX	One 4KB physical page of a multi-page control structure
PT_TDVPR	PAMT_4K	TDVPR	Root page of the multi-page TDVPS control structure
PT_EPT	PAMT_4K	Secure EPT	Secure EPT page

5 ### 8.3.4. PAMT Hierarchy

Table 8.3 below shows the page type (PT) of PAMT entries at the three levels of hierarchy, depending on whether the page is assigned to the Intel TDX module manages the page, whether the page is mapped in secure EPT, and the mapping size.

Table 8.3: PAMT Hierarchy and Page Types

Intel TDX Module Management			PAMT Entry Page Type		
Assigned to TDX?	**Physical Page Size**	**GPA Mapping Size (Secure EPT Level)**	**PAMT_1G (Level 2)**	**PAMT_2M (Level 1)**	**PAMT_4K (Level 0)**
No	**4KB**	**N/A**	PT_NDA	PT_NDA	**PT_RSVD**
No	**4KB**	**N/A**	PT_NDA	PT_NDA	PT_NDA
Yes	**4KB**	**None**	PT_NDA	PT_NDA	PT_TDR, PT_TDCX, PT_TDVPR, PT_EPT
Yes	**4KB**	**4KB (Level 0)**	PT_NDA	PT_NDA	**PT_REG**
Yes	**2MB**	**2MB (Level 1)**	PT_NDA	**PT_REG**	PT_NDA
Yes	**1GB**	**1GB (Level 2)**	**PT_REG**	PT_NDA	PT_NDA

10

Note the following:

- A 4KB page is considered **free** (i.e., not assigned to TDX) if its PAMT.PT at all three PAMT levels is PT_NDA. Any function that attempts to assigns an HPA to TDX (e.g., TDH.MEM.PAGE.ADD) is designed to check this.
- In all other cases, PAMT.PT is different than PT_NDA in only one of the three PAMT levels.

- When a page is mapped by Secure EPT at 4KB, 2MB or 1GB GPA mapping size, it is managed by the Intel TDX module as a physical page of the same size. Secure EPT is described in Chapter 9.
- PT_RSVD pages cannot be used by the Intel TDX module. They are used for implementing reserved areas within TDMRs. See 4.1.4.2.1 for details.

8.4. Adding Physical Pages

8.4.1. Preventing Cache Line Aliasing

Before adding a physical page, the host VMM is responsible for making sure no MODIFIED cache lines exist for that page. The host VMM can flush cache lines to memory – e.g., using CLFLUSH (only for pages containing data encrypted with a shared HKID – the VMM cannot directly use an HPA with a private HKID), or TDH.PHYMEM.PAGE.WBINVD (for pages containing data encrypted with any HKID, as long as the page is within a TDMR). Flushing cache lines to memory is required to avoid corruption due to cache line aliasing.

8.4.2. Adding Pages not Mapped to the Guest TD

By design, TD control structure pages TDR, TDCX and TDVPR are not mapped to the guest TD's GPA space, and they are only managed using their HPA. The functions TDH.MNG.CREATE, TDH.MNG.ADDCX, TDH.VP.CREATE and TDH.VP.ADDCX are designed to add 4KB control structure pages PT_TDR, PT_TDCX and PT_TDVPR, respectively. The overall process is described in 7.3 and 7.4.

8.4.3. Adding Pages and Mapping to the Guest TD's GPA

The following page types are associated with a guest TD's GPA:

- Guest TD private pages
- Secure EPT pages are mapped to the guest TD's GPA space.

Those pages are added given their HPA and the required GPA. The functions TDH.MEM.PAGE.ADD, TDH.MEM.PAGE.AUG, TDH.MEM.PAGE.RELOCATE and TDH.IMPORT.MEM add a PT_REG page, and the functions TDH.MEM.SEPT.ADD and TDH.MEM.PAGE.DEMOTE add a 4KB PT_EPT page. TD private memory management functions are described in Chapter 9. This section describes only their physical page management aspects.

8.5. Reclaiming Physical Pages

8.5.1. Reclaiming Pages not Mapped to the Guest TD

There are two cases where pages are not considered as mapped to the guest TD:

- Control structure pages are not mapped to the guest TD.
- In TD_TEARDOWN state, as described below, no mapping is in effect.

8.5.2. Reclaiming TD Pages in TD_TEARDOWN State

As part of the TD teardown process, the VMM needs to put the TD into a TD_TEARDOWN state, as described in 7.4. This is a non-recoverable state where TD keys have been reclaimed, all address translations and caches have been flushed, and the TD private memory and control structures (except TDR) are no longer accessible.

By design, in the TD_TEARDOWN state, all TD pages are effectively unmapped. Secure EPT is not accessible, and no GPA-to-HPA mapping can be used. The host VMM must treat all the TD private pages and control structure pages as physical memory and reclaim them using the TDH.PHYMEM.PAGE.RECLAIM function in any order, as long as the TDR page is the last one to be reclaimed.

For TDR page, the intention is for the host VMM to call TDH.PHYMEM.PAGE.WBINVD after calling TDH.PHYMEM.PAGE.RECLAIM. This is required to avoid corruption due to cache line aliasing because the TDR page has still been accessed and modified, even when the TD was in TD_TEARDOWN state.

8.5.3. Reclaiming Physical Pages as Part of TD Private Memory Management

Functions such as TDH.MEM.PAGE.REMOVE and TDH.MEM.PAGE.PROMOTE are designed to remove TD private pages and Secure EPT pages, respectively. By design, they first make sure the pages are no longer accessible using a GPA, then they mark the physical page as free. This is described in Chapter 9; this section only highlights the physical page reclamation.

9. TD Private Memory Management

This chapter described how the Intel TDX module helps manage TD private memory and guest-physical address (GPA) translation.

9.1. Overview

Intel TDX ISA introduced the concept of private GPA vs. shared GPA, depending on the GPA.SHARED bit. In SEAM non-root mode, the controlling VMCS has two EPT pointer fields:

- The legacy EPT pointer is used for translating the guest TD's memory accesses using a private GPA (i.e., GPA.SHARED == 0).
- A new Shared EPT pointer is used for translating the guest TD's memory accesses using shared GPAs (i.e., GPA.SHARED == 1).

A new GPAW execution control determines the position of the SHARED bit in the GPA, and a new HKID execution control defines the HKID used for accessing TD private memory.

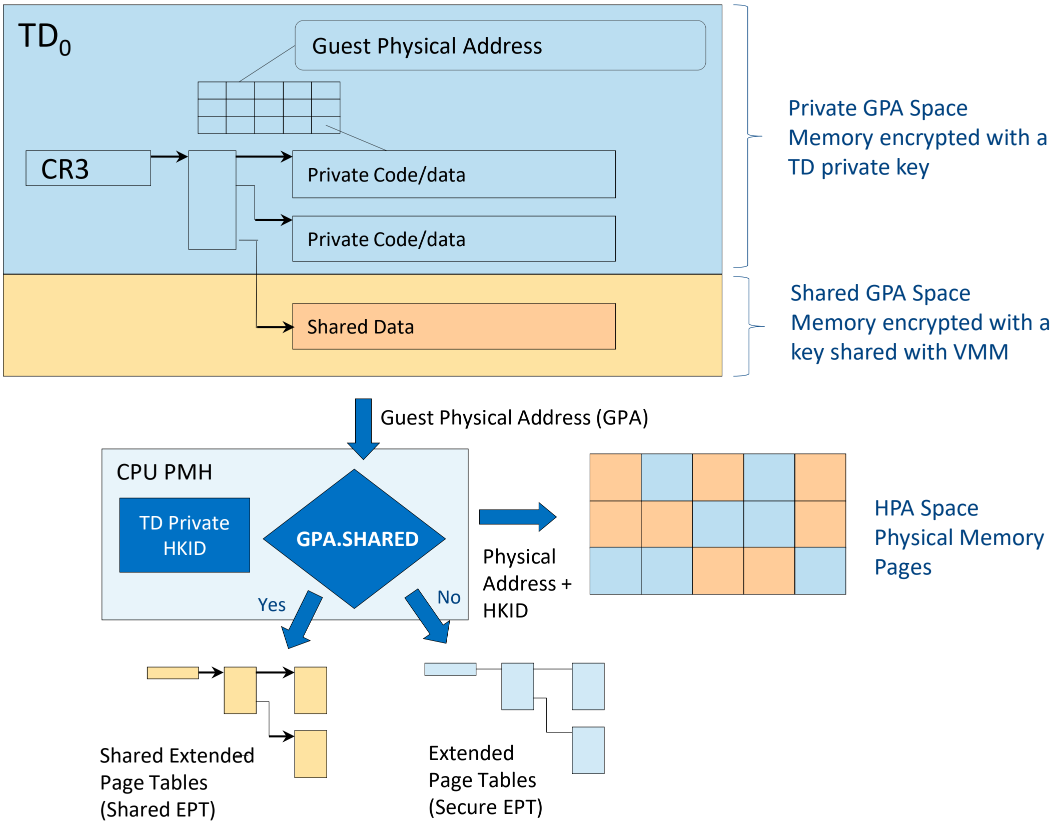


Figure 9.1: Secure EPT Concept

The Intel TDX module maintains a single Secure EPT structure per TD. Secure EPT pages are designed to be opaque; they reside in ordinary memory, and they are encrypted and integrity-protected with the TD's ephemeral private key. The Intel TDX module does not map Secure EPT pages to the guest TD GPA space. Thus, Secure EPT is effectively not accessible by any software besides the Intel TDX module, nor by any devices. Any such access using shared HKID to Secure EPT can lead to data corruption that triggers integrity check failure leading to a machine check fault.

Secure EPT is intended to be managed indirectly by the host VMM using Intel TDX functions. The Intel TDX module helps ensure that the Secure EPT is managed correctly.

The CPU translates shared GPAs using the Shared EPT which resides in host VMM memory. The translation uses a shared HKID, and it is directly managed by the host VMM, just as with legacy VMX.

9.2. Secure EPT Entry

9.2.1. UPDATED: Overview

From the CPU perspective, Secure EPT has the same structure as a legacy VMX EPT.

For the purpose of private memory management, the Intel TDX module hold a state value in each Secure EPT entry. This state value is encoded by multiple bits.

Table 9.1: UPDATED: Secure EPT Entry State High Level Description

State Name	Public State Number	Description
FREE	0	Secure EPT entry does not map a GPA range.
REMOVED	5	Secure EPT entry is of a removed page
MAPPED	4	Secure EPT entry maps a private GPA range which is accessible by the guest TD.
BLOCKED	1	Secure EPT entry maps a private GPA range, but new address translations to that range are blocked.
BLOCKEDW	8	Secure EPT entry maps a private GPA range, but new address translations for write operations to that range are blocked.
EXPORTED_BLOCKEDW	9	Secure EPT entry maps a private page that has been blocked for writing and exported.
EXPORTED_DIRTY	11	Secure EPT entry maps a private page that was exported, but is not blocked for writing and its content and/or attributes may have since been modified.
EXPORTED_DIRTY_BLOCKEDW	12	Secure EPT entry maps a private page that was previously exported, its content and/or attributes may have since been modified and then it was blocked for writing.
PENDING	2	Secure EPT entry maps a 4KB or a 2MB page that has been dynamically added to the guest TD using TDH.MEM.PAGE.AUG and is pending acceptance by the guest TD using TDG.MEM.PAGE.ACCEPT. This page is not yet accessible by the guest TD.
PENDING_BLOCKED	3	Secure EPT entry is both pending and blocked.
PENDING_BLOCKEDW	16	Secure EPT entry is both pending and blocked for writing.
PENDING_EXPORTED_BLOCKEDW	17	Secure EPT entry is both pending and exported.
PENDING_EXPORTED_DIRTY	19	Secure EPT entry is both pending and exported, and is not blocked for writing.
PENDING_EXPORTED_DIRTY_BLOCKEDW	20	Secure EPT entry is both pending and exported, and is blocked for writing.

Secure EPT entry is opaque; the host VMM may not access it directly. The host VMM may read a Secure EPT entry information using the TDH.MEM.SEPT.RD interface function. In addition, multiple other interface functions return the same information in case of an error that is related to a Secure EPT entry. For details, see the [TDX Module ABI].

9.2.2. UPDATED: SEPT Entry State Diagrams

The figures below show partial state diagrams for the basic memory management operation for a leaf and a non-leaf SEPT entry.

Additional SEPT entry state diagrams for TD migration are provided in the [TD Migration Spec].

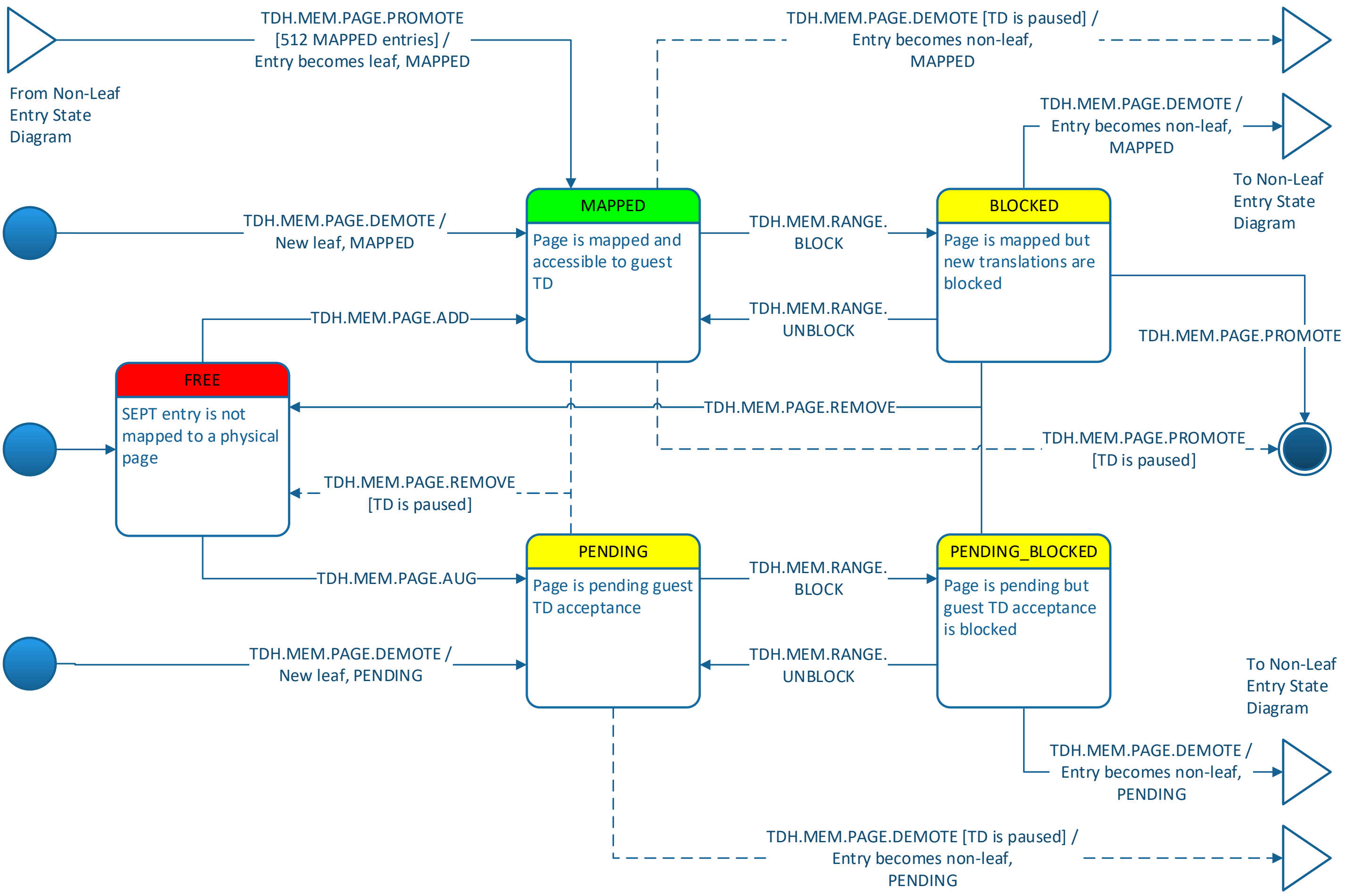
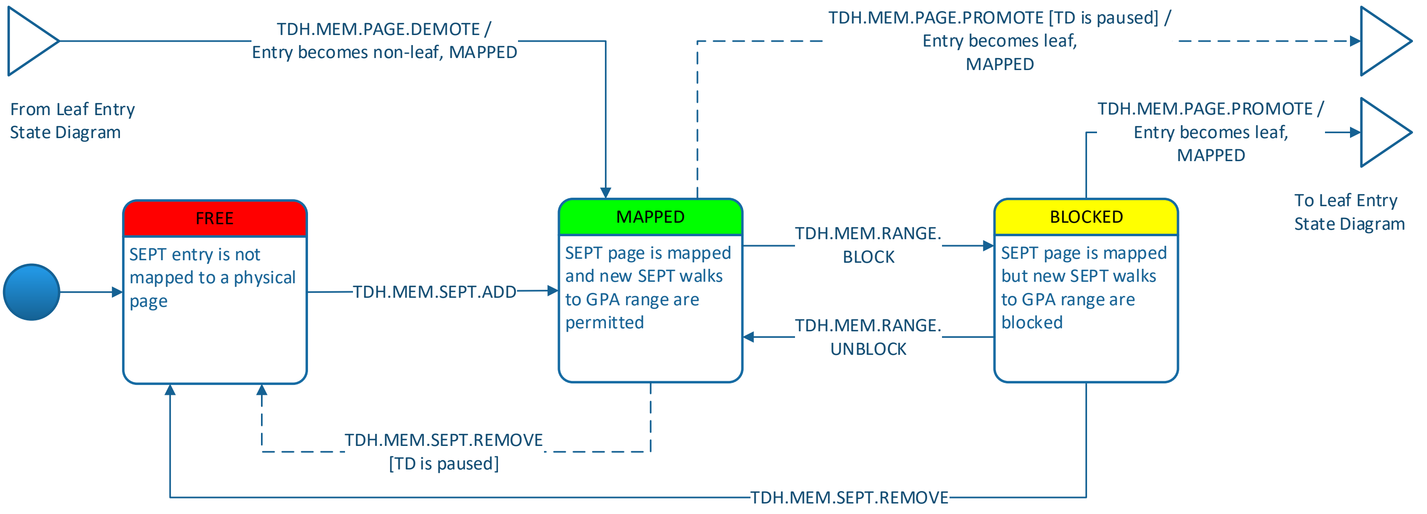

From Non-Leaf Entry State Diagram
TDH.MEM.PAGE.PROMOTE [512 MAPPED entries] / Entry becomes leaf, MAPPED
TDH.MEM.PAGE.DEMOTE [TD is paused] / Entry becomes non-leaf, MAPPED
TDH.MEM.PAGE.DEMOTE / Entry becomes non-leaf, MAPPED
To Non-Leaf Entry State Diagram
TDH.MEM.PAGE.DEMOTE / New leaf, MAPPED
MAPPED
Page is mapped and accessible to guest TD
TDH.MEM.RANGE. BLOCK
BLOCKED
Page is mapped but new translations are blocked
TDH.MEM.RANGE. UNBLOCK
TDH.MEM.PAGE.ADD
TDH.MEM.PAGE.PROMOTE
FREE
SEPT entry is not mapped to a physical page
TDH.MEM.PAGE.REMOVE
TDH.MEM.PAGE.PROMOTE [TD is paused]
TDH.MEM.PAGE.REMOVE [TD is paused]
PENDING
Page is pending guest TD acceptance
TDH.MEM.PAGE.AUG
TDH.MEM.RANGE. BLOCK
PENDING_BLOCKED
Page is pending but guest TD acceptance is blocked
TDH.MEM.PAGE.DEMOTE / New leaf, PENDING
TDH.MEM.RANGE. UNBLOCK
To Non-Leaf Entry State Diagram
TDH.MEM.PAGE.DEMOTE / Entry becomes non-leaf, PENDING
TDH.MEM.PAGE.DEMOTE [TD is paused] / Entry becomes non-leaf, PENDING

Figure 9.2: Secure EPT Leaf Entry Basic Operation Partial State Diagram


From Leaf Entry State Diagram
TDH.MEM.PAGE.DEMOTE / Entry becomes non-leaf, MAPPED
TDH.MEM.PAGE.PROMOTE [TD is paused] / Entry becomes leaf, MAPPED
TDH.MEM.PAGE.PROMOTE / Entry becomes leaf, MAPPED
To Leaf Entry State Diagram
FREE
SEPT entry is not mapped to a physical page
TDH.MEM.SEPT.ADD
MAPPED
SEPT page is mapped and new SEPT walks to GPA range are permitted
TDH.MEM.RANGE. BLOCK
BLOCKED
SEPT page is mapped but new SEPT walks to GPA range are blocked
TDH.MEM.RANGE. UNBLOCK
TDH.MEM.SEPT.REMOVE [TD is paused]
TDH.MEM.SEPT.REMOVE

5 **Figure 9.3: Secure EPT Non-Leaf Entry Basic Operation Partial State Diagram**

9.3. Secure EPT Walk

Host-side (SEAMCALL) Intel TDX functions that manage TD private memory usually accept GPA and Level parameters. They perform a Secure EPT walk which locates the target Secure EPT entry.

If the Secure EPT walk is completed successfully, the Intel TDX function may operate on the located Secure EPT entry. 10 Otherwise, the function typically returns the last visited EPT entry and its level to the host VMM.

Guest-side (TDCALL) Intel TDX functions typically perform an EPT walk similar to the EPT walk done by the CPU. Only the GPA is provided as an input, and the function may walk the Shared EPT or the Secure EPT, depending on the specific function and the GPA's SHARED bit.

9.4. Secure EPT Induced TD Exits

Guest TD memory access to a non-present private GPA causes an asynchronous TD exit with an EPT Violation exit reason. As discussed in 9.2 above, a non-present GPA is any private GPA for which there is either no Secure EPT entry, or the Secure EPT entry is not in the MAPPED state. There is no TD exit with an EPT Misconfiguration on the Secure EPT.

On EPT violation TD exit, the content of the Secure EPT entry is provided to the host VMM to avoid the need for explicitly reading it using TDH.MEM.SEPT.RD.

Secure EPT-induced TD exits may also be triggered during a guest-side local flow, performing some function on behalf of the guest TD, and executed by the Intel TDX module.

9.5. Secure EPT Induced Exceptions

Guest TD memory access to a private GPA for which the Secure EPT entry state is PENDING causes a #VE.

Guest TD memory access with any GPA bit higher than the SHARED bit set to 1 causes a #PF exception. See 11.11.1.

9.6. UPDATED: Secure EPT Concurrency

Secure EPT concurrency rules are designed to allow concurrent operations on multiple Secure EPT entries.

Host-Side (SEAMCALL) Interface Functions

- TDX module interface functions that use GPA as an input acquire a **shared lock** on the whole Secure EPT tree of the target TD to help prevent changes to the tree while they execute.
- Most interface functions that use GPA as an input acquire an **exclusive host-side lock** on the Secure EPT entry or entries which they use. An exception to this is TDH.MEM.SEPT.RD, which just reads a Secure EPT entry and does not use it to actually access memory.
- In specific cases where a Secure EPT entry update may collide with a concurrent update done by the guest TD, host-side interface functions update the Secure EPT entry as a transaction, using atomic compare and exchange operation.

Guest-Side (TDCALL) Interface Functions

Guest-side TDX module interface functions that need to translate a GPA to an HPA emulate the CPU's top-down EPT walk operation.

- Guest-side interface functions have no concurrency restrictions on the whole Secure EPT tree.
- Guest-side interface functions that need to update a Secure EPT entry (currently, only TDG.MEM.PAGE.ACCEPT) acquire an **exclusive guest-side lock** on that entry. This lock is only checked by other similar guest-side functions, but not by host-side functions. Thus, Secure EPT entry update is done as a transaction, using atomic compare and exchange operation.

9.7. Introduction to TLB Tracking

The goal of TLB tracking is to be able to prove (when needed) that no logical processor holds any cached Secure EPT address translations to a given TD private **GPA range**. TLB tracking is required when removing a mapped TD private page (TDH.MEM.PAGE.REMOVE) or when changing the page mapping size (TDH.MEM.PAGE.PROMOTE), etc.

GPA Range TLB Tracking Sequence

This sequence is intended to be used by the host VMM to help guarantee no EPT TLB entries exist to a set of GPA ranges.

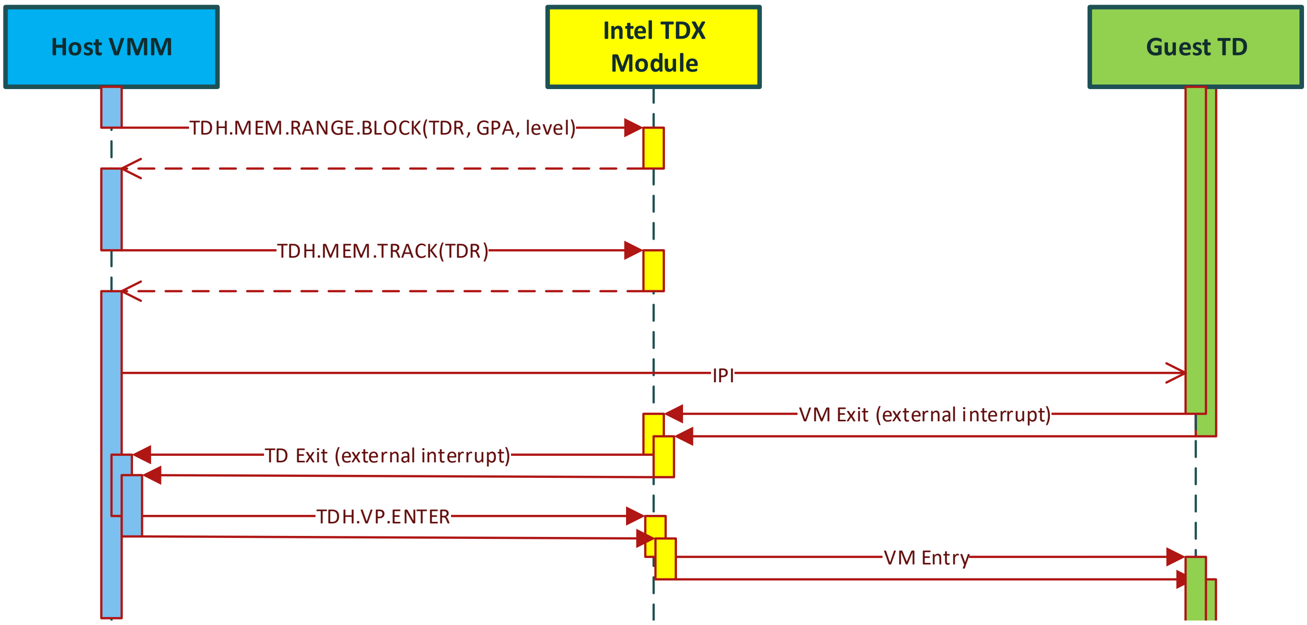

Host VMM
Intel TDX Module
Guest TD
TDH.MEM.RANGE.BLOCK(TDR, GPA, level)
TDH.MEM.TRACK(TDR)
IPI
VM Exit (external interrupt)
TD Exit (external interrupt)
TDH.VP.ENTER
VM Entry

Figure 9.4: Typical TLB Tracking Sequence

The sequence typically includes five steps:

1. Execute TDH.MEM.RANGE.BLOCK on each GPA range, blocking subsequent creation of TLB translation to that range. Note that cached translations may still exist at this stage.
2. Execute TDH.MEM.TRACK, advancing the TD's epoch counter.
3. Send an Inter-Processor Interrupt (IPI) to each Remote Logical Processor (RLP) on which any of the TD's VCPUs is currently scheduled.
4. Upon receiving the IPI, each RLP will TD exit to the host VMM.

At this point the target GPA ranges are considered tracked. Even though some LPs may still hold TLB entries to the target GPA ranges, the following TD entry is designed to flush them.

5. Normally, the host VMM on each RLP will treat the TD exit as spurious and will immediately re-enter the TD.

9.8. Secure EPT Build and Update: TDH.MEM.SEPT.ADD

The host VMM can use the TDH.MEM.SEPT.ADD function to add a Secure EPT page to a guest TD. TDH.MEM.SEPT.ADD inputs are:

- Target TD, identified by its TDR HPA
- Destination physical page for the new Secure EPT table
- Mapping information: GPA and EPT level

At a high level, TDH.MEM.SEPT.ADD works as follows:

1. Check the TD keys are configured.
2. Check the destination physical page is marked as free in the PAMT.
3. Perform a Secure EPT walk to locate the Secure EPT non-leaf entry which will become the parent entry that maps the new Secure EPT page. To help prevent re-maps, TDH.MEM.SEPT.ADD checks the mapping does not already exist, else it aborts the operation.
4. Initialize the target page to zero using the target TD's private HKID and direct writes (MOVDIR64B).
5. Update the parent Secure EPT entry to map the page as MAPPED.
6. Update the page's PAMT entry with the PT_EPT page type and the TDR PA as the OWNER.

The Secure EPT's root page (EPML4 or EPML5, depending on whether the host VMM uses 4-level or 5-level EPT) does not need to be explicitly added. It is created during TD initialization (TDH.MNG.INIT) and is stored as part of TDCS. On each

VCPU initialization, TDH.VP.INIT copies the address of the Secure EPT root page to the VCPU's TD VMCS's EPTP field clearing the HKID bits to 0[5].

The following example illustrates the build process of a 4-level Secure EPT hierarchy:

1. The host VMM calls TDH.MNG.CREATE(TDR_PA = TDR$_0$) to create the TD.
2. The host VMM calls TDH.MNG.ADDCX(TDR_PA = TDR$_0$, DST_PA = TDCX_PAGE_PA) multiple times to allocate pages for TDCS. One of those pages will be used to host the Secure EPT root page D$_0$.
3. Host VMM calls TDH.MNG.INIT(TDR_PA = TDR$_0$) to initialize the TD and set an EPML4 page in one of the previously added TDCX pages as the Secure EPT root page. This updates TDCS.EPTP.
4. TDH.VP.INIT of each VPCU copies TDCS.EPTP to the TD VMCS's EPTP field.
5. Host VMM calls TDH.MEM.SEPT.ADD(TDR_PA = TDR$_0$, DST_PA = D$_1$, GPA = G$_0$, LVL= 3) to add an EPDPT page.
6. Host VMM calls TDH.MEM.SEPT.ADD(TDR_PA = TDR$_0$, DST_PA = D$_2$, GPA = G$_0$, LVL= 2) to add an EPD page.
7. Host VMM calls TDH.MEM.SEPT.ADD(TDR_PA = TDR$_0$, DST_PA = D$_3$, GPA = G$_0$, LVL= 1) to add an EPT page.

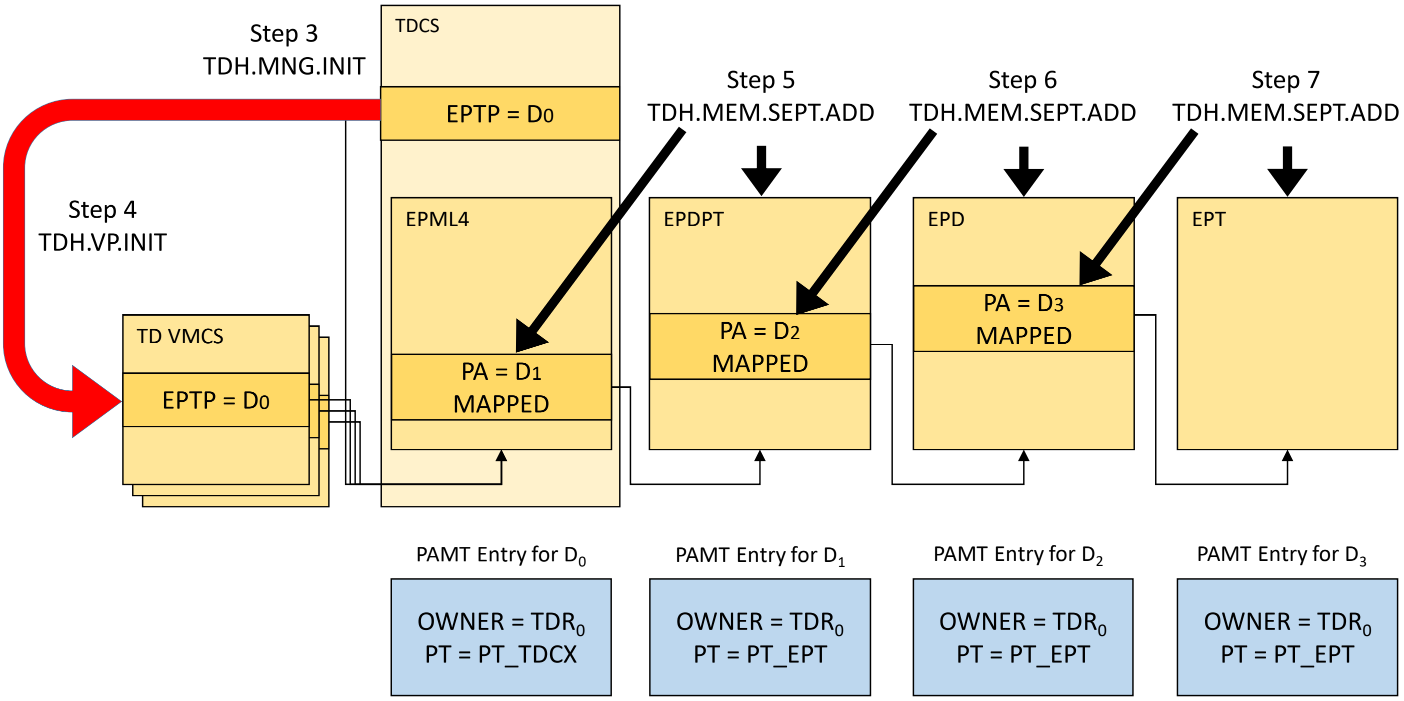


Figure 9.5: Typical Secure EPT Hierarchy Build Process

To help avoid stability issues caused by cache line aliasing, the VMM should assure that no cache lines associated with the added physical SEPT page are in a Modified state, before calling TDH.MEM.PAGE.AUG. This is typically done by calling TDH.PHYMEM.PAGE.WBINVD.

9.9. Adding TD Private Pages during TD Build Time: TDH.MEM.PAGE.ADD

Adding TD private pages with arbitrary content is allowed only during TD build time (before TDH.MR.FINALIZE). The host VMM adds and maps 4KB private pages to a guest TD using TDH.MEM.PAGE.ADD with the following inputs:

- Target TD, identified by its TDR physical address
- Source page physical address
- Destination page physical address
- Destination page GPA

At a high level, TDH.MEM.PAGE.ADD works as follows:

1. Check the TD has not been initialized.
2. Check the TD keys are configured.
3. Check the destination physical page is marked as free in the PAMT.

[5] The CPU adds the TD's private HKID on EPT walks. Having HKID as 0 allows the host VMM to use INVEPT, for managing the usage of shared EPT which shares the ASID with the TD's secure EPT (see ⍰).

4. Perform a pseudo Secure EPT walk to locate the parent Secure EPT leaf entry that is going to map the new TD private page. To help prevent re-maps, TDH.MEM.PAGE.ADD checks the mapping does not already exist, else it aborts the operation.
5. Copy the source page to the destination page using the target TD's private HKID and direct writes (MOVDIR64B).
6. Update the previously located parent Secure EPT leaf entry to map the page as MAPPED.
7. Update the TD measurement with the new page GPA (as described in 12.1.1).
8. Update the PAMT entry with the PT_REG page type and the TDR PA as the OWNER.

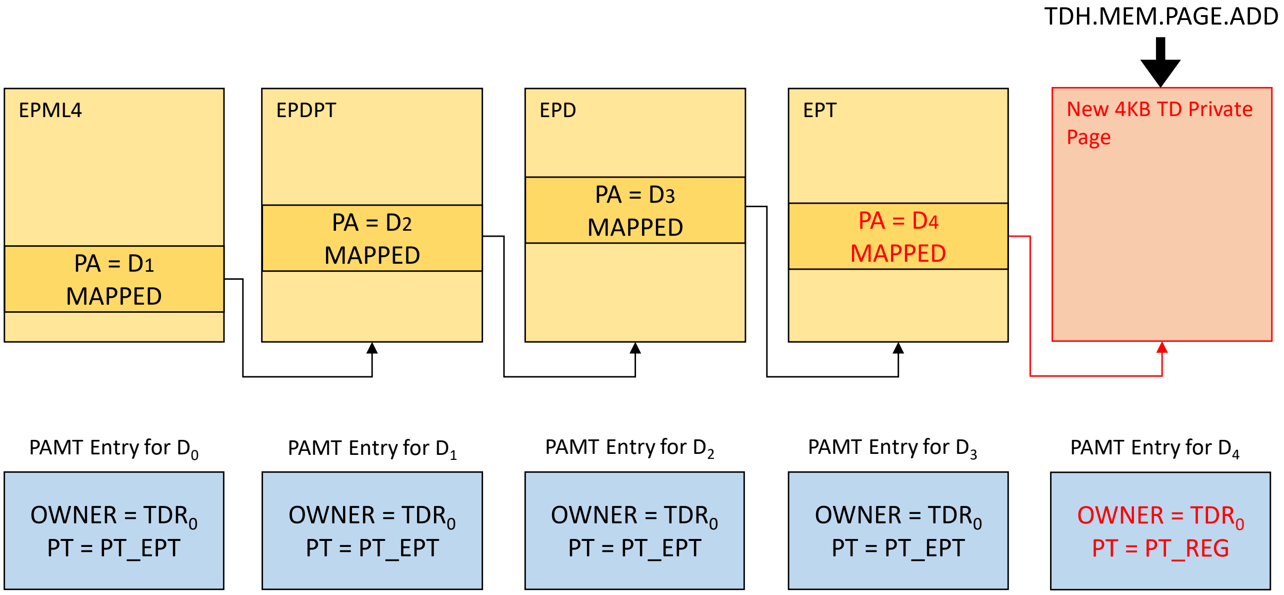


Figure 9.6: Typical Sequence for Adding a TD Private Page during TD Build Time

To help avoid stability issues caused by cache line aliasing, the VMM should assure that no cache lines associated with the added physical page are in a Modified state, before calling TDH.MEM.PAGE.AUG. This is typically done by calling TDH.PHYMEM.PAGE.WBINVD.

9.10. Dynamically Adding TD Private Pages

9.10.1. Overview

Dynamically adding TD private pages after the guest TD has been initialized is typically done as a three-step process:

- The host VMM can update Secure EPT using TDH.MEM.SEPT.ADD and TDH.MEM.SEPT.REMOVE.
- The host VMM adds and maps a 4KB or a 2MB TD private page using TDH.MEM.PAGE.AUG. This page is not measured. The Secure EPT entry state for that added page is PENDING.
- The guest TD must accept the page before it can access it, using TDG.MEM.PAGE.ACCEPT. The page content is zeroed out.

This process is designed to help prevent attacks where the host VMM could remove arbitrary pages from the guest TD's GPA space (using TDH.MEM.PAGE.REMOVE) and replace them with zeroed-out pages.

A guest TD attempt to access a page that has been dynamically added by TDH.MEM.PAGE.AUG but has not yet been accepted by TDH.MEM.PAGE.ACCEPT results in a #VE exception.

Refer also to the software flow described in 3.3.1.1.

9.10.2. Page Addition by the Host VMM: TDH.MEM.PAGE.AUG

The host VMM can add and map 4KB and 2MB private pages to a guest TD in a non-present and pending state using TDH.MEM.PAGE.AUG, with the following inputs:

- Target TD, identified by its TDR physical address
- Destination page physical address
- Destination page GPA

At a high level, TDH.MEM.PAGE.AUG works as follows:

1. Check the TD keys are configured.
2. Check that the TD has either been initialized (by TDH.MNG.INIT) and no migration session is in progress, or that migration is in progress, but the TD is runnable (live export or import).
3. Check the destination physical page is marked as free in the PAMT.
4. Perform a pseudo Secure EPT walk to locate the parent Secure EPT leaf entry that is going to map the new TD private page. To help prevent re-maps, TDH.MEM.PAGE.AUG checks the mapping does not already exist, else it aborts the operation.
5. Update the previously located parent Secure EPT leaf entry to map the page as PENDING.
6. Update the PAMT entry with the PT_REG page type and the TDR PA as the OWNER.

Note that TDH.MEM.PAGE.AUG does not need to access the destination page itself; the page is initialized later on by TDG.MEM.PAGE.ACCEPT.

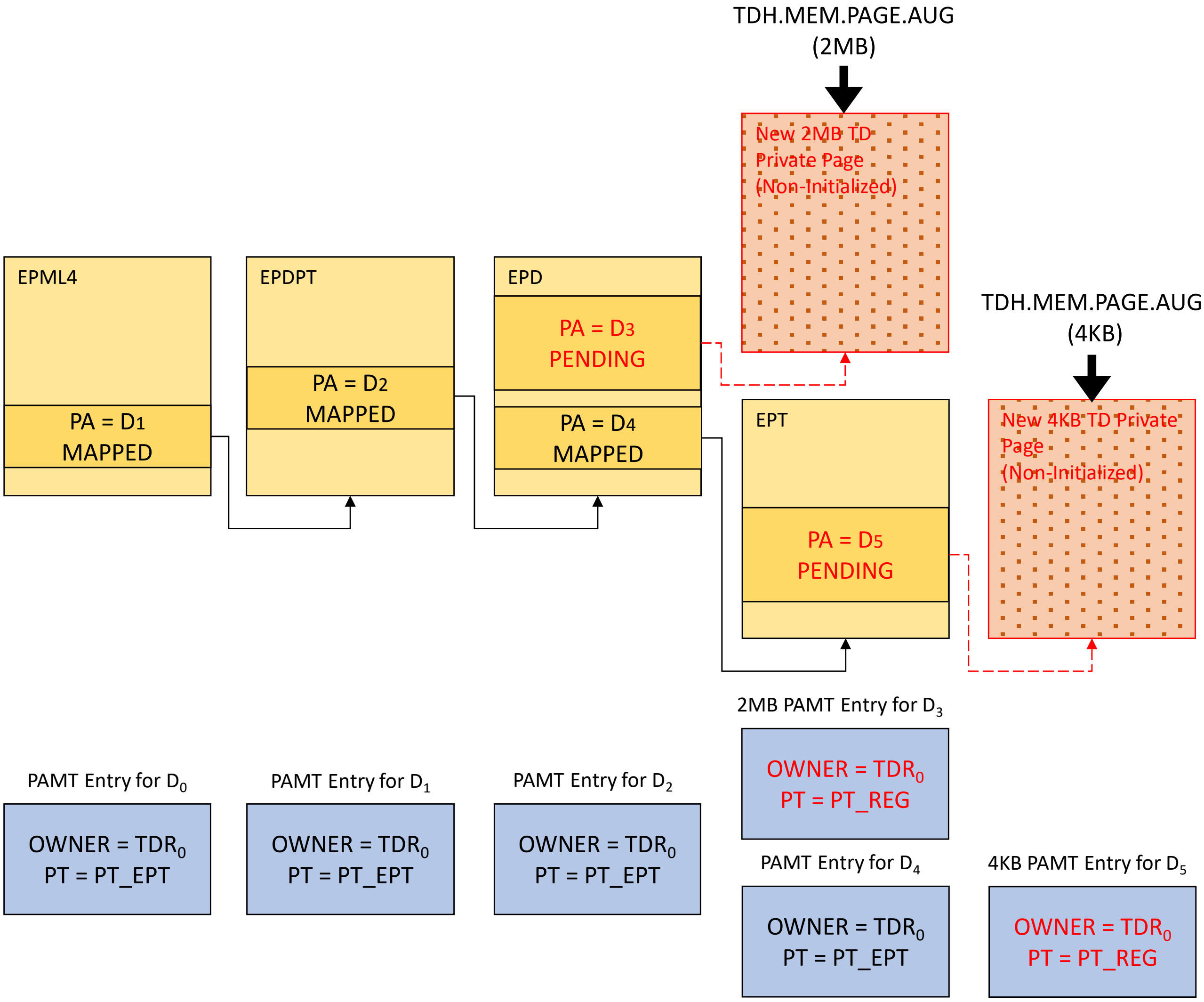


Figure 9.7: Host VMM Adding a 4KB or a 2MB TD Private Page

To help avoid stability issues caused by cache line aliasing, the VMM should assure that no cache lines associated with the added physical page are in a Modified state, before calling TDH.MEM.PAGE.AUG. This can be done be calling TDH.PHYMEM.PAGE.WBINVD.

9.10.3. Page Acceptance by the Guest TD: TDG.MEM.PAGE.ACCEPT

9.10.3.1. *Description*

The guest TD can accept a dynamically added 4KB or 2MB page using TDG.MEM.PAGE.ACCEPT with the page GPA and size inputs.

At a high level, TDG.MEM.PAGE.ACCEPT works as follows:

1. Perform a Secure EPT walk to locate the parent Secure EPT leaf entry that maps the TD private page, and handle the walk results as described in the table below.

Table 9.2: TDG.MEM.PAGE.ACCEPT SEPT Walk Cases

SEPT Walk Terminal Entry			TDG.MEM.PAGE.ACCEPT Operation	Typical Software Handling
Level	**Leaf or Non-Leaf**	**State**		
Higher than requested	Leaf	Guest-accessible, i.e., MAPPED or EXPORTED_DIRTY (e.g., 2MB PTE present for a 4KB request).	Return a status code indicating a success, with a warning that the page is already present and mapped at a level higher than requested.	Option 1: This is OK, the host VMM did not use the memory released by the TD. Option 2: This is a guest bug; the status code helps debugging it.
		Not guest-accessible and not FREE (e.g., 2MB PTE pending for a 4KB request).	TD exit with EPT violation indicating the error SEPT entry level and state, and the guest-requested accept level. See the [TDX Module ABI].	The host VMM demotes the page to match the requested accept size.
	Non-leaf	Not guest-accessible (e.g. blocked PDE for a 4KB request).	TD exit with EPT violation indicating the error SEPT entry level and state, and the guest-requested accept level. See the [TDX Module ABI].	This may be used as a guest TD request from the host VMM to add a page. The host VMM adds SEPT pages (TDH.MEM.SEPT.ADD) and the requested page (TDH.MEM.PAGE.AUG), and resumes the guest.
Same as requested	Non-leaf	Other than FREE (e.g., requested 2MB entry is mapped to a EPT page instead of being a leaf)	Return a status code indicating a size mismatch error.	The guest falls back to accept the range using 4K size.
	Leaf	Guest-accessible, i.e., MAPPED or EXPORTED_DIRTY	Return a status code indicating a success, with a warning that the page is already present.	Option 1: This is OK, the host VMM did not use the memory released by the TD. Option 2: This is a guest bug; the status code helps debugging it.
		Not PENDING nor PENDING_EXPORTED_DIRTY	TD exit with EPT violation indicating the error SEPT entry level and state, and the guest-requested accept level. See the [TDX Module ABI].	The host VMM resolves the blocking (e.g., completes the memory management operation that required blocking) and resumes the guest.
		FREE	TD exit with EPT violation indicating the error SEPT entry level and state, and the guest-requested accept level. See the [TDX Module ABI].	This may be used as a guest TD request from the host VMM to add a page. The hosts VMM adds the requested page (TDH.MEM.PAGE.AUG) and resumes the guest.
		PENDING	Complete the operation as described below.	Success

If passed:

Note: Since initializing a 2MB page may take a long time, TDG.MEM.PAGE.ACCEPT is interruptible and resumable.

2. If all the above checks pass, loop until done or interrupted:
   2.1. Initialize the next 4KB chunk of the page to zero using the target TD's private HKID and direct writes (MOVDIR64B).
   2.2. If the whole page has been initialized, update the parent Secure EPT entry to set its state to SEPT_PRESENT.
   2.3. Else, if there is a pending interrupt, resume the guest TD without updating RIP and any GPR. The CPU may handle the interrupt, causing a TD exit. When the TD is resumed, TDH.MEM.PAGE.ACCEPT will re-invoked.

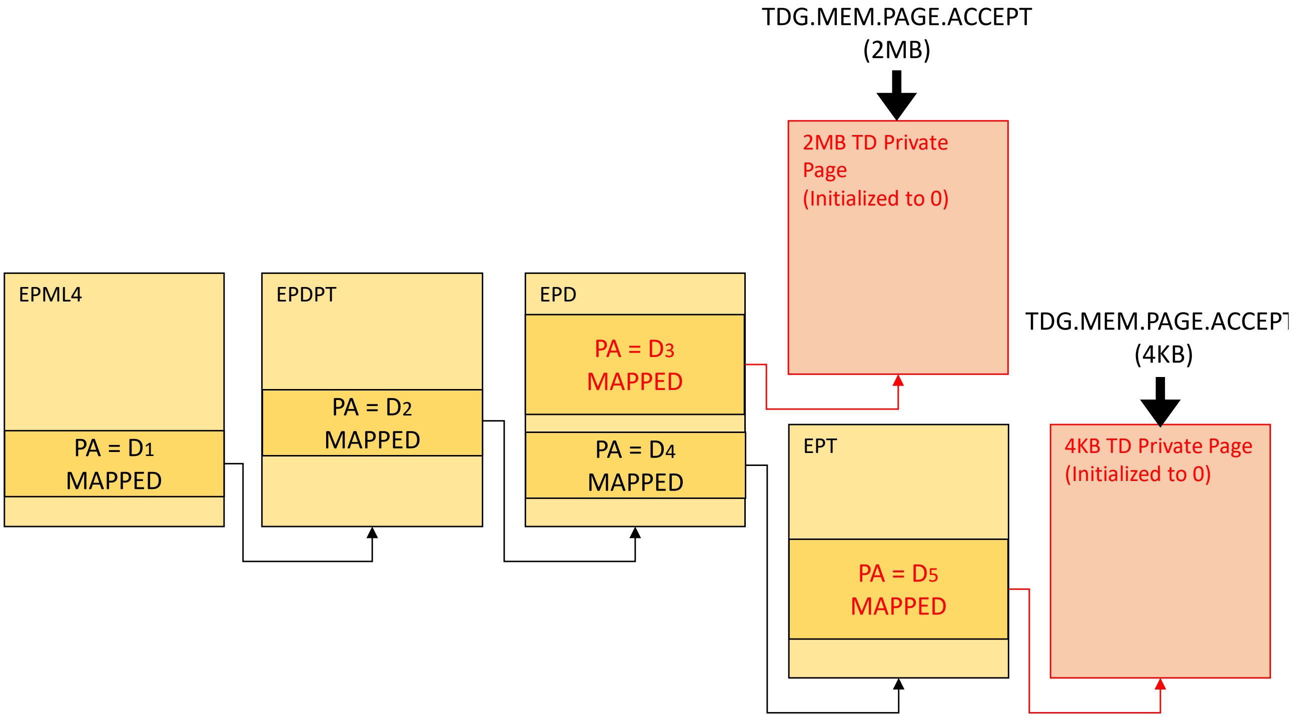


Figure 9.8: Guest TD Accepting a 4KB or 2MB Pending TD Private Page

9.10.3.2. *TDG.MEM.PAGE.ACCEPT Concurrency*

Guest-Side

TDG.MEM.PAGE.ACCEPT prevents the guest TD from concurrently accepting the same page by multiple threads. TDG.MEM.PAGE.ACCEPT may also encounter a concurrent host-side operation, such as TDH.MEM.RANGE.BLOCK, that attempts to update the same Secure EPT entry. In such cases, an error is returned to the guest TD, indicating that the Secure EPT entry is busy.

Host-Side

TDG.MEM.PAGE.ACCEPT does not prevent host-side operation, such as TDH.MEM.RANGE.BLOCK, from concurrently modifying the Secure EPT entry. TDG.MEM.PAGE.ACCEPT updates the entry using a locked compare and exchange operation. If the update failed, a TD Exit is caused, with an EPT Violation exit reason and an indication that the violation is due to TDG.MEM.PAGE.ACCEPT. For details, see the TDH.VP.ENTER definition in the [TDX Module ABI].

9.11. Page Merge: TDH.MEM.PAGE.PROMOTE

The host VMM can merge the mapping of 512 consecutive 4KB or 2MB pages to a single 2MB or 1GB page, respectively. To do that, the host VMM should first perform the TLB tracking protocol on the large (2MB or 1GB) GPA range.

The host VMM should first call TDH.MEM.RANGE.BLOCK which operates on the EPT page for the large range (EPT for 2MB, EPD for 1GB). TDH.MEM.RANGE.BLOCK marks the parent EPT entry for that EPT page as **BLOCKED** and records the TD epoch in the PAMT entry of the EPT page. Figure 9.9 below shows the situation after TDH.MEM.RANGE.BLOCK blocked a 2MB GPA range.

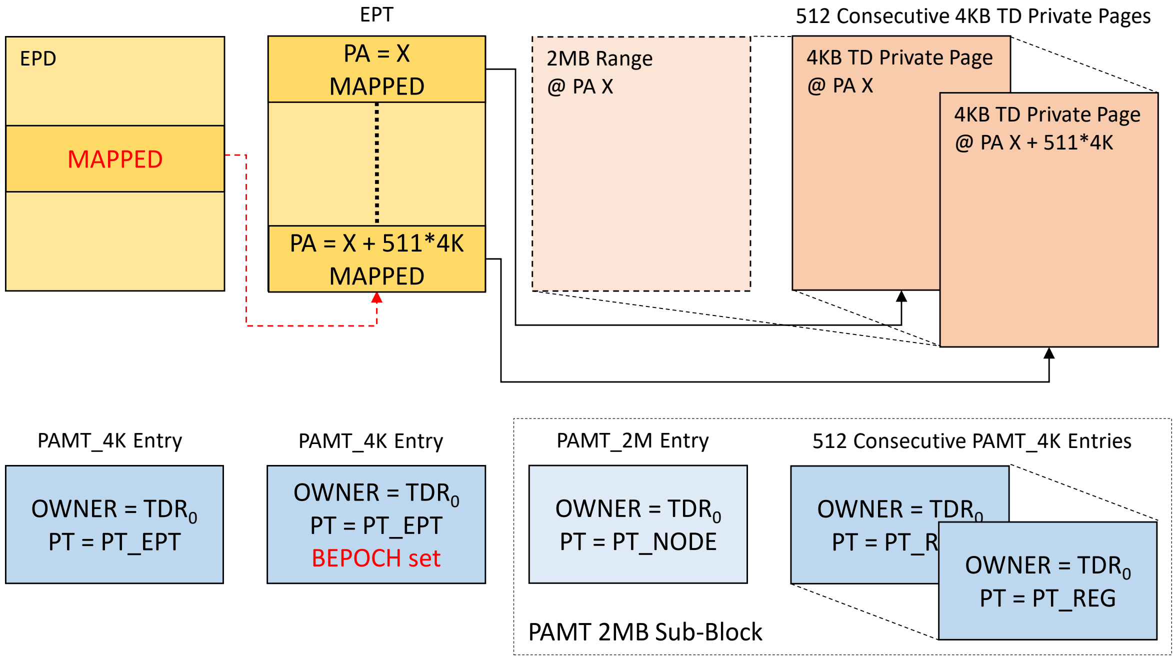


Figure 9.9: Typical State after Blocking a Range of 512 Consecutive 4KB TD Private Pages

Typically, the host VMM then calls TDH.MEM.TRACK and performs a round of IPIs. After that, there should be no active address translation to the large (2MB or 1GB) GPA range.

The actual merge is done by TDH.MEM.PAGE.PROMOTE which has the following inputs:

- The large range GPA
- The large page level (2MB or 1GB)

At a high level, TDH.MEM.PAGE.PROMOTE works as follows:

1. Check the TLB tracking condition for the large range GPA (i.e., the EPT or EPD page for that range).
2. Check that all 512 entries of that EPT or EPD page are in the MAPPED state and point to leaf pages whose physical address is contiguous within the same 2MB or 1GB range.

If all checks pass, TDH.MEM.PAGE.PROMOTE does the following:

1. Mark all the PAMT_4K or PAMT_2M entries of the small leaf pages (4KB or 2MB, respectively) as PT_NDA.
2. Mark the PAMT_2M or PAMT_1G entry of the merged large (2MB or 1GB, respectively) pages as PT_REG.
3. Set the parent EPT entry to point to the merged large page, and mark it as present.
4. Mark the original EPT or EPD page's PAMT entry as PT_NDA, effectively removing this for any use by the host VMM.

Once complete, the former EPT or EPD physical page should be free for use by the host VMM for any purpose. To help avoid stability issues caused by cache line aliasing, the host VMM should also assure that no cache lines associated with the page are in a Modified state. This is done be calling TDH.PHYMEM.PAGE.WBINVD.

Figure 9.10 below shows a typical 2MB merged page after TDH.MEM.PAGE.PROMOTE.

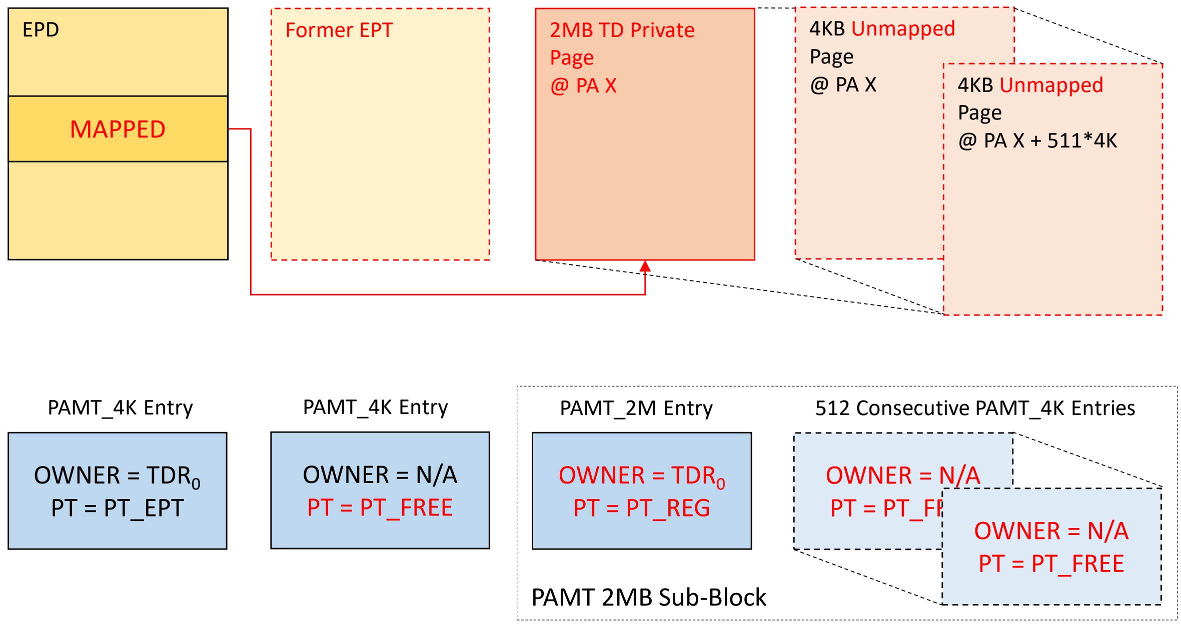


Figure 9.10: Typical State of a 2MB TD Private Page after TDH.MEM.PAGE.PROMOTE

Refer also to the software flow described in 3.3.1.3.

9.12. Page Split: TDH.MEM.PAGE.DEMOTE

The host VMM can split the mapping of a single 2MB or 1GB page to 512 consecutive 4KB or 2MB pages, respectively. To do that, the host VMM should first perform the TLB tracking protocol on the large (2MB or 1GB) page.

The host VMM should first call TDH.MEM.RANGE.BLOCK on the large page. TDH.MEM.RANGE.BLOCK marks the parent EPT entry for that page as **BLOCKED** and records the TD epoch in the PAMT entry of the page. Figure 9.11 below shows the typical situation after TDH.MEM.RANGE.BLOCK blocked a 1GB large page.

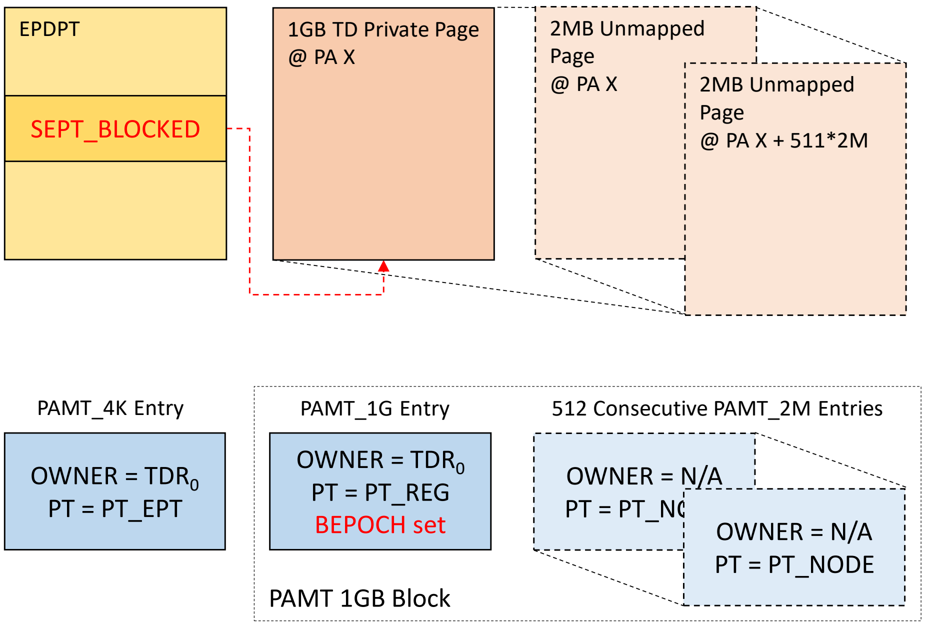


Figure 9.11: Typical State after Blocking a 1GB Page

Typically, the host VMM then calls TDH.MEM.TRACK and performs a round of IPIs. After that, there should be no active address translation to the large (2MB or 1GB) page.

The actual split is done by TDH.MEM.PAGE.DEMOTE which has the following inputs:

- The large page GPA
- The large page level (2MB or 1GB)

- The physical address of a free page that will be used for a new EPT or EPD page

At a high level, TDH.MEM.PAGE.DEMOTE works as follows:

1. Check the TLB tracking condition for the large page.
2. Check that the physical page for the new EPT or EPD is marked as free in the PAMT.

If all checks pass, TDH.MEM.PAGE.DEMOTE does the following:

3. Mark the PAMT_2M or PAMT_1G entry of the large (2MB or 1GB respectively) page as PT_NDA.
4. Mark all the PAMT_4K or PAMT_2M entries of the small (4KB or 2MB respectively) consecutive leaf pages as PT_REG.
5. Initialize the new EPT or EPD page with 512 EPT entries pointing to the 512 consecutive leaf pages.
6. Mark the new EPT or EPD page's PAMT entry as PT_EPT.
7. Set the parent EPT entry to point to the new EPT or EPD page.

Figure 9.12 below shows the typical state of a 1GB GPA range after TDH.MEM.PAGE.DEMOTE.

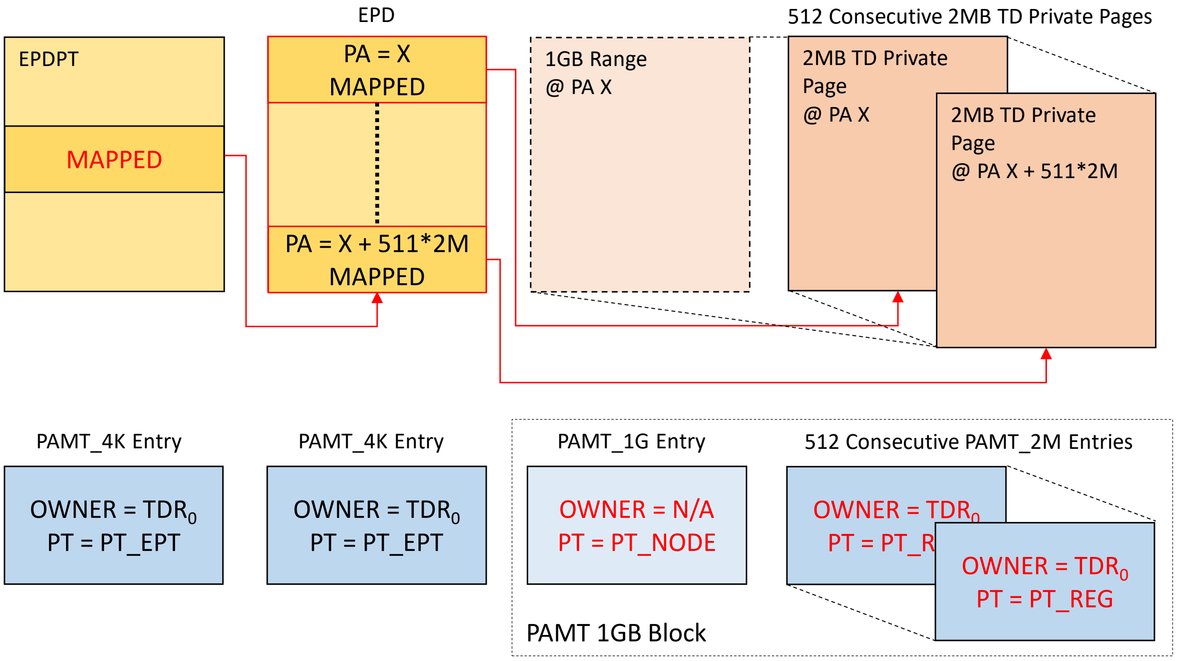


Figure 9.12: Typical State of a 1GB TD Private Range after TDH.MEM.PAGE.DEMOTE

TDH.MEM.PAGE.DEMOTE supports demotion of PENDING pages.

Refer also to the software flow described in 3.3.1.4.

9.13. Relocating TD Private Pages: TDH.MEM.PAGE.RELOCATE

The host VMM can relocate a 4KB TD private page to another HPA using TDH.MEM.PAGE.RELOCATE. This is useful for, e.g., physical address space de-fragmentation. The host VMM must first perform the TLB tracking protocol on the page.

The host VMM should first call TDH.MEM.RANGE.BLOCK on the target page. TDH.MEM.RANGE.BLOCK marks the parent EPT entry for that page as **BLOCKED** (if it was MAPPED) or **PENDING_BLOCKED** (if it was PENDING) and records the TD epoch in the PAMT entry of the page.

Typically, the host VMM then calls TDH.MEM.TRACK and performs a round of IPIs. After that, there should be no active address translation to target page.

The actual relocation is done by TDH.MEM.PAGE.RELOCATE which has the following inputs:

- The page GPA
- The target HPA to which the page will be relocated

At a high level, TDH.MEM.PAGE.RELOCATE works as follows:

1. Check the TD keys are configured.
2. Check the TD has been initialized.
3. Check the target physical page is marked as free in the PAMT.

4. Perform a pseudo Secure EPT walk to locate the parent Secure EPT leaf entry that maps the TD private page. Check that the entry has been blocked and get the current HPA.
5. Check the TLB tracking condition for the page.

If all checks pass, TDH.MEM.PAGE.RELOCATE does the following:

6. Copy the current physical page to the target physical page using direct writes (MOVDIR64B).
7. Mark the PAMT entry of the old physical page as PT_NDA.
8. Mark the PAMT entry of the target page as PT_REG.
9. Update the Secure EPT entry with the new physical page HPA. Set its state to MAPPED or PENDING depending on whether its previous state was BLOCKED or PENDING_BLOCKED, respectively.

Once complete, the old physical page should be free for use by the VMM for any purpose. To help avoid stability issues caused by cache line aliasing, the VMM should also assure that no cache lines associated with the old page are in a Modified state. This is typically done by calling TDH.PHYMEM.PAGE.WBINVD.

9.14. Removing TD Private Pages: TDH.MEM.PAGE.REMOVE

The host VMM can remove TD private pages using TDH.MEM.PAGE.REMOVE, freeing them for any use. 4KB, 2MB and 1MB pages can be removed – no demotion is required for large pages. The host VMM should first perform the TLB tracking protocol on the page.

The host VMM should first call TDH.MEM.RANGE.BLOCK on the target page. TDH.MEM.RANGE.BLOCK marks the parent EPT entry for that page as **BLOCKED** (if it was MAPPED) or **PENDING_BLOCKED** (if it was PENDING) and records the TD epoch in the PAMT entry of the page.

Typically, the host VMM then calls TDH.MEM.TRACK and performs a round of IPIs. After that, there should be no active address translation to target page.

The actual removal is done by TDH.MEM.PAGE.REMOVE which has the following inputs:

- The page GPA
- The page level (4KB, 2MB or 1GB)

At a high level, TDH.MEM.PAGE.REMOVE works as follows:

1. Check the TLB tracking condition for the page.
2. Check that the mapping size of the page fits the input parameter.

If all checks pass, TDH.MEM.PAGE.REMOVE does the following:

3. Mark the EPT entry for the target page as FREE.
4. Mark the PAMT entry of the page as PT_NDA.

Once complete, the physical page should be free for use by the VMM for any purpose. To help avoid stability issues caused by cache line aliasing, the VMM should also assure that no cache lines associated with the removed page are in a Modified state. This is typically done by calling TDH.PHYMEM.PAGE.WBINVD.

Refer also to the software flow described in 3.3.1.2.

9.15. Removing a Secure EPT Page: TDH.MEM.SEPT.REMOVE

The host VMM can remove a Secure EPT page using TDH.MEM.SEPT.REMOVE, freeing it for any use, provided all its entries are FREE. The host VMM should first perform the TLB tracking protocol on the page.

The host VMM should first call TDH.MEM.RANGE.BLOCK on the Secure EPT page. TDH.MEM.RANGE.BLOCK marks the parent EPT entry for that page as **BLOCKED** and records the TD epoch in the PAMT entry of the page.

Typically, the host VMM then calls TDH.MEM.TRACK and performs a round of IPIs. After that, there should be no active address translation to GPA range presented by the Secure EPT page to be removed.

The actual removal is done by TDH.MEM.SEPT.REMOVE which has the following inputs:

- The Secure EPT page GPA
- The EPT level

At a high level, TDH.MEM.SEPT.REMOVE works as follows:

1. Check the TLB tracking condition for the page.

2. Check that the mapping size of the page fits the input parameter.
3. Check that all 512 entries of the Secure EPT page are PT_NDA.

If all checks pass, TDH.MEM.SEPT.REMOVE does the following:

4. Mark the EPT entry for the Secure EPT page as FREE.
5. Mark the PAMT entry of the Secure EPT page as PT_NDA.

Once complete, the physical page should be free for use by the VMM for any purpose. To help avoid stability issues caused by cache line aliasing, the VMM should also assure that no cache lines associated with the page are in a Modified state. This is typically done by calling TDH.PHYMEM.PAGE.WBINVD.

9.16. Unblocking a GPA Range: TDH.MEM.RANGE.UNBLOCK

The host VMM can unblock previously blocked TD private GPA ranges using TDH.MEM.RANGE.UNBLOCK, returning them to their original state. 4KB, 2MB and 1MB GPA ranges can be unblocked.

The host VMM should first complete the TLB tracking protocol on the GPA range. It typically calls TDH.MEM.TRACK and performs a round of IPIs. After that, there should be no active address translation to target page.

The actual unblocking is done by TDH.MEM.RANGE.UNBLOCK which has the following inputs:

- The GPA
- The GPA range level (4KB, 2MB or 1GB)

At a high level, TDH.MEM.RANGE.UNBLOCK works as follows:

1. Check the TLB tracking condition for the GPA range.
2. Check that the mapping size of the GPA range fits the input parameter.

If all checks pass, TDH.MEM.RANGE.UNBLOCK does the following:

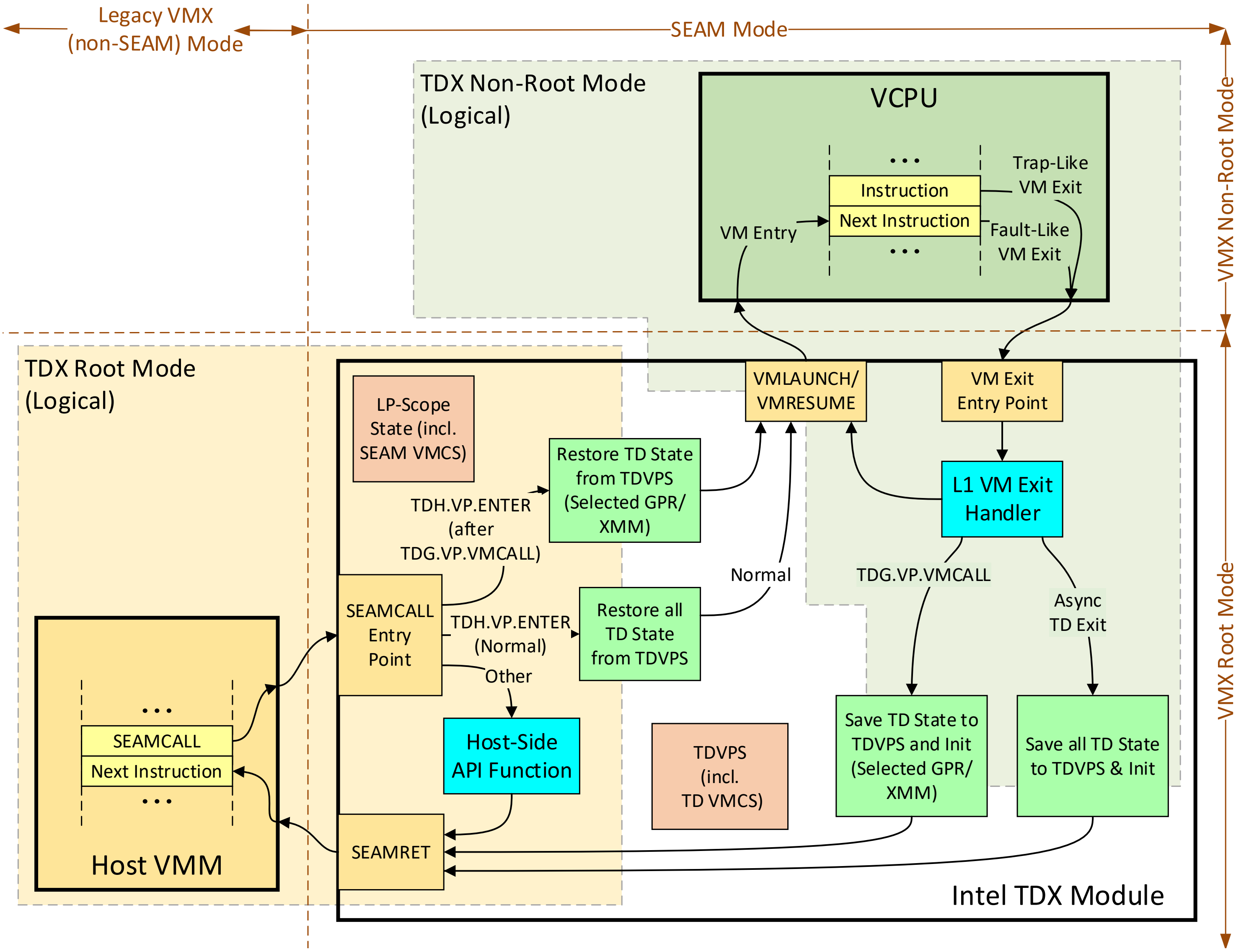
3. Mark the EPT entry for the target GPA as MAPPED (if it was BLOCKED) or PENDING (if it was PENDING_BLOCKED).

Refer also to the software flow described in 3.3.1.5.

10. TD VCPU

This chapter discusses multiple items related to TD VCPUs.

10.1. VCPU Transitions



5 **Figure 10.1: TD VCPU Transitions Overview**

10.1.1. Initial TD Entry, Asynchronous TD Exit and Subsequent TD Entry

On the initial TD entry to a TD VCPU, the TDX module restores the initial TD VCPU state from TDVPS (including TD VMCS).

Following a successful TDH.VP.ENTER, asynchronous TD exit may happen as a result of events such as interrupts, EPT violations etc. In such case, the TDX module saves the TD VCPU state into TDVPS (including TD VMCS). Most of the host VMM VCPU state that may have been used by the TD is initialized. For a detailed description of VMM state following TDH.VP.ENTER, see the [TDX Module ABI].

On the subsequent TD entry following an asynchronous TD exit, the TDX module restores the TD VCPU state from TDVPS (including TD VMCS). The host VMM does not impact the VCPU state except in one case: a trap-like asynchronous TD exit from a guest-side interface function may indicate that the host VMM can apply a recoverability hint in the following TD entry. In this case, the host VMM provides a recoverability hist to the guest TD, which is combined into the guest-side interface function’s completion status returned in RAX.

Section 2: Introduction and Overview

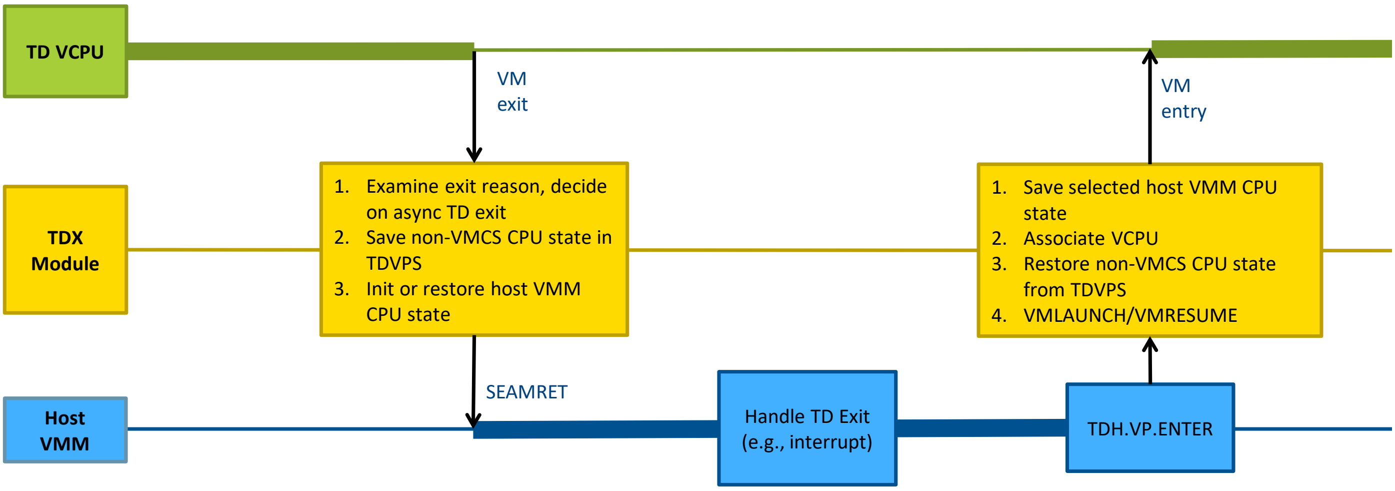

TD VCPU
TDX Module
Host VMM
VM exit
VM entry
1. Examine exit reason, decide on async TD exit
2. Save non-VMCS CPU state in TDVPS
3. Init or restore host VMM CPU state
1. Save selected host VMM CPU state
2. Associate VCPU
3. Restore non-VMCS CPU state from TDVPS
4. VMLAUNCH/VMRESUME
SEAMRET
Handle TD Exit (e.g., interrupt)
TDH.VP.ENTER

Figure 10.2: Example of Asynchronous TD Exit and TD Resumption

10.1.2. Synchronous TD Exit and Subsequent TD Entry

TDG.VP.VMCALL provides a channel for the guest TD to communicate with the host VMM.

The guest TD can initiate a synchronous TD exit by invoking TDG.VP.VMCALL. The RCX input parameter of selects the GPRs (from RBX, RDX, RBP, RDI, RSI and R8 through R15) and XMM registers whose value is passed through to the host VMM as the output of TDH.VP.ENTER. RCX itself is passed as-is to the output of TDH.VP.ENTER. Other CPU state components, including GPRs and XMM registers not selected by RCX, are saved in TDVPS and set to fixed values.

On the subsequent TDH.VP.ENTER, the RCX value that was used for TDG.VP.VMCALL selects the GPRs (from RBX, RDX, RBP, RDI, RSI and R8 through R15) and XMM registers whose value is passed through to the guest TD. Other CPU state components, including GPRs and XMM registers not selected by RCX, are restored from RCX.

For details, see the TDH.VP.ENTER and TDG.VP.VMCALL definitions in the [TDX Module ABI].

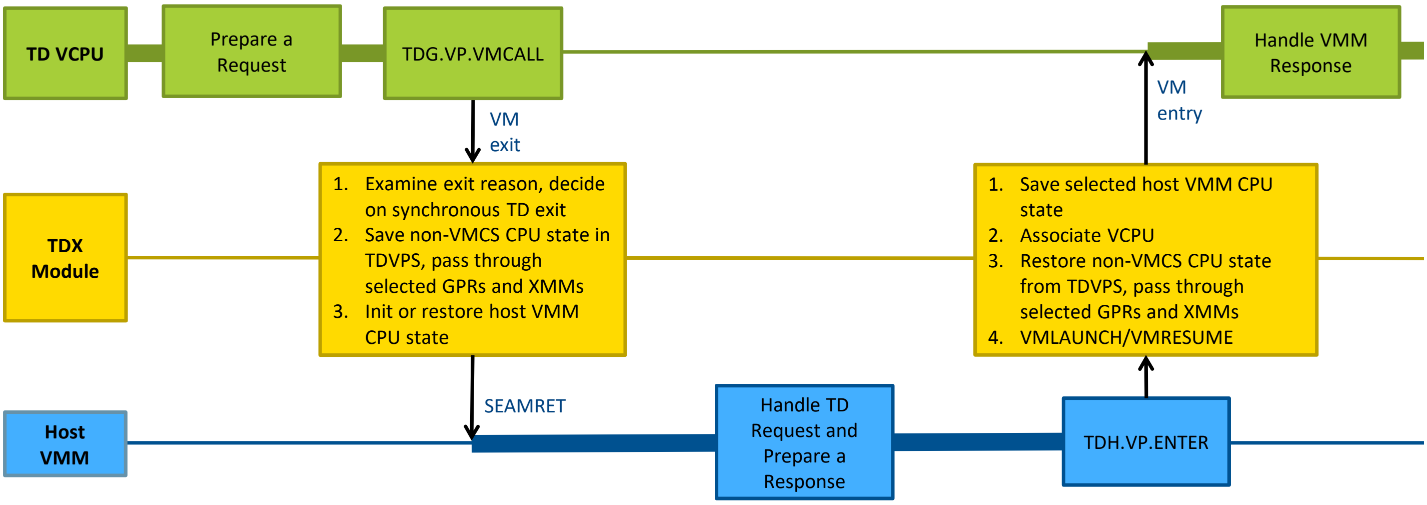

TD VCPU
Prepare a Request
TDG.VP.VMCALL
Handle VMM Response
TDX Module
Host VMM
VM exit
VM entry
1. Examine exit reason, decide on synchronous TD exit
2. Save non-VMCS CPU state in TDVPS, pass through selected GPRs and XMMs
3. Init or restore host VMM CPU state
1. Save selected host VMM CPU state
2. Associate VCPU
3. Restore non-VMCS CPU state from TDVPS, pass through selected GPRs and XMMs
4. VMLAUNCH/VMRESUME
SEAMRET
Handle TD Request and Prepare a Response
TDH.VP.ENTER

Figure 10.3: Example of Synchronous TD Exit and TD Resumption

10.1.3. UPDATED: VCPU Activity State Machine

The VCPU activity state machine, controlled by TDVPS.VCPU_STATE as shown in Table 10.1 below and shown in Figure 10.4 below, helps assure the following:

- A VCPU can be entered only when its logical TDVPS control structure, composed of TDVPR and TDCX pages, is available in memory and has been initialized by TDH.VP.INIT or successfully imported by TDH.IMPORT.STATE.VP.
- A VCPU can be entered only if its state is consistent (no non-recoverable TD exit happened).
- TD entry is done properly, depending on whether it is the first entry or on the last TD exit type.

Table 10.1: TDVPS.VCPU_STATE Definition

State Name	Description
VCPU_UNINITIALIZED	VCPU has not been initialized yet by TDH.VP.INIT.
VCPU_IMPORT	The VCPU state has been incompletely imported.
VCPU_READY	The VCPU is ready to be executed.
VCPU_ACTIVE	VCPU is active (TDX non-root mode) on some LP.
VCPU_DISABLED	VCPU is being torn down.

TD Entry and TD Exit transitions normally toggle between the VCPU_READY state and the VCPU_ACTIVE state, except when a non-recoverable VCPU TD Exit (due to a Triple Fault) transitions to a VCPU_DISABLED state.

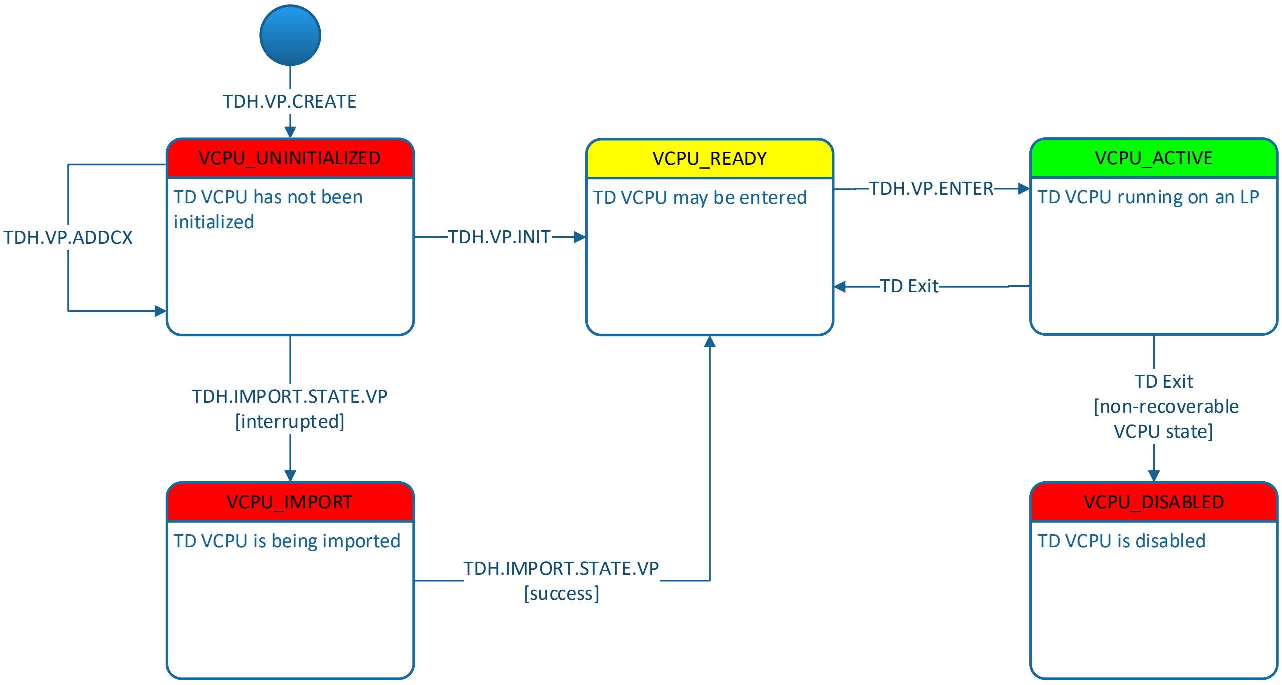


Figure 10.4: VCPU Activity State Machine

In the VCPU_READY and VCPU_IMPORT states, a LAST_TD_EXIT sub-state indicates what was the last TD exit and how a subsequent TD entry should be done.

Table 10.2: TDVPS.LAST_TD_EXIT Definition

Name	Description
ASYNC_FAULT	Last TD exit was due to an asynchronous event (non-TDG.VP.VMCALL) which caused a fault-like exit, i.e., the VCPU state is as if the guest instruction has not been executed. VCPU state has been fully saved on TD exit and will be restored on the next TD entry.
ASYNC_TRAP	Last TD exit was due to an asynchronous event that happen as part of a guest-side interface function (non-TDG.VP.VMCALL) which caused a trap-like exit, i.e., the VCPU state is as if the guest instruction has been executed. VCPU state has been fully saved on TD exit and will be restored on the next TD entry. On the next TD entry, the host VMM provides the guest with a recoverability hint.
TDVMCALL	Last TD exit was due to a TDG.VP.VMCALL. On the next TD entry, most GPR and all XMM state will be forwarded to the guest TD from the host VMM.

10.2. TD VCPU TLB Address Space Identifier (ASID)

Non-root mode cached address translations are tagged with unique Address Space Identifiers (ASIDs). The goal of TD ASIDs is to reduce the need to flush TLB entries on TD Entry and TD Exit due the associated performance costs as a result of the flushing.

10.2.1. TD ASID Components

Table 10.3 below shows a high-level view of the components of the TD ASID. The exact structure is micro-architectural.

Table 10.3: TD ASID

Field	Size (Bits)	Description and TDX Usage
SEAM	1	This is an implicit bit 16 of VPID not directly visible to software. It is set to 1 by the CPU in SEAM mode. This bit prevents overlap with legacy (non-TDX) ASIDs.
VPID	16	Set by the Intel TDX module to VCPU_INDEX, a unique index of a VCPU in a TD, plus 1 (since VCPU_INDEX starts with 0 which is not a value VPID number for non-root mode)
EPTP	40	Bits [51:12] of the EPTP, which for a TD points to the **Secure** EPT root – HKID bits are cleared to 0 Note that EPTP is unique per TD and is used as an ASID component for **both Secure EPT and Shared EPT** translations caching.
PCID	16	Same as legacy PCID

10.2.2. INVEPT by the Host VMM for Managing the Shared EPT

The same ASID based on the TD's EPTP is used for caching both secure and shared EPT translations (remember: EPTP is the HPA of the **secure** EPT root page). Thus, to flush shared EPT translations, the host VMM uses INVEPT specifying the TD's EPTP, not its Shared EPTP. The host VMM can obtain the value of EPTP from the TD VMCSs using TDH.VP.RD.

10.3. VCPU-to-LP Association

10.3.1. Non-Coherent Caching

Some TD VCPU state is non-coherently cached. This includes:

- Address translations (TLB/PxE entries) must be explicitly flushed in case they may be stale.
- TD VMCS is cached by the CPU. VMX architecture requires making a VMCS current by VMPTRLD before using it with most VMX instructions, and then explicitly writing it to memory and making it non-current by VMCLEAR before the VMCS memory image can be handled (e.g., by making it current on another LP).

This non-coherent caching implies that some explicit and/or implicit operations are done to help guarantee correctness. This is described in the following sections.

10.3.2. Intel TDX Functions for VCPU-LP Association and Dis-Association

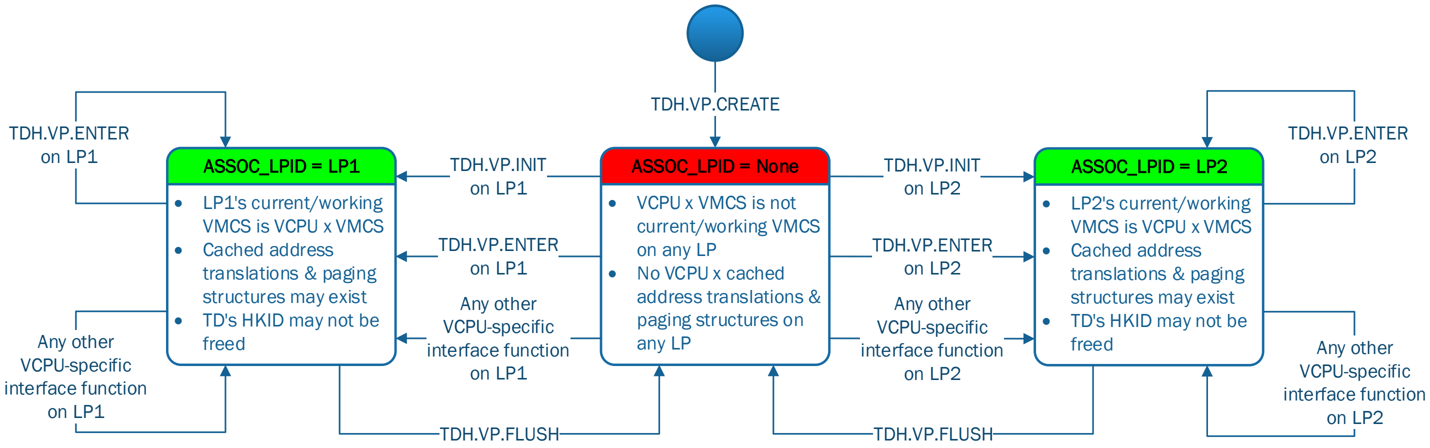

TDH.VP.CREATE
ASSOC_LPID = None
VCPU x VMCS is not current/working VMCS on any LP
No VCPU x cached address translations & paging structures on any LP
ASSOC_LPID = LP1
LP1's current/working VMCS is VCPU x VMCS
Cached address translations & paging structures may exist
TD's HKID may not be freed
ASSOC_LPID = LP2
LP2's current/working VMCS is VCPU x VMCS
Cached address translations & paging structures may exist
TD's HKID may not be freed
TDH.VP.ENTER on LP1
TDH.VP.INIT on LP1
TDH.VP.ENTER on LP1
Any other VCPU-specific interface function on LP1
Any other VCPU-specific interface function on LP1
TDH.VP.INIT on LP2
TDH.VP.ENTER on LP2
Any other VCPU-specific interface function on LP2
TDH.VP.ENTER on LP2
Any other VCPU-specific interface function on LP2
TDH.VP.FLUSH
TDH.VP.FLUSH

Figure 10.5: VCPU Association State Machine

The following Intel TDX module mechanisms are designed to help assure correct and secure operation:

- TD VCPU to LP association is many-to-one. A TD VCPU can be associated with at most one LP at any given time. An LP may be associated with multiple VCPUs.
- VCPU to LP association is implicitly done by any VCPU-specific SEAMCALL flow, including TDH.VP.ENTER. Those flows check that the VCPU is either already associated with the current LP or is not associated with any LP.
- If the host VMM wishes to associate a VCPU with another LP, it must explicitly flush the VCPU state on the LP currently associated with it using **TDH.VP.FLUSH**. This function performs TD ASID, and extended paging structure (EPxE) caches TLB flush and VMCLEAR. For details, see the [TDX Module ABI].
- If the VMM wishes to reclaim the TD's private HKID, thus making the TDVPS memory inaccessible, it must explicitly flush the VCPU state on the LP currently associated with it. This is described in 5.4.

10.3.3. Performance Considerations

- Migrating VCPUs between LPs is costly. As described above, it involves flushing address translation caches, paging structure caches and VMCS cache. The host VMM should minimize that for best performance.
- Address translation and paging structure caches are flushed at TD-scope on the current LP. This flushing impacts the (possibly non-typical) case where multiple VCPUs of the same TD are associated with a single LP.

11.UPDATED: CPU Virtualization (Non-Root Mode Operation)

This chapter describes how the Intel TDX module virtualizes the CPU to a guest TD.

11.1. Initial State

Intel SDM, Vol. 3, 9.1.1 Processor State after Reset

11.1.1. Overview

As designed, most of the TD VCPU initial state is the same as the processor architectural state after INIT. However, there are some differences:

- The TD VCPU starts its life in protected (32-bit) non-paged mode, not in real mode. It is allowed only to switch to 64b mode. This impacts the initial state of segment registers, CRs and MSRs. Mode restrictions in TDX non-root mode are described in 11.1.
- The IA32_EFER MSR is initialized to support the CPU modes described in 11.1.
- The initial values of some GPRs provide some basic information to the guest TD as described in 11.1.2 below. This information should be sufficient for the vBIOS to set up paging tables and switch as soon as possible to 64b mode, where it can use the TDCALL leaf functions.

See also the TDVPS fields and TD VMCS guest state area in the [TDX Module ABI].

11.1.2. Initial State of Guest TD GPRs

As designed, the following initial state is different than the architectural INIT state:

Table 11.1: Initial Values of GPRs Different from their Architectural INIT Values

Register	Bits	Initial Value
RBX	5:0	GPAW, the effective GPA width (in bits) for this TD (do not confuse with MAXPA) – SHARED bit is at GPA bit GPAW-1 Only GPAW values 48 and 52 are possible.
	63:6	Reserved: set to 0
RCX, R8	63:0	The value of RCX and R8 is provided as an input to TDH.VP.INIT (the same value in both GPRs). No checking is done on this value; the intention is for vBIOS to read RCX immediately after the first TDH.VP.ENTER, and use the RCX value appropriately as set by software convention.
RDX	31:0	Set to the virtualized Family/Model/Stepping returned by CPUID(1).EAX. The value is calculated by TDH.SYS.INIT as to have the minimum Stepping ID across all packages.
	63:32	Reserved: set to 0
RSI	31:0	Virtual CPU index, starting from 0 and allocated sequentially on each successful TDH.VP.INIT
	63:32	Reserved: set to 0
RIP	63:0	Set to 0xFFFFFFF0 (i.e., 4GB - 16B)

11.1.3. Initial State of CRs

As designed, the following initial state is different than the architectural INIT state:

- CR0 is initialized to 0x0021 – bits PE (0) and NE (5) are set to 1, and all other bits are cleared to 0. See 11.6.1 for details.
- CR4 is initialized to 0x2040 – bits MCE (6) and VMXE (13) are set to 1, and all other bits are cleared to 0. Note that the virtualized value of VMXE is 0, due to the setting of the TD VMCS "CR4 guest/host mask" and "CR4 read shadow" controls. See 11.6.2 for details.

11.1.4. Initial State of Segment Registers

As designed, the following initial state is different than the architectural INIT state:

- CS, DS, ES, FS, GS and SS are initialized with a base of 0 and limit of 0xFFFFFFFF.
- LDTR, TR and GDTR are initialized with a base of 0 and limit of 0xFFFF.
- IDTR is initialized as invalid (limit of 0).

For details, see the [TDX Module ABI].

11.1.5. Initial State of MSRs

As designed, the following initial state is different than the architectural INIT state:

- IA32_EFER is initialized to 0x901 – SCE (bit 0), LME (bit 8) and NXE (bit 11) are set to 1, and all other bits are cleared to 0.

11.2. UPDATED: Guest TD Run Time Environment Enumeration

Guest software can be designed to run either as a TD, as a legacy virtual machine, or directly on the CPU, based on enumeration of its run-time environment. Figure 11.1 below shows a typical flow used by guest software.

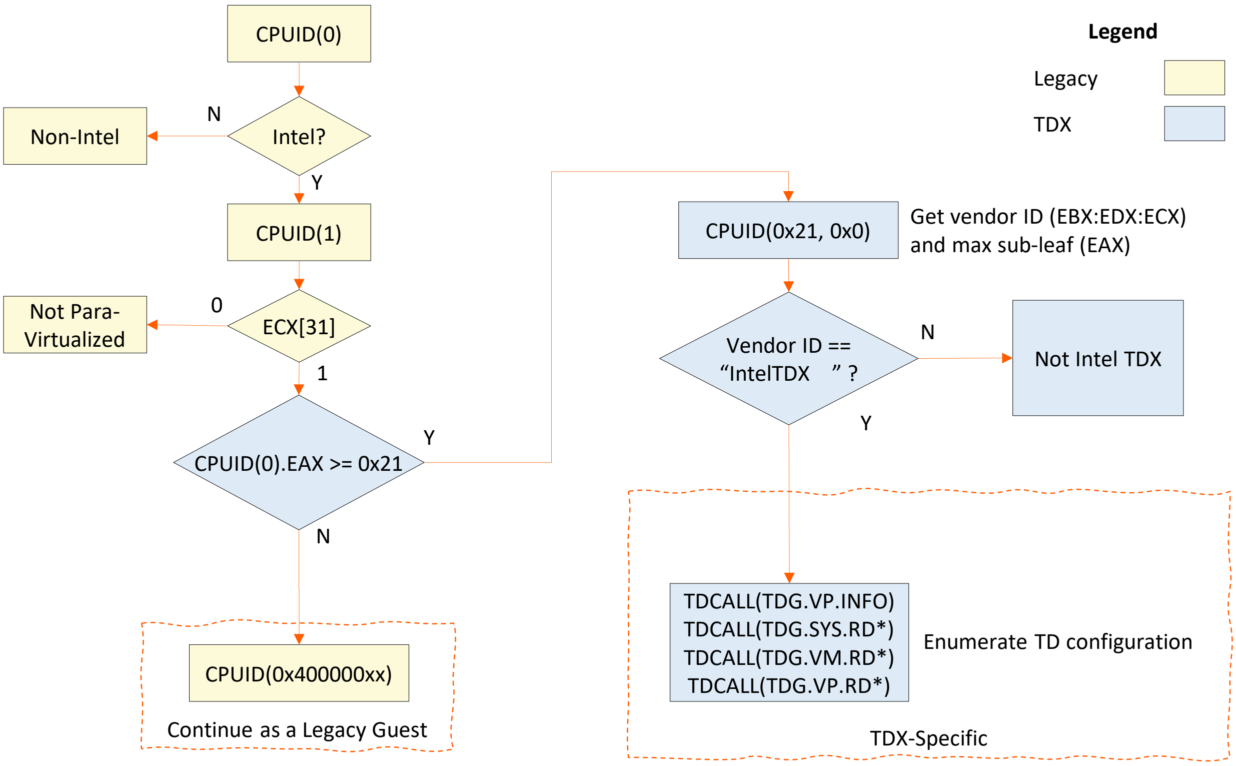


Figure 11.1: UPDATED: Typical Run-Time Environment Enumeration by a Guest TD

CPUID leaf 0x21 emulation is done by the Intel TDX module. Sub-leaf 0 returns the values shown below. Other sub-leaves return 0 in EAX/EBX/ECX/EDX.

Table 11.2: TDX Enumeration by CPUID(0x21,0)

GPR	Value (Hex)	Description
EAX	0x00000000	Maximum sub-leaf number
EBX	0x65746E49	“Inte”
ECX	0x20202020	“ “
EDX	0x5844546C	“lTDX”

Once the guest software discovers that it runs as a TD, it can use TDG.VP.INFO to get basic information. It can also use the metadata read functions TDG.SYS.RD*, TDG.VM.RD* and TDG.VP.RD*.

11.3. CPU Mode Restrictions

Intel SDM, Vol. 3, 2.2	Modes of Operation
Intel SDM, Vol. 3, 9.8.5	Initializing IA-32e Mode
Intel SDM, Vol. 3, 11.5.1	Cache Control Registers and Bits
Intel SDM, Vol. 3, 24.6.6	Guest/Host Masks and Read Shadows for CR0 and CR4

A TD OS running in TDX non-root mode is required to be a 64-bit OS. The Intel TDX module helps enforce this with the restrictions described below.

Table 11.3: CPU Mode Restrictions in TDX Non-Root Mode

Restriction	Description
CPU and Paging Modes	In TDX non-root mode, the CPU is allowed to run in the following modes: • Protected mode (32-bit) with no paging (CR0.PG == 0) • IA-32e mode with 4-level or 5-level paging (CR0.PG == 1), with the sub-modes controlled by CS.L: ○ 64-bit mode ○ Compatibility (32-bit) mode To achieve this, CR0.PE and IA32_EFER.LME are enforced to 1, as described in the following sections.
Execute Disable	When running in IA-32e mode, the PT Execute Disable bit (63) is always enabled. To achieve this, IA32_EFER.NXE is enforced to 1, as described in the following sections.
Caching is Always Enabled	The guest TD runs in Normal Cache Mode. To achieve this, CR0.CD and CR0.NW are enforced to 0, as described in the following sections.

11.4. Instructions Restrictions

The Intel TDX module is designed to block certain instructions from executing in TDX non-root mode. Execution of those instructions results in a VM exit to the Intel TDX module, which then injects either a #UD or a #VE to the guest TD, as described in 11.10.

Execution of other instructions may be conditionally blocked, depending on feature enabling, as described in the following sections.

11.4.1. Instructions that Cause a #UD Unconditionally

- ENCLS, ENCLV
- Most VMX instructions: INVEPT, INVVPID, VMCLEAR, VMFUNC, VMLAUNCH, VMPTRLD, VMPTRST, VMREAD, VMRESUME, VMWRITE, VMXOFF, VMXON
- RSM
- GETSEC
- SEAMCALL, SEAMRET

11.4.2. Instructions that Cause a #VE Unconditionally

- String I/O (INS*, OUTS*), IN, OUT
- HLT
- MONITOR, MWAIT
- WBINVD, INVD
- VMCALL

11.4.3. Instructions that Cause a #UD or #VE Depending on Feature Enabling

- PCONFIG (see 11.15)

11.4.4. Other Cases

- Guest TD execution of ENQCMD results in a #GP(0).
- Guest TD execution of ENQCMDS when CPL is 0 results in a #UD. Otherwise it results in a #GP(0).

11.5. Extended Feature Set

Intel SDM, Vol. 1, 13	Managing State Using the XSAVE Feature Set
Intel SDM, Vol. 3, 13	System Programming for Instruction Set Extensions and Processor Extended State

11.5.1. Allowed Extended Features Control

At the guest TD scope, **TDCS.XFAM (Extended Features Allowed Mask)** is provided as an input during guest TD build process. XFAM is a 64b mask, using the **state-component bitmap** format used by extended state ISA (XSAVE, XRSTOR, XCR0, IA32_XSS etc.), which specifies the set of extended features the TD is allowed to use.

XFAM is checked to be compliant with the set of extended features supported by the CPU, as enumerated by CPUID and the allowed bit combinations, as shown in Table 11.4 below.

11.5.2. Extended State Isolation

The Intel TDX module helps assure that any guest TD extended state is saved and isolated from the host VMM across TD exit and entry. It is the VMM's responsibility to save its own extended state across TD entry and exit.

- Before TDH.VP.ENTER, the host VMM should save (e.g., using XSAVES) any extended state that the guest TD VCPU is allowed to use (per XFAM) and the host VMM expects to need after TDH.VP.ENTER is complete.
- The TDH.VP.ENTER function loads the extended state that the TD VCPU is allowed to use, per XFAM, from the VCPU's TDVPS. An exception to this is when TDH.VP.ENTER follows a previous TDG.VP.VMCALL – in the case TDH.VP.ENTER does not load the XMM state (corresponding to XFAM bit 1) from TDVPS, but passes it directly from the host VMM.
- On an asynchronous TD exit, the Intel TDX module saves the extended state that the TD VCPU was allowed to use, per XFAM, to the VCPU's TDVPS. It then clears the extended state.
- On TDG.VP.VMCALL, the Intel TDX module works similarly, but it selectively does not clear some of the XMM register state (corresponding to XFAM bit 1). That XMM state is passed directly to the host VMM.
- On completion of TDH.VP.ENTER (following TD exit), the VMM may restore any extended state that it saved before TDH.VP.ENTER.

11.5.3. Extended Features Execution Control

The Intel TDX module is designed to prohibit the guest TD from using any extended feature not allowed by XFAM. Many extended state features are controlled by XCR0 and IA32_XSS MSR. Other features are controlled by CR4 or by specific MSRs.

XCR0 and IA32_XSS MSR On XSETBV, which attempts to write to XCR0, and on WRMSR of IA32_XSS, the TDX module emulates the architectural behavior of the CPU. The following cases cause a #GP(0):

- The new value is not natively valid for XCR0 or IA32_XSS (it sets reserved bits, sets bits for features not recognized by the Intel TDX module, or uses illegal bit combinations).
- The new value has any bits set that are not allowed by XFAM.

CR4 On MOV to CR4, the guest TD attempts to set bits not allowed according to XFAM will cause a #GP(0).

Other MSRs The guest TD attempts to write or read certain MSRs that are not enabled according to XFAM can cause a #GP(0) or a #VE, as described below.

The following table describes how a guest TD executes each of the extended features.

Table 11.4: Extended Features Enumeration and Execution Control

Bits	U/S	Feature	Enumeration[6]	XFAM Value	Description
0	U	FP	Always available	1	Always enabled
1	U	SSE	Always available	1	Always enabled
2	U	AVX	CPUID(0xD,0x0).EAX[2] CPUID(0x7,0x0).EBX[2] CPUID(0x7,0x0).ECX[10:9] CPUID(0x7,0x1).EAX[5] CPUID(0xD, 0x2).*	0 or 1	Execution is directly controlled by XCR0.
4:3	U	MPX	CPUID(0xD,0x0).EAX[4:3] CPUID(0x7,0x0).EBX[14] CPUID(0xD, 0x3).* CPUID(0xD, 0x4).*	00	MPX is being deprecated.
7:5	U	AVX512	CPUID(0xD,0x0).EAX[7:5] CPUID(0x7,0x0).EBX[31:30, 28:26, 21, 17:16] CPUID(0x7,0x0).ECX[14, 12:11, 6, 1] CPUID(0x7,0x0).EDX[8] CPUID(0x7,0x1).EAX[5] CPUID(0xD, 0x5).* CPUID(0xD, 0x6).* CPUID(0xD, 0x7).*	000 or 111	Execution is directly controlled by XCR0. AVX512 may be enabled only if AVX is enabled – i.e., XFAM[7:5] may be set to 111 only when XFAM[2] is set to 1.
8	S	PT (RTIT)	CPUID(0xD,0x1).ECX[8] CPUID(0x7,0x0).EBX[25] CPUID(0x14).* CPUID(0xD, 0x8).*	0 or 1	Execution is controlled by IA32_RTIT_CTL. If PT is enabled by XFAM, the guest TD is allowed access to all IA32_RTIT_* MSRs. Otherwise, any access causes #GP(0).
9	U	PK	CPUID(0xD,0x0).EAX[9] CPUID(0xD, 0x9).*	0 or 1	Execution is controlled by CR4.PKE (bit 22). If PK is disabled by XFAM, the guest TD is disallowed from setting CR4.PKE. An attempt to set this bit causes a #GP(0).
10	S	ENQCMD (PASID)	CPUID(0xD,0x1).ECX[10] CPUID(0xD, 0xA).*	0	Execution is controlled by IA32_PASID MSR. There is no direct I/O from guest TDs. ENQCMD and ENQCMDS from the guest TD are not supported and cause a #UD. Access to IA32_PASID causes a #GP(0).
12:11	S	CET	CPUID(0xD,0x1).ECX[12:11] CPUID(0xD, 0xB).* CPUID(0xD, 0xC).*	00 or 11	Execution is controlled by CR4.CET (bit 23). If CET is disabled by XFAM, the guest TD is disallowed from setting CR4.CET. An attempt to set this bit causes a #GP(0).

[6] An extended feature controlled by bits N:M is available if all bits in the range N:M returned by CPUID are set to 1.

Bits	U/S	Feature	Enumeration[6]	XFAM Value	Description
13	S	HDC	CPUID(0xD,0x1).ECX[13] CPUID(0xD, 0xD).*	0	Hardware Duty Cycle is controlled by package-scope IA32_PKG_HDC_CTL and LP-scope IA32_PM_CTL1 MSRs. HDC is disabled. Any guest TD access to the above MSRs causes a #VE.
14	S	ULI	CPUID(0xD,0x1).ECX[14] CPUID(0x7,0x0).EDX[5] CPUID(0xD, 0xE).*	0 or 1	Execution is controlled by CR4.UINT (bit 25). If ULI is disabled by XFAM, then: • The guest TD is disallowed from setting CR4.ULI. An attempt to set this bit causes a #GP(0). • The guest TD is disallowed access to all IA32_UINT_* MSRs. Any access causes a #GP(0).
15	S	LBR	CPUID(0xD,0x1).ECX[15] CPUID(0x7,0x0).EDX[19] CPUID(0xD, 0xF).* CPUID(0x1C).*	0 or 1	Execution is controlled by IA32_LBR_CTL. If LBR is disabled by XFAM, the guest TD is disallowed access to all IA32_LBR_* MSRs. Any access causes a #GP(0).
16	S	HWP	CPUID(0xD,0x1).ECX[16] CPUID(0xD, 0x10).*	0	Execution of Hardware-Controlled Performance State is controlled by IA32_HWP MSRs. This feature is disabled. Access to any of the above MSRs causes a #VE.
18:17	U	AMX	CPUID(0xD,0x0).EAX[18:17] CPUID(0xD, 0x11).* CPUID(0xD, 0x12).*	00 or 11	Advanced Matrix Extensions (AMX) is directly controlled by XCR0.

11.6. UPDATED: CR Handling

11.6.1. CR0

Intel SDM, Vol. 3, 2.5	Control Registers
Intel SDM, Vol. 3, 23.8	Restrictions on VMX Operation
Intel SDM, Vol. 3, 24.6.6	Guest/Host Masks and Read Shadows for CR0 and CR4
Intel SDM, Vol. 3, 25.6	Unrestricted Guests

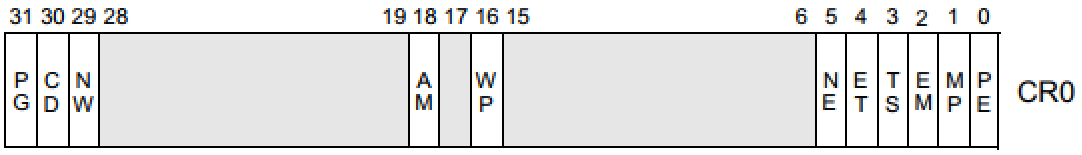


Figure 11.2: CR0

From the guest TD's point of view, as virtualized by the Intel TDX module, CR0 bits PE (0) and NE (5) are always set to 1, and bits NW (29) and CD (30) are always cleared to 0.

Guest TD writes to CR0 are handled by the Intel TDX module as follows:

- Writes to CR0 that are architecturally illegal (such as attempts to set bits that must be 0), or writes to CR0 that set architecturally illegal bit combinations, result in a #GP(0).
- Writes to CR0 that are architecturally illegal, but not permitted by the TDX architecture (such as clearing CR0.CD) result in a #VE.
- Other writes are allowed.

For TD migration, the same rules are used for checking the imported value of guest CR0. Any violation results in a failed import.

11.6.2. UPDATED: CR4

Intel SDM, Vol. 3, 24.6.6 Guest/Host Masks and Read Shadows for CR0 and CR4

If a CPU feature is not enabled for the guest TD, the guest TD's attempt to set the corresponding CR4 bit can result in a #GP(0):

1. Depending on the TD's XFAM, guest TD modification of CR4 bits PKE (22), CET (23) and UINT (25) is prevented. Any guest TD attempt to change those bits results in a #GP(0).
2. If the TD's ATTRIBUTES.KL is 0, guest TD attempts to set bit KL (19) results in a #GP(0).
3. If the TD's ATTRIBUTES.PKS is 0, guest TD attempts to set bit PKS (24) results in a #GP(0). See 11.14 below.
4. If the TD's ATTRIBUTES.PERFMON is 0, guest TD attempts to set bit PCE (8) results in a #GP(0). See 15.2.

In addition, any guest TD attempts to modify any of the architecturally reserved CR4 bits, or to set architectural-illegal bit combinations, can result in a #GP(0).

From the guest TD's point of view, the following bits are virtualized as fixed by Intel TDX module. Guest TD attempts to modify their values result in a #VE:

- CR4 bit MCE (6) is fixed to 1.
- CR4 bits VMXE (13) and SMXE (14) are fixed to 0.

For TD migration, the same rules are used for checking the imported value of guest CR4. Any violation results in a failed import.

11.7. MSR Handling

11.7.1. Overview

From the guest TD's point of view, as virtualized by the Intel TDX module, MSRs are divided into the following categories:

- MSRs that are context-switched on TD entry and exit – guest TD access to such MSRs may be full, partial or none
- MSRs that are not context-switched, but guest TD access is read-only
- MSRs that are not context-switched, and are inaccessible to the guest TD

MSR behavior can be either fixed or dependent on the TD configuration via the XFAM, ATTRIBUTES and CPUID configuration parameters. The host VMM has no direct interface to configure specific MSR behavior (e.g., it cannot set a specific MSR to TD exit on write). Instead, guest TD access violations to MSRs can cause a #GP(0) in most cases where the MSR is enumerated as inaccessible by the Intel TDX module via CPUID virtualization. In other cases, guest TD access violations to MSRs can cause a #VE. A guest TD that wishes to access an MSR that is not allowed by the Intel TDX module should do so via explicit requests from the host VMM using TDCALL(TDG.VP.VMCALL).

A detailed list of MSR virtualization is provided in the [TDX Module ABI].

11.8. UPDATED: CPUID Virtualization

11.8.1. UPDATED: CPUID Configuration by the Host VMM

For some CPUID leaves and sub-leaves, the virtualized bit fields of CPUID return values (in guest EAX/EBX/ECX/EDX) are configurable by the host VMM. For such cases, the Intel TDX module architecture defines two virtualization types:

Table 11.5: Host VMM Configurable CPUID Field Virtualization

CPUID Field Virtualization	Description	Comments
As Configured	Bit fields for which the host VMM configures the value seen by the guest TD. Configuration is done on TDH.MNG.INIT.	

CPUID Field Virtualization	Description	Comments
As Configured (if Native)	Bit fields for which the host VMM configures the value such that the guest TD either sees their native value or a value of 0. Configuration is done on TDH.MNG.INIT.	If a CPUID bit enumerates a CPU feature, and the feature is natively supported, then the feature can either be allowed by the host VMM, or it will be effectively deprecated for the guest TD.

The above CPUID fields can be specified by the host VMM at guest TD initialization time TDH.MNG.INIT using the TD_PARAMS input structure of TDH.MNG.INIT. TDH.MNG.INIT and its input TD_PARAMS structure are described in the [TDX Module ABI]. Configuration is further classified as follows:

Table 11.6: CPUID Configuration by the TD_PARAMS Input of TDH.MNG.INIT

TD_PARAMS Section	Description	Notes
CPUID_CONFIG	Bit fields configurable directly based on a configuration table	Some bit fields are configurable by both CPUID_CONFIG and XFAM. See the discussion below.
XFAM	Bit fields configurable based on the guest TD's XFAM XFAM control of extended features virtualization is described in 11.5.	
ATTRIBUTES	Bit fields configurable based on the guest TD's ATTRIBUTES	
Other	Bits fields configurable based on some other field of TD_PARAMS	

Some CPUID bit fields are configurable based on both XFAM and CPUID_CONFIG sections of TD_PARAMS. This is intended to support **fine-grained virtualization of sub-features of extended features**. E.g., it allows the host VMM to virtualize some AVX512 as available, but to virtualize some AVX512 instructions as unavailable. This is useful for TD migration, as it allows the host VMM to configure a common subset of supported sub-features.

A detailed list of CPUID virtualization is provided in the [TDX Module ABI]. For any valid CPUID leaf / sub-leaf combination that is not listed, the Intel TDX module injects a #VE.

The host VMM should always consult the list of directly configurable CPUID leaves and sub-leaves, as enumerated by TDH.SYS.RD/RDALL or TDH.SYS.INFO, described in 4.1.3.3.

11.8.2. Guest TD Control of CPUID Virtualization

11.8.2.1. *Unconditional #VE for all CPUID Leaves and Sub-Leaves*

The guest TD may toggle on or off the unconditional injection of #VE on all CPUID leaves and sub-leaves, per VCPU. That can be done in supervisor mode (CPL == 0) and/or user mode (CPL > 0). For example, this enables the TD OS to control CPUID as seen by drivers or by user-level code.

The guest TD may do this by writing to the VCPU-scope metadata fields CPUID_SUPERVISOR_VE and CPUID_USER_VE using TDG.VP.WR.

For backward compatibility, the guest TD may use TDG.VP.CPUIDVE.SET, described in the [TDX Module ABI].

11.8.2.2. *NEW: Leaf/Sub-leaf Specific Control*

A finer grained control is provided per CPUID leaf and sub-leaf that is virtualized by the TDX module. The guest TD may configure the following, per VCPU:

- #VE injection instead of the normal CPUID virtualization is the guest executed CPUID in supervisor mode (CPL == 0).
- #VE injection instead of the normal CPUID virtualization is the guest executed CPUID in user mode (CPL > 0).

The guest TD may do this by writing to the VCPU-scope metadata field array CPUID_CONTROL using TDG.VP.WR.

11.8.3. NEW: CPUID configuration & Checks at Guest TD Migration

The CPUID virtualization configuration stored in TDCS is exported by TDH.EXPORT.STATE.IMMUTABLE and checked on import to the destination TD by TDH.IMPORT.STATE.IMMUTABLE to be compatible with the destination platform.

CPUID fields that are virtualized as fixed values (defined as "FIXED"), are based on some calculation (defined as "ASSIGNED") or that their value depends on the underlying CPU capabilities (defined as "ALLOWED" or "DIRECT") must retain the same value across migration.

CPUID fields that are virtualized as pass-through (defined as "NATIVE") are considered fixed once exported and are checked for compatibility on import.

11.9. Interrupt Handling and APIC Virtualization

Intel SDM, Vol. 3, 24.6.8	Controls for APIC Virtualization
Intel SDM, Vol. 3, 29	APIC Virtualization and Virtual Interrupts

11.9.1. Virtual APIC Mode

- Guest TDs must use virtualized x2APIC mode. xAPIC mode (using memory mapped APIC access) is not allowed.
- Guest TD attempts to RDMSR or WRMSR the IA32_APIC_BASE MSR cause a #VE to the guest TD. The guest TD cannot disable the APIC.

11.9.2. Virtual APIC Access by Guest TD

Intel SDM, Vol. 3, 29.5	Virtualizing MSR-Based APIC Access

Guest TDs are allowed access to a subset of the virtual APIC registers, which are virtualized by the CPU as described in [Intel SDM, Vol. 3, 29.5]. Access to other registers can cause a #VE. The guest TD is expected to use a software protocol over TDG.VP.VMCALL to request such operations from the host VMM.

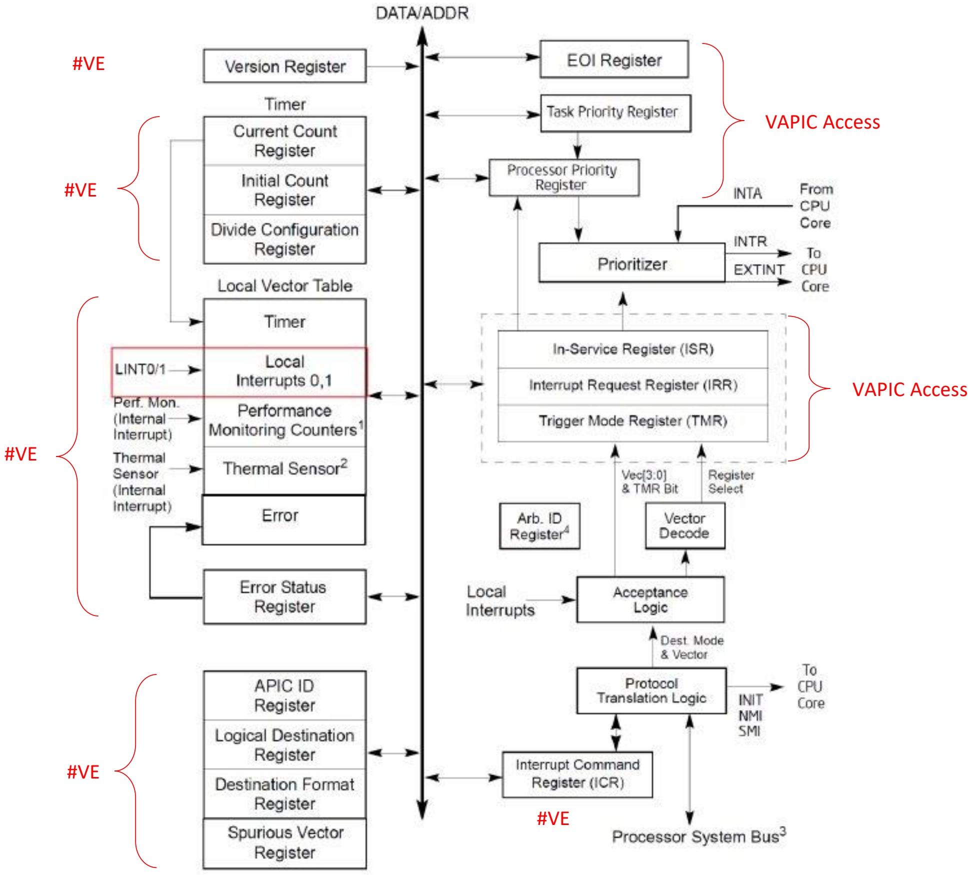


Figure 11.3: Virtual APIC Access by Guest TD

Table 11.7: x2APIC MSRs Access

MSR Range	MSR Name(s)	Description	Operation
0x802	IA32_X2APIC_APICID	APIC ID	#VE
0x803	IA32_X2APIC_VERSION	APIC version	#VE
0x80D	IA32_X2APIC_LDR	Local destination register	#VE
0x80F	IA32_X2APIC_SIVR	Spurious interrupt vector	#VE
0x828	IA32_X2APIC_ESR	Error status	#VE
0x830	IA32_X2APIC_ICR	Interrupt command	#VE
0x82F, 0x837-0x832, 0x83A	IA32_X2APIC_LVT_*	Local vector table registers	#VE
0x838, 0x839, 0x83E	IA32_X2APIC_*_COUNT, IA32_X2APIC_DCR	APIC timer registers	#VE
0x801-0x800, 0x807-0x804, 0x82E-0x829, 0x831, 0x8FF-0x840	Reserved		#GP(0)
Other MSR in the range 0x8FF-0x800			Access to VAPIC page (see [Intel SDM, Vol. 3, 29.5])

11.9.3. UPDATED: Posted Interrupts

Intel SDM, Vol. 3, 29.6 Posted-Interrupt Processing

Non-NMI interrupt injection into the guest TD by the host VMM or the IOMMU can be done through the posted-interrupt mechanism. If there are pending interrupts in the posted-interrupt descriptor (PID), the VMM can post a self IPI with the notify vector prior to TD entry.

- The posted-interrupt descriptor (PID) resides in a shared page, directly accessible by the host VMM. The VMM must set the TD VMCS's "posted-interrupt descriptor address" control (using the TDH.VP.WR function) to the PA and shared HKID of the posted-interrupt descriptor.
- The host VMM must set the TD VMCS's "posted-interrupt notification vector" control using the TDH.VP.WR function.
- To post pending interrupts in the PID, the host VMM can generate a self IPI with the notification vector prior to TD entry.

When a posted-interrupt notification vector is recognized in TDX non-root mode, the CPU processes the posted-interrupt descriptor as described in the [Intel SDM].

If needed, the guest TD may use a software protocol over TDCALL(TDG.VP.VMCALL) to ask the VMM to stop interrupt delivery through the PID.

The TD VMCS posted interrupt execution controls are reset to their initial values when the TD is migrated. The host VMM on the destination platform must set them in order to use posted interrupts.

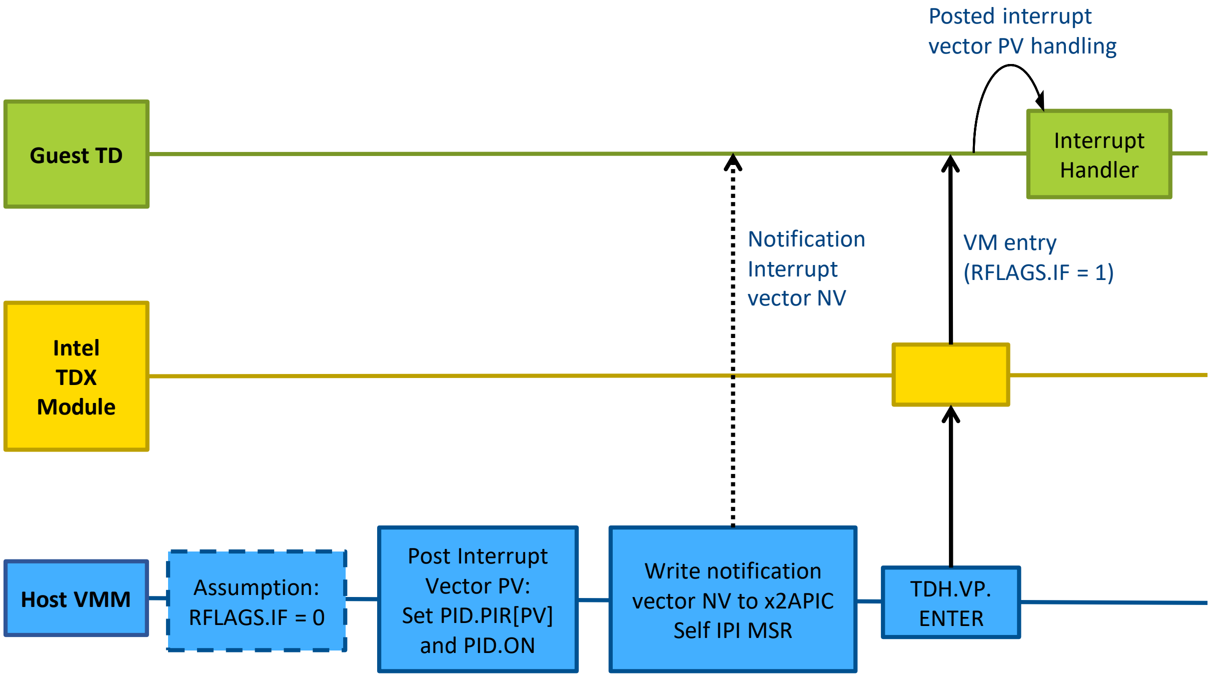

Posted interrupt vector PV handling
Guest TD
Interrupt Handler
Notification Interrupt vector NV
VM entry (RFLAGS.IF = 1)
Intel TDX Module
Host VMM
Assumption: RFLAGS.IF = 0
Post Interrupt Vector PV: Set PID.PIR[PV] and PID.ON
Write notification vector NV to x2APIC Self IPI MSR
TDH.VP. ENTER

Figure 11.4: Typical Sequence for Posted Interrupt Injection to the Current LP

11.9.4. Pending Virtual Interrupt Delivery Indication

The host VMM can detect whether there is a pending virtual interrupt delivery to a VCPU, using TDH.VP.RD to read the VCPU_STATE_DETAILS TDVPS field.

The typical use case is when the guest TD VCPU indicates to the host VMM, using TDG.VP.VMCALL, that it has no work to do and can be halted. The guest TD is expected to pass an “interrupt blocked” flag. The guest TD is expected to set this flag to 0 if and only if RFLAGS.IF is 1 or the TDCALL instruction that invokes TDG.VP.VMCALL immediately follows an STI instruction. If the “interrupt blocked” flag is 0, the host VMM can determine whether to re-schedule the guest TD VCPU based on VCPU_STATE_DETAILS.

For further details, see the TDVPS definition in the [TDX Module ABI].

11.9.5. Cross-TD-VCPU IPI

To perform a cross-VCPU IPI, the guest TD ILP should request an operation from the host VMM using TDG.VP.VMCALL. The VMM can then inject an interrupt into the guest TD’s RLPs using the posted interrupt mechanism, as described in 11.9.3 above. This is an untrusted operation; thus, the TD needs to track its completion.

11.9.6. Virtual NMI Injection

The host VMM can request the Intel TDX module to inject an NMI into a guest TD VCPU using the TDH.VP.WR function, by setting the PEND_NMI TDVPS field to 1. This can be done only when the VCPU is not active (a VCPU can be associated with at most one LP). Following that, the host VMM can call TDH.VP.ENTER to run the VCPU; the Intel TDX module will attempt to inject the NMI as soon as possible.

The host VMM can use TDH.VP.RD to read PEND_NMI and get the status of NMI injection. A value of 0 indicates that NMI has been injected into the guest TD VCPU. The host VMM also may choose to clear PEND_NMI before it is injected.

11.10. Virtualization Exception (#VE)

Intel SDM, Vol. 3, 24.9.4	Information for VM Exits Due to Instruction Execution
Intel SDM, Vol. 3, 25.5.6	Virtualization Exceptions
Intel SDM, Vol. 3, 27.2.5	Information for VM Exits Due to Instruction Execution

The Intel TDX module extends the VMX architectural usage of #VE to para-virtualize memory address translation. It injects #VE into the guest TD in multiple cases where an operation is not allowed by TDX, but an architectural exception is not applicable (e.g., #GP(0)). Such cases include disallowed instruction executions, disallowed MSR accesses, many CPUID leaves, etc.

The intended usage is for the TDX-enlightened guest TD OS to have a #VE handler. By analyzing the #VE information, the handler would be able to virtualize the requested operation for non-enlightened parts of the guest TD – e.g. drivers and applications.

11.10.1. Virtualization Exception Information

The **virtualization-exception information area** (VE_INFO) is maintained as part of TDVPS. It is not intended to be directly accessible to the guest TD. Instead, the information can be retrieved using the **TDG.VP.VEINFO.GET** function (see the [TDX Module ABI]). This is a simple way to help assure the availability and privacy of this area.

Table 11.8: Virtualization Exception Information Area (VE_INFO), based on [Intel SDM, Vol. 3, Table 24-1]

Section	Field	Offset (Bytes)	Size (Bytes)	Description
Architectural	**EXIT_REASON**	0	4	The value that would have been saved into the VMCS as an exit reason if a VM exit had occurred instead of the virtualization exception.
	VALID	4	4	0 indicates that VE_INFO has no valid contents. The CPU and the Intel TDX module will not update VE_INFO if VALID is not 0. After updating VE_INFO, the CPU and the Intel TDX module write 0xFFFFFFFF to the VALID field.
	EXIT_ QUALIFICATION	8	8	The value that would have been saved into the VMCS as an exit qualification if a VM exit had occurred instead of the virtualization exception.
	GLA	16	8	The value that would have been saved into the VMCS as a guest-linear address if a VM exit had occurred instead of the virtualization exception.
	GPA	24	8	The value that would have been saved into the VMCS as a guest-physical address if a VM exit had occurred instead of the virtualization exception.
	EPTP_INDEX	32	2	The current value of the EPTP index VM-execution control
Non-Architectural	**INSTRUCTION_ LENGTH**	Non-arch.	4	The 32-bit value that would have been saved into the VMCS as VM-exit instruction length if a legacy VM exit had occurred instead of the virtualization exception.
	INSTRUCTION_ INFORMATION	Non-arch.	4	The 32-bit value that would have been saved into the VMCS as VM-exit instruction information if a legacy VM exit had occurred instead of the virtualization exception.

The architectural section format for VE_INFO is as defined in the [Intel SDM], and it is used directly by the CPU when it injects a #VE (see 11.10.2 below). VE_INFO can also be used for #VE injected by the Intel TDX module. Some VE_INFO fields are applicable only for some exit reasons.

VE_INFO.VALID is initialized to 0, and it is set to 0xFFFFFFFF when a #VE is injected to the guest TD. When handling a #VE, the guest TD retrieves the #VE information using the **TDG.VP.VEINFO.GET** function (see the [TDX Module ABI]). TDG.VP.VEINFO.GET checks that VE_INFO.VALID is 0xFFFFFFFF. After reading the information, it sets VE_INFO.VALID to 0.

11.10.2. #VE Injection by the CPU due to EPT Violations

#VE is enabled unconditionally for TDX non-root operation. The Intel TDX module sets the TD VMCS **EPT-violation #VE** VM-execution control to 1.

For shared memory accesses (i.e., when GPA.SHARED == 1), as with legacy VMX, the VMM can choose which pages are eligible for #VE mutation based on the value of the Shared EPTE bit 63.

For private memory accesses (GPA.SHARED == 0), an EPT Violation causes a TD Exit in most cases, except when the Secure EPT entry state is PENDING (an exception to this is described in 11.11.1 below). The Intel TDX module sets the Secure EPT entry's Suppress VE bit (63) to 0 if the entry's state is PENDING. It sets that bit to 1 for all other entry states.

11.10.3. #VE Injected by the Intel TDX Module

#VE may be injected by the Intel TDX module in several cases:

- Emulation of the architectural #VE injection on EPT violation, done by a guest-side Intel TDX module flow that performs an EPT walk.
- As a result of guest TD execution of a disallowed instruction (see 11.4 above), a disallowed MSR access (see 11.7 above), or CPUID virtualization (see 11.8 above).
- A notification to the guest TD about anomalous behavior (e.g., too many EPT violations reported on the same TD VCPU instruction without making progress). This kind of #VE is raised only if the guest TD enabled the specific notification (using TDG.VM.WR to write the TDCS.NOTIFY_ENABLES field) and when a #VE can be injected. See 17.3 for details.

If, when attempting to inject a #VE, the Intel TDX module discovers that the guest TD has not yet retrieved the information for a previous #VE (i.e., VE_INFO.VALID is not 0), the TDX module injects a #DF into the guest TD to indicate a #VE overrun.

11.11. Secure and Shared Extended Page Tables (EPTs)

EPT is enabled in TDX non-root mode. TDX non-root mode uses two EPTs: Secure EPT, and Shared EPT.

EPT level is the same for both Secure and Shared EPT. If the guest TD's GPA width is greater than 48 bits (TDCS.GPAW is 1), then 5-level EPT trees are used. Otherwise, 4-level EPT trees can be used.

For further Secure EPT details, refer to Chapter 9.

EPT violations and misconfigurations generally cause a TD Exit, except for the cases described below.

11.11.1. GPAW-Relate EPT Violations

GPA bits higher than the SHARED bit are considered reserved and must be 0. Address translation with any of the reserved bits set to 1 cause a #PF with PFEC (Page Fault Error Code) RSVD bit set.

11.11.2. EPT Violation Mutated into #VE

An EPT violation is converted into #VE in the following cases:

- For Secure EPT, if the EPT entry state is PENDING.
- For Shared EPT, if the EPT entry has been configured by host VMM deliver EPT violations to the guest TD as #VE exceptions for usages such as MMIO, as described in 11.10 above.

11.12. Prevention of TD-Induced Denial of Service

VMs, including TDs, can exploit Intel ISA characteristics to cause performance and functional Denial of Service (DOS) to the VMM. The Intel architecture has several mechanisms that help prevent such DOS cases. This section describes how those mechanisms are used in the context of TDX.

11.12.1. Bus Lock Detection by the TD OS

The guest TD OS can enable debug exception traps due to bus locks by setting IA32_DEBUGCTL.BUS_LOCK_DETECT bit (2), which is disabled by default. When enabled, the feature works identically to how it functions in legacy VMX non-root mode or in non-VMX mode. The IA32_DEBUGCTL MSR and DR6 are part of the state that is saved and restored on VM exit and VM entry, respectively. If the delivery of #DB was pre-empted by a trap-like VM exit, then the pending debug exceptions (including due to BUS_LOCK_DETECT if pending) are saved in TD VMCS and restored on subsequent VM Entry. For fault-like VM Exit due to conditions such as EPT violation and EPT misconfiguration that are encountered during execution of an instruction, there is no pending debug exception recorded, including the bus lock debug exception.

11.12.2. Impact of MSR_TEST_CTRL (MSR 0x33)

The host VMM can set bits in MSR_TEST_CTRL (MSR 0x33) to cause exceptions in VMs (including TDs) in case of bus locks:

- Bit 28 (UC_LOCK_DISABLE): If set to 1, a UC load lock will trigger a #GP(0) fault.
- Bit 29 (SPLIT_LOCK_DISABLE): If set to 1, a split lock will trigger an #AC fault.

MSR 0x33 is not virtualizable; it is a core-scope MSR and may be modified by the host VMM on one SMT thread while another SMT thread is running a TD VCPU. The TDX module does not allow a guest TD to access this MSR (a #VE is generated).

To avoid any security issues, **a correctly written TD OS should always be ready to handle #AC and #GP(0) faults** if the TD software might cause UC locks or split locks.

11.12.3. Bus Lock TD Exit

Bus lock TD exit is disabled by default. The host VMM can enable the TD VMCS "bus-lock detection" VM execution control using the TDH.VP.WR function. If enabled, the processor generates a bus lock VM exit (exit reason 74) following execution of an instruction (or an iteration of a REP prefixed instruction) if the processor detects that one or more bus locks were caused by the instruction that was executed. The Intel TDX module then completes a TD exit.

- If an instruction (or an iteration of a REP prefixed string instruction) that successfully caused a bus lock subsequently faults, then a bus lock VM exit (exit reason 74) occurs on the instruction boundary following delivery of the fault (including any nested faults).
- If delivery of bus lock VM exit was pre-empted by a higher priority VM exit (e.g., EPT Misconfiguration, EPT Violation, etc.), then a "bus lock detected" notification bit (bit 26) is set in the exit reason to indicate that one or more bus locks were successfully acquired prior to this VM Exit. The Intel TDX module reflects this to the host VMM on TD exit.
- If the delivery of bus lock VM exit was pre-empted by an EPT Violation and that EPT Violation was mutated into a #VE, then the bus lock VM exit is pending at the EOM of the delivery of the #VE.

11.12.4. Notification TD Exit

Notification TD exit is disabled by default. The host VMM can write the TD VCMS "notify window" and "notification exiting" execution controls using the TDH.VP.WR function. If enabled and configured, then if the processor detects a no-commit case, the processor causes a notification VM exit (exit reason 75) which the Intel TDX module converts to the TD exit.

The conditions that cause a notification TD exit are the same as those in legacy VMX non-root mode. An example of such a case is the nested #AC exception. If an #AC exception occurs during the delivery of a previous #AC exception, then the CPU may get into an endless loop of #AC without responding to external events.

Bit 0 (VM context invalid) of the exit qualification indicates whether the guest TD context is corrupted and not valid in the TD VMCS. If this bit is set to 1, then it is a non-recoverable situation; thus, the Intel TDX module marks the TD as disabled to help prevent further TD entry. If no TD context corruption occurred (exit qualification bit 0 is cleared to 0), then the TD may be resumed normally.

11.13. Time Stamp Counter (TSC)

Intel SDM, Vol. 3, 10.5.4.1	TSC-Deadline Mode
Intel SDM, Vol. 3, 24.6.5	Time-Stamp Counter Offset and Multiplier
Intel SDM, Vol. 3, 25.3	Changes to Instruction Behavior in VMX Non-Root Operation

11.13.1. TSC Virtualization

For virtual time stamp counter (TSC) values read by guest TDs, the Intel TDX module is designed to achieve the following:

- Virtual TSC values are consistent among all the TD's VCPUs at the level supported by the CPU, see below.
- The virtual TSC value for any single VCPU is monotonously incrementing (except roll over from 2^{64}-1 to 0).
- The virtual TSC frequency is determined by TD configuration.

The host VMM is required to do the following:

- Set up the same IA32_TSC_ADJUST values on all LPs before initializing the Intel TDX module.

- Make sure IA32_TSC_ADJUST is not modified from its initial value before calling SEAMCALL.

The Intel TDX module checks the above as part of TDH.VP.ENTER and any other SEAMCALL leaf function that reads TSC.

The virtualized TSC is designed to have the following characteristics:

- The virtual TSC frequency is specified by the host VMM as an input to TDH.MNG.INIT in units of 25MHz – it can be between 4 and 400 (corresponding to a range of 100MHz to 10GHz).
- The virtual TSC starts counting from 0 at TDH.MNG.INIT time.
- TSC parameters are enumerated to the guest TD by CPUID(0x15).
- Guest TDs are not allowed to modify the TSC. WRMSR attempts of IA32_TIME_STAMP_COUNTER result in a #VE.
- Guest TDs are not allowed to access IA32_TSC_ADJUST because its value is meaningless to them. WRMSR or RDMSR attempts result in a #VE.
- RDTSCP is supported. This instruction returns the contents of the IA32_TSC_AUX MSR in RCX. the Intel TDX module allows the guest TD to access that MSR and context-switches it on TD entry and exit as part of the VCPU state in TDVPS.
- Guest TDs are not allowed to access IA32_TSC_DEADLINE. WRMSR or RDMSR attempts result in a #VE.

11.14. Supervisor Protection Keys (PKS)

By design, guest TD usage of Supervisor Protection Keys (PKS) is controlled by the ATTRIBUTES.PKS bit (see the [TDX Module ABI]). When PKS is supported by the CPU and ATTRIBUTES.PKS is set to 1, the following features are available to the guest TD:

- CPUID virtualization enumerates PKS availability to the guest TD.
- Guest TDs may enable PKS by setting CR4.PKS flag.
- Guest TDs may access the PKS state using the IA32_PKRS MSR.

11.15. Intel® Total Memory Encryption (Intel® TME) and Multi-Key Total Memory Encryption (MKTME)

Guest TDs may not directly use the Intel TME and MKTME MSRs and the PCONFIG instruction. The Intel TDX module supports para-virtualization of this ISA, as described below.

11.15.1. TME Virtualization

TME is enumerated by CPUID(0x7, 0x0).ECX[13]. The host VMM can configure the virtualization of this bit as enabled or disabled on TDH.MNG.INIT. If enabled, then a guest TD access to the IA32_TME_* MSRs (0x981 – 0x984) causes a #VE, allowing the guest TD's #VE handler to emulate the desired operation. Else, guest TD access to those MSRs causes a #GP(0).

11.15.2. MKTME Virtualization

MKTME is enumerated by CPUID(0x7, 0x0).EDX[18]. The host VMM can configure the virtualization of this bit as enabled or disabled on TDH.MNG.INIT. If enabled, then the following operations cause a #VE (e.g., the guest TD #VE handler can then communicate with the host VMM over TDG.VP.VMCALL to request the desired operation):

- Guest TD access to the IA32_MKTME_PARTITIONING MSR (0x87)
- PCONFIG execution by the guest TD

If the host VMM configured CPUID(0x7, 0x0).EDX[18] virtualized value as 0, then:

- Guest TD access to the IA32_MKTME_PARTITIONING MSR (0x87) causes a #GP(0).
- PCONFIG execution by the guest TD causes a #UD.

11.16. Other Changes in TDX Non-Root Mode

11.16.1. Tasking

Any task switch results in a VM exit to the Intel TDX module (this is a fixed-1 exit) which then performs a TD exit to the host VMM.

The VMM is expected not to reenter the TD VCPU since this case is non-recoverable; the instruction that caused the task switch (CALL, JMP or IRET) will re-execute and cause another VM exit. If the task switch was incidental to an exception delivery, then the VM entry following TDH.VP.ENTER will reattempt the delivery and cause another task switch VM exit. The expected response from the VMM is to terminate this TD.

11.16.2. PAUSE-Loop Exiting

Intel SDM, Vol. 3, 25.1.3 Instructions That Cause VM Exits Conditionally

The host VMM can only set the guest TD's "PAUSE-loop exiting" VM-execution control if the guest TD runs in debug mode (ATTRIBUTES.DEBUG is 1).

"PAUSE-loop exiting" allows the VMM to request an exit if the guest (in ring 0) executes PAUSE in a loop (e.g., busy-wait). This is intended to help avoid cases where a guest thread loops, waiting for another thread that is not currently scheduled by the VMM. However, modern enlightened guests use a VMM-provided service (hypercall) instead of PAUSE loops – this is the expected usage for Intel TDX.

12.Measurement and Attestation

12.1. TD Measurement

TDs have two types of measurement registers:

- **MRTD** helps provide static measurement of the TD build process and the initial contents of the TD.
- **RTMR** is an array of general-purpose measurement registers made available to the TD software to enable measuring additional logic and data loaded into the TD at run-time.

All TD measurements are reflected in TD attestations.

12.1.1. MRTD: Build-Time Measurement Register

The Intel TDX module measures the TD during the build process. The measurement register TDCS.MRTD is a SHA384 digest of the build process, designed as follows:

- TDH.MNG.INIT begins the process by initializing the digest.
- TDH.MEM.PAGE.ADD adds a TD private page to the TD and inserts its properties (GPA) into the MRTD digest calculation.
- Control structure pages (TDR, TDCX and TDVPR) and Secure EPT pages are not measured.
- For pages whose data contribute to the TD, that data should be included in the TD measurement via TDH.MR.EXTEND. TDH.MR.EXTEND inserts the data contained in those pages and its GPA, in 256-byte chunks, into the digest calculation. If a page will be wiped and initialized by TD code, the loader may opt not to measure the initial contents of the page with TDH.MR.EXTEND.
- The measurement is then completed by TDH.MR.FINALIZE. Once completed, further TDH.MEM.PAGE.ADDs or TDEXTENDs will fail.

MRTD extension by GPA uses a 128B buffer which includes the GPA and the leaf function name for uniqueness.

12.1.2. RTMR: Run-Time Measurement Registers

The RTMR array is initialized to zero on build, and it can be extended at run-time by the guest TD using the TDCALL(TDG.MR.RTMR.EXTEND) leaf. The syntax of the RTMR registers is designed to be similar to that of TPM PCRs, where a register’s value after TDG.MR.RTMR.EXTEND(index=i, value=x) is:

```
RTMR[i] = SHA384(RTMR[i] || x);
```

Four RTMR registers are provided.

Typical expected usage is for TPM emulation during guest TD OS secure boot by the VBIOS.

12.2. UPDATED: TD Measurement Reporting

TD attestation is initiated from inside the TD by calling TDG.MR.REPORT and specifying a REPORTDATA value. TDG.MR.REPORT creates a TDREPORT_STRUCT structure containing the TD measurements, initial configuration of the TD that was locked at finalization (TDH.MR.FINALIZE), the Intel TDX module measurements, and the REPORTDATA value. TDREPORT_STRUCT structure and TDG.MR.REPORT are detailed in the [TDX Module ABI].

TDREPORT_STRUCT is HMAC’ed using an HMAC key that is designed to be accessible only to the CPU. This helps protect the integrity of the structure and, by design, can only be verified on the local platform via the SGX ENCLU(EVERIFYREPORT2) instruction. By design, TDREPORT_STRUCT cannot be verified off platform; it first must be converted into signed Quotes, as described in 12.3 below.

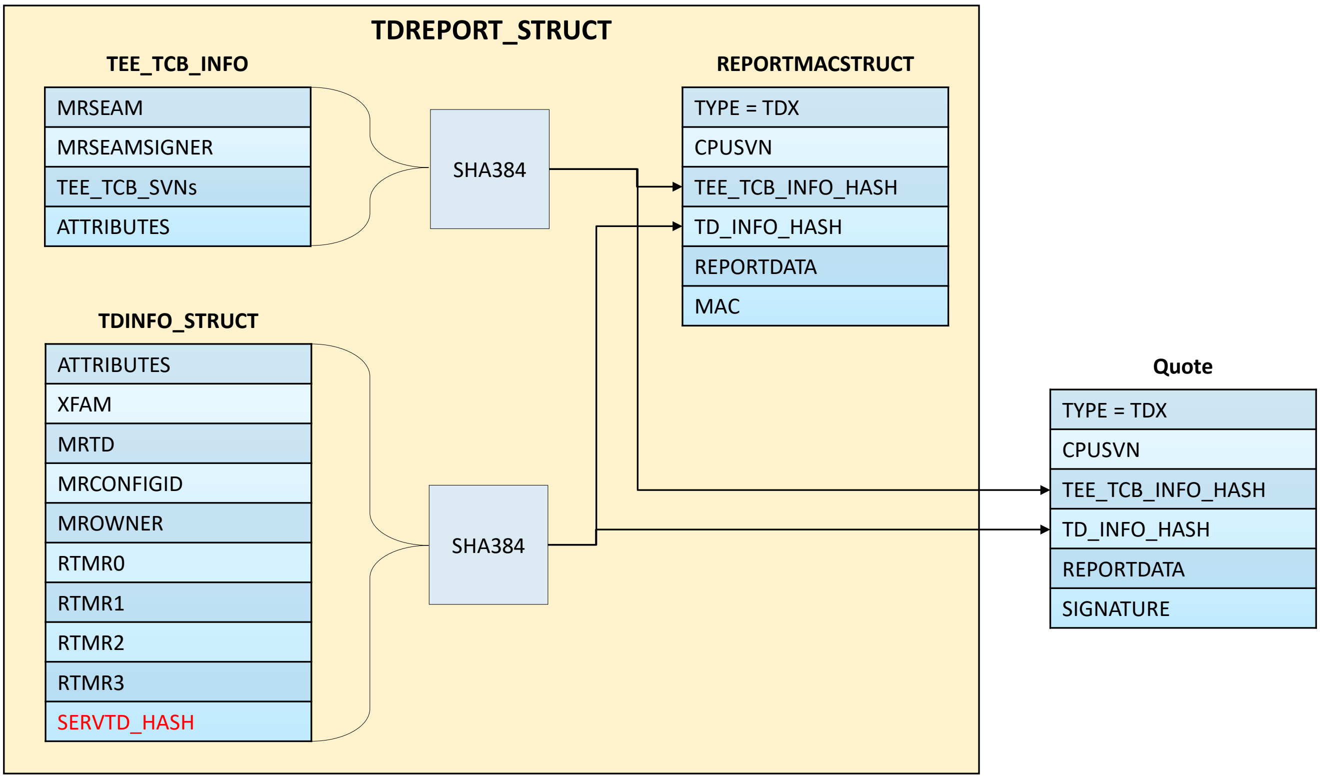


Figure 12.1: UPDATED: TD Measurement Reporting

12.3. TD Measurement Quoting

To create a remotely verifiable attestation, the TDREPORT_STRUCT should be converted into a Quote signed by a certified Quote signing key.

12.3.1. Intel SGX-Based Attestation

The Intel SGX attestation architecture is designed to provide facilities for multiple Quoting Enclaves from multiple providers. This is intended to allow the host to instantiate a Quoting Enclave for Intel SGX attestations and another Quoting Enclave for TD attestation without interference — i.e., each provider can supply its own quoting enclave, and the quoting enclave for Intel SGX and for Intel TDX may be separate; the design does not require the quoting enclave to run inside the TD.

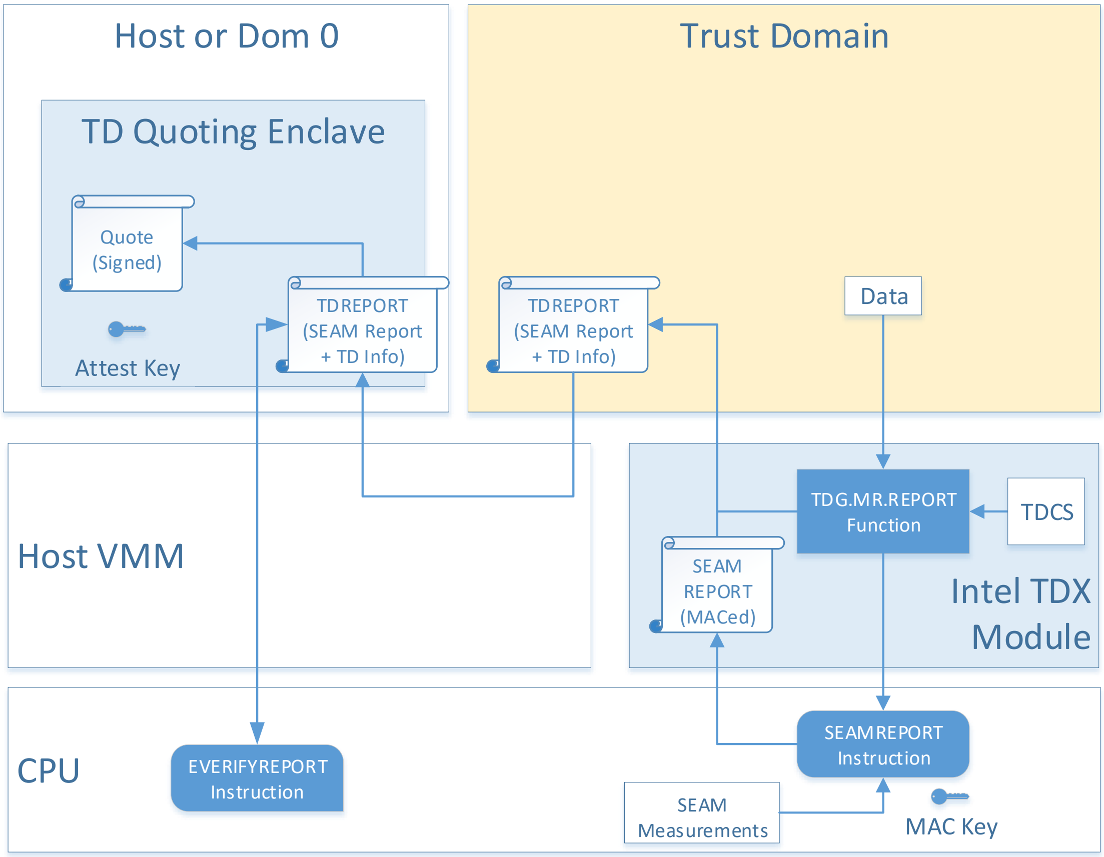


Figure 12.2: High-Level View of the Intel SGX-Based TD Attestation

Quote generation using a quoting enclave is typically performed as follows:

1. Guest TD invokes the TDCALL(TDG.MR.REPORT) function.
2. The Intel TDX module uses the SEAMREPORT instruction to create MAC'ed TDREPORT_STRUCT with the Intel TDX module measurements from CPU and TD measurements from TDCS.
3. Guest TD uses TDCALL(TDG.VP.VMCALL) to request that TDREPORT_STRUCT be converted into Quote.
4. The TD Quoting enclave uses EVERIFYREPORT2 to check the TDREPORT_STRUCT. This allows the Quoting Enclave to check the report without requiring direct access to the CPU's HMAC key. Once the integrity of the TDREPORT_STRUCT has been verified, the TD Quoting Enclave signs the TDREPORT_STRUCT body with an ECDSA 384 signing key.

12.4. Quote Signing Key

The Intel SGX infrastructure provides primitives and a certificate infrastructure to allow Quoting Enclaves to certify their own Quoting Keys. The Intel SGX Provisioning Certification Enclave (PCE) uses an Intel-Certified ECDSA-256 signing key to issue certificates to Quoting Enclaves for their attestation keys. Intel offers a service to allow third parties to download these certificates.

Typically, on first launch, the TD Quoting Enclave generates a random ECDSA 384-bit quoting key. It then contacts the Provisioning Certification Enclave which uses its signing key to sign the new quoting key's public key.

Note that the TD Quoting Enclave uses an ECDSA 384 bit key, while the PCE certifies it with an ECDSA-256 key. This is due to limitations of the SPR platform.

12.5. TCB Recovery

The Intel TDX architecture has several levels of TCB:

- CPU HW level, which includes microcode patch, ACMs and PFAT
- Intel TDX module software
- Attestation Enclaves which include the TD Quoting Enclave and Provisioning Certification Enclave

The TCB Recovery story is different for each level. The existing SGX TCB Recovery model for CPU level items applies in the same way with TDX and SGX. The model requires a restart of the platform to take effect. The Intel TDX module can be unloaded and reloaded to reflect an upgraded Intel TDX module. The enclaves can be upgraded at run-time, but if the PCE is upgraded, the design requires a new certificate to be downloaded.

13.New: Service TDs

13.1. Overview

One or more **service TDs** may be bound to a **target TD**. Service TD binding relationship has the following characteristics:

- A service TD has a **type** (SERVTD_TYPE).
- A service TD may **read and/or write certain target TD metadata**. Access permission to target TD metadata fields depends on SERVTD_TYPE.
- **Unsolicited service TD binding** is done without target TD approval. The target TD needs not be aware of the binding.
- The target TD's TDREPORT indicates binding to service TDs.
- The service TD protocol consists of:
    - Binding
    - Metadata access
- Service TD to target TD binding relationship is many-to-many
    - Multiple service TDs of different types may be bound to a single target TD.
    - Multiple target TDs may be bound to a single service TD.
- A service TD may itself be a target TD to other service TDs.

Typical Unsolicited Service TD Binding and Metadata Access Use Case

1. **Optional Pre-Binding:** During target TD build, before calling TDH.MR.FINALIZE, the host VMM calls TDH.SERVTD.PREBIND to write the binding fields (SERVTD_HASH etc.) in the target TD's service TD table.
2. **Binding:** Sometime later, the host VMM calls TDH.SERVTD.BIND to bind the service TD. It gets back a binding handle. The VMM communicates the binding handle, target TD_UUID and other binding parameters to the service TD.
3. **Metadata Access:** The service TD uses TDG.SERVTD.RD/WR* to access target TD metadata.
4. **Rebinding:** May be required due to, e.g., both target TD and service TD have been migrated or a new service TD instance replaces the original one. The host VMM calls TDH.SERVTD.BIND to rebind the service TD. It gets back a binding handle. The VMM communicates the binding handle, target TD_UUID and other binding parameters to the service TD.

13.2. Service TD Binding

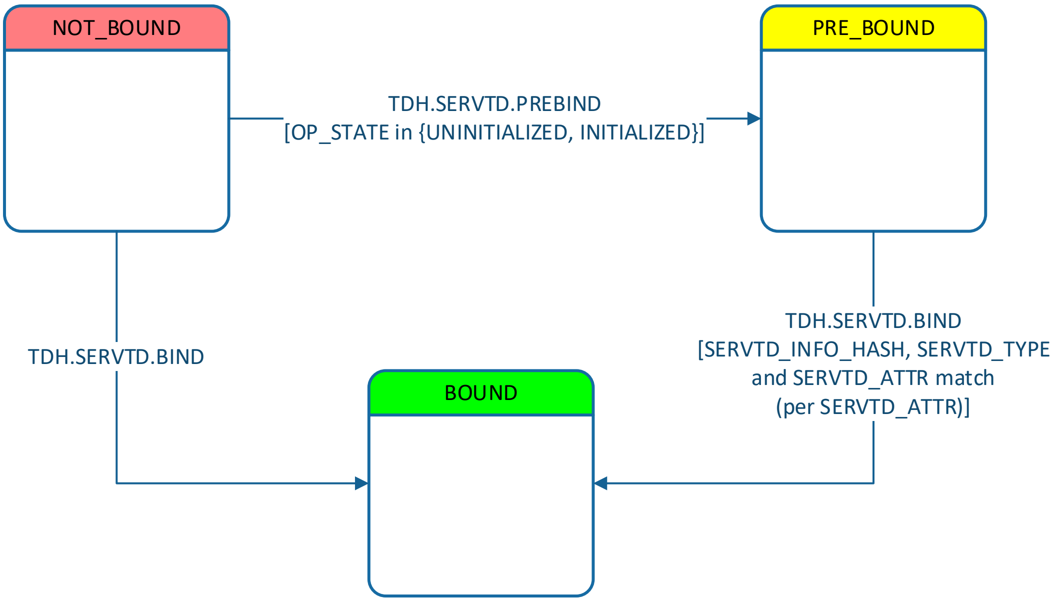


Figure 13.1: Service TD Binding State Machine

13.2.1. Service TD Binding Table in the Target TD's TDCS

The target TD's TDCS holds a service TD binding table. Each row (binding slot) in the table contains the following fields, which are detailed in the following sections:

- SERVTD_BINDING_STATE

- SERVTD_INFO_HASH
- SERVTD_TYPE
- SERVTD_ATTR
- SERVTD_UUID

The available number of slots in the table is enumerated by TDH.SYS.RD*.

13.2.2. SERVTD_BINDING_STATE: Service TD Binding State

SERVTD_BINDING_STATE indicates the state of the service TD binding slot. It has the following values:

Table 13.1: SERVTD_BINDING_STATE Definition

Value	Name	Meaning
0	NOT_BOUND	No service TD is bound. The binding fields in this slot are N/A.
1	PRE_BOUND	No service TD is bound. SERVTD_INFO_HASH, SERVTD_TYPE and SERVTD_ATTR have been set. They will be included in SERVTD_HASH calculation and be checked on any following binding.
2	BOUND	A service TD is bound. SERVTD_UUID, SERVTD_INFO_HASH, SERVTD_TYPE and SERVTD_ATTR have been set and be checked on any following binding. SERVTD_INFO_HASH, SERVTD_TYPE will be included in SERVTD_HASH calculation and be checked on any following binding.

13.2.3. SERVTD_TYPE: Service TD Binding Type

A service TD implements one or more SERVTD_TYPEs. A specific SERVTD_TYPE is specified per binding; the same service TD may be bound multiple times if it implements more than one SERVTD_TYPE.

SERVTD_TYPE controls the following:

- The target TD metadata fields that the service TD may read and/or write.
- Whether or not multiple bindings of this SERVTD_TYPE can exist at the same time for a specific target TD.

SERVTD_TYPE values supported by the TDX module are defined in the [TDX Module ABI].

13.2.4. SERVTD_ATTR: Service TD Binding Attributes

SERVTD_ATTR is a set of service TD binding attributes. It includes the following fields:

13.2.4.1. *INSTANCE_BINDING: Class vs. Instance Binding*

Specifies whether a specific Service TD instance or a class of a service TD is bound.

With **class binding**, rebinding can be done with any TD with the same SERVTD_INFO_HASH (unless PLATFORM_BINDING is 1), SERVTD_TYPE and SERVTD_ATTR as the original binding. Those parameters are migrated when the target TD is migrated.

With **instance binding**, rebinding can be done with the same TD instance (same SERVTD_UUID), using the same SERVTD_TYPE and SERVTD_ATTR as the original binding. SERVTD_INFO_HASH is not checked; instead, it is updated. This allows the service TD instance to be migrated together with the target TD; the service TD may itself bind with a different Migration TD at the destination.

13.2.4.2. *MIGRATABLE_BINDING: Binding Migratability*

Specifies whether a service TD binding can be migrated.

With **non-migratable binding**, SERVTD_INFO_HASH is not migrated, but other binding attributes (SERVTD_BINDING_STATE, SERVTD_TYPE, SERVTD_ATTR) are migrated together with the target TD's mutable state. A service TD with the same SERVTD_TYPE and SERVTD_ATTR may be bound at the destination platform. If binding on the service platform happens before import begins, and the imported SERVTD_BINDING_STATE is not NOT_BOUND, then the imported SERVTD_TYPE and SERVTD_ATTR are checked to be the same as the existing values. This is the case used for **Migration TD**.

13.2.4.3. *IGNORE_TDINFO: TDINFO Component Filtering*

A bit array determines which component of the service TD's TDINFO_STRUCT field is included in the calculation of SERVTD_INFO_HASH. For details see 13.2.6 below.

13.2.5. SERVTD_UUID: Service TD Instance Identifier

TD_UUID is a 256-bit random number that serves as a universally unique identifier of a TD. TD_UUID is created by TDH.MNG.CREATE and is stored in the TD's TDR. When a service TD is bound to a target TD, its TD_UUID is stored in the target TD's service TD table slot's SERVTD_UUID field.

13.2.6. Service TD's Binding SERVTD_INFO_HASH Calculation

For the purpose of service TD binding, a SHA384 hash of the service TD's measurable attribute is calculated in a similar way to the calculation done by TDG.MR.REPORT (see 12.2), except that filtering is applies based on the binding SERVTD_ATTR:

- The SERVTD_ATTR.IGNORE_TDINFO selects which TDINFO_STRUCT field is ignored (a value of 0 is used in the calculation).

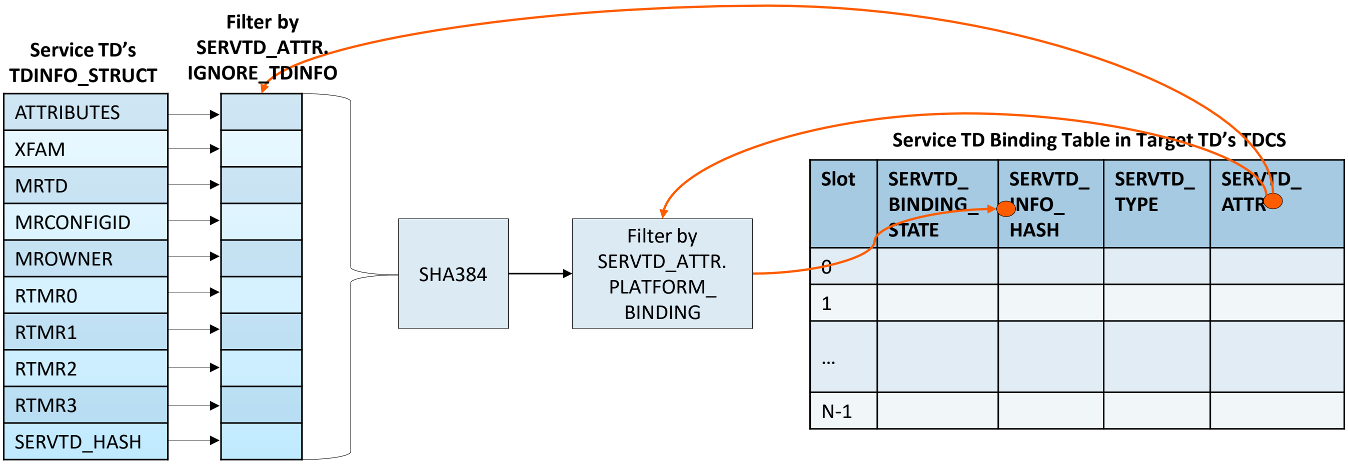

Service TD's TDINFO_STRUCT
Filter by SERVTD_ATTR. IGNORE_TDINFO
ATTRIBUTES
XFAM
MRTD
MRCONFIGID
MROWNER
RTMR0
RTMR1
RTMR2
RTMR3
SERVTD_HASH
SHA384
Filter by SERVTD_ATTR. PLATFORM_ BINDING
Service TD Binding Table in Target TD's TDCS
Slot
SERVTD_ BINDING_ STATE
SERVTD_ INFO_ HASH
SERVTD_ TYPE
SERVTD_ ATTR
0
1
…
N-1

Figure 13.2: SERVTD_INFO_HASH Calculation

13.2.7. Target TD's SERVTD_HASH Calculation

SERVTD_HASH is a single field that summarizes all the service TDs bound or pre-bound to the target TD in an unsolicited mode. On TD build, SERVTD_HASH is calculated at TDH.MR.FINALIZE time. At that time, the binding information for all bound or pre-bound service TDs is known. On import, SERVTD_HASH is re-calculated at TDH.IMPORT.STATE.IMMUTABLE time, because it may change due to non-migratable service TD binding (at least the Migration TD).

Calculation is done as follows:

1. Get all service TD binding slots whose SERVTD_BINDING_STATE is not NOT_BOUND.
2. Sort by SERVTD_TYPE as the primary key, SERVTD_INFO_HASH as a secondary key (if multiple service TDs of the same type are bound).
3. Concatenate SERVTD_INFO_HASH, SERVTD_TYPE and SERVTD_ATTR of each slot, and concatenate all slots.
4. Calculate SHA384.

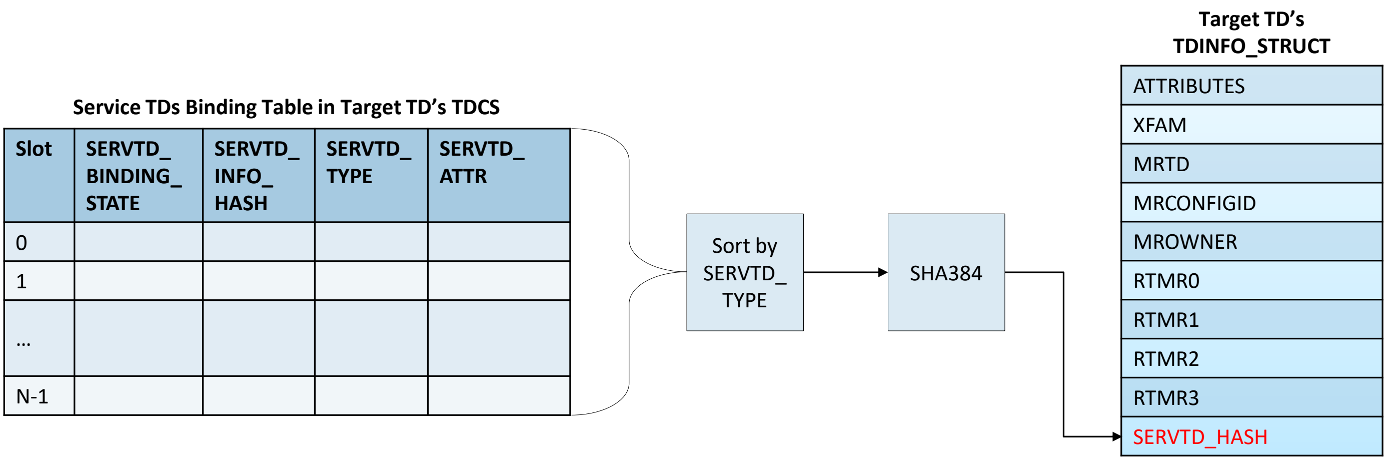


Figure 13.3: SERVTD_HASH Calculation

13.2.8. TDH.SERVTD.PREBIND: Pre-Binding a Service TD

TDH.SERVTD.BIND is used by the host VMM to bind a service TD. It is detailed in the [TDX Module ABI].

Inputs

- Target TD's TDR HPA
- SERVTD_INFO_HASH
- SERVTD_TYPE
- SERVTD_ATTR
- Service TD Index (slot number in the target TD's binding table)

Operation

- Check that the target TD's measurements have not been finalized (by TDH.MR.FINALIZE).
- Check that no service TD is already bound in the given slot number.
- Store the service TD's SERVTD_INFO_HASH, SERVTD_TYPE and SERVTD_ATTR.

13.2.9. TDH.SERVTD.BIND: Binding a Service TD

TDH.SERVTD.BIND is used by the host VMM to bind a service TD. It is detailed in the [TDX Module ABI].

Binding Scenarios

Initial Binding: No pre-binding has been done; initial service TD binding can only be done before TDH.MR.FINALIZE of the target TD.

Late Initial Binding: Pre-binding has been done; initial service TD binding can be done at any time. SERVTD_INFO_HASH and SERVTD_ATTR must match.

Rebinding: Binding has been done; rebinding conditions depend on SERVTD_ATTR as described before.

Inputs

- Target TD's TDR HPA
- Service TD's TDR HPA – NULL_PA (-1) if pre-binding is requested
- SERVTD_TYPE
- SERVTD_ATTR
- Service TD Index (slot number in the target TD's binding table)

Outputs

- Binding Handle (described below)

Operation

- Calculate the service TD's SERVTD_INFO_HASH.
- Check binding conditions vs. the target TD's binding table.

- Store the service TD's SERVTD_INFO_HASH, SERVTD_TYPE, SERVTD_ATTR and SERVTD_UUID in the target TD's binding table.
- Calculate the binding handle as f(service TD's TD_UUID, target TD's TDR HPA, slot number).

13.2.10. Binding Handle

The binding handle is used as a shortcut, to quickly identify both the target TD and the binding slot. It should be noted that the target TD identity is verified by its TD_UUID; the binding handle does not replace it. The binding handle is not a secret.

The binding handle is calculated from the following variables, using a simple addition:

- Least significant 64 bits of SERVTD_UUID – this serves to obfuscate the handle, so the service TD does not use HPA or slot number directly.
- Target TD's TDR HPA (platform-specific unique identifier of the target TD)
- Target TD's binding slot number

Given the handle, the TDX module can reconstruct TDR_HPA and binding slot number.

The binding handle is platform-specific and must be recreated after migration. This may be triggered when the service TD attempts to access target TD metadata using TDG.SERVTD.RD/WR* and an error is returned.

13.3. Target TD Metadata Access by a Service TD

13.3.1. TDG.SERVTD.RD/WR*: Metadata Read/Write Interface Functions

TDG.SERVTD.RD, TDG.SERVTD.WR, TDG.SERVTD.RDM, TDG.SERVTD.WRM are similar to other metadata access functions, e.g.:

- Host-side: TDH.MNG.RD/WR
- Guest-side: TDG.VM.RD/WR

Refer to 18.4 for a description of the TDX module metadata interface.

Inputs

- Target TD_UUID, uniquely identifying the target TD
- Binding handle, identifies the binding slot and a shortcut for identifying the target TD
- A single metadata field ID or metadata field list

Output

- For a single field access: Field value

Operation

1. Calculate the target TD's TDR HPA and binding slot number from the binding handle.
2. Check that the target TD_UUID is the same as specified.
    2.1. A special case (used by Migration TDs) is when the binding had been done on destination platform before the TD was imported. In this case the target TD_UUID is overwritten at the beginning of import, as part of the TD's immutable state import by TDH.IMPORT.STATE.IMMUTABLE. The pre-import TD_UUID is saved in the target TD's TDCS. If the specified target TD_UUID doesn't match the actual value, but matches the pre-import value, a status code is returned to the service TD, with the updated TD_UUID.
3. Get the binding parameters from the target TD's service TD table binding slot.
4. Check that the service TD's TD_UUID is equal to the target TD's bind slot's SERVTD_UUID.
5. Access the metadata (similar to other metadata access operations).

13.3.2. Metadata Access Error Handling

TDG.SERVTD.RD/WR* interface functions run in the context of the service TD, but access the target TD's control structures. This introduces an opportunity for the service TD to create a denial-of-service to the host VMM, which is handled as described below.

Local Errors (in the Service TD Context)

Local errors that only impact the service TD context are normally, as in other TDCALL flows. These include, e.g., the following cases:

- Errors such as incorrect service TD state result in an error code returned to the caller service TD.
- EPT violations when accessing the service TD's memory cause a fault-like TD exit ; The VMM may resolve the situation (e.g., TDH.EXPORT.UNBLOCKW if the service TD is being live-migrated) and resume the service TD.

Cross-TD Errors:

Cross-TD errors impact the target TD. For example, errors may happen due to target TD state, e.g., the target TD may be migrated or may be torn down. The service TD may not be aware of the target TD state when invoking the interface function.

Cross-TD errors cause a trap-like TD exit:

1. TDG.SERVTD.RD/WR* flow sets output operands (e.g., completion status returned in RAX) and advances the virtual CPU state to the next service TD guest instruction, but TD-exits immediately before resuming the guest TD.
2. The host VMM may take action to detect denial of service, e.g., the guest calling TDG.SERVTD.RD/WR* in a tight loop.
3. The host VMM may let the service TD resolve the situation by resuming it, using TDH.VP.ENTER. On TD entry, the service TD gets the status code as returned by TDG.SERVTD.RD*/WR*.

13.3.3. Cross-TD Concurrency Handling: Maintaining Host-Side Priority

13.3.3.1. *Problem Description*

Host VMM access to the target TD have a higher priority than service TD access to that target TD. This helps mitigate denial-of-service cases such as when the service TD loops on TDG.SERV.RD/WR*, locking target TD resources and preventing the host VMM from doing host-side operations that require access to such resources.

Applicable target TD resources are, e.g.:

- TDG.SERV.RD/WR* locks the target TD's TDR in a shared mode, to help assure that the target TD is available throughout the guest-side flow. This may interfere with critical host-side operations (e.g., disabling a TD) that require locking that target TD's TDR in an exclusive mode.
- TDG.SERV.RD/WR* locks the target TD's TDCS.OP_STATE to help assure that OP_STATE doesn't change in a way that prevents access during the guest-side flow. This may interfere with critical host-side operations (e.g., pausing a TD during export) that require locking that target TD's OP_STATE in an exclusive mode.

We currently assume that guest-side flows can only acquire locks in shared mode; thus, they only compete with the host-side flows acquiring locks in exclusive mode.

13.3.3.2. *Solution*

A new HOST_PRIORITY flag is added to shared/exclusive locks protecting resources that may be accessed by the host VMM and a guest service TD. For details, see 18.1.4.

14. I/O Support

This chapter specifies the Intel TDX I/O model.

14.1. Overview

Intel TDX architecture does not prescribe a specific software convention to perform I/O from the guest TD. Guest TD providers have many choices to provide I/O to the guest. The common I/O models are emulated devices, para-virtualized devices, SRIOV devices and Direct Device assignments. Guest TD providers can choose to offer the combinations of I/O models based on the workload and use case. To virtualize MMIO, the following options can be utilized:

- **Para-Virtualized Drivers** can replace MMIO accesses with TDG.VP.VMCALL to invoke VMM provided MMIO emulation functions.
- **MMIO Emulation by #VE Handlers** can use non-para-virtualized drivers in the guest TD, with the emulation performed by the #VE handler. EPT and #VE mechanisms can be used to reflect violations to the #VE handler in the guest TD on access to virtual MMIO ranges. These violations can invoke VMM-provided MMIO emulation functions through TDG.VP.VMCALL. In this model, the #VE handler is expected to emulate the faulting instruction in the guest TD.

14.2. Paravirtualized I/O

Para-virtualization (e.g., using virtio APIs in KVM, etc.) helps provide a mechanism for the guest TD to use devices on the host machine that are owned and managed by the VMM. The guest TD drivers can use the TDG.VP.VMCALL function to invoke the functions provided by the VMM to perform I/O. The TD drivers must ensure that the data buffers passed to/from functions invoked using TDG.VP.VMCALL are placed in the TD's shared memory space.

14.3. MMIO Emulation and Emulated Devices

An alternate technique that the guest TD may employ to invoke VMM functions for I/O is to emulate MMIO access from legacy device drivers. To support this use model, the VMM may enable reflection of EPT violation to emulated MMIO guest physical addresses as virtualization exceptions (#VE), as described in 11.10. A #VE exception handler in the guest TD OS can emulate the instruction causing the #VE, and as part of the emulation, it can invoke the I/O functions provided by the VMM using TDCALL(TDG.VP.VMCALL). Similar to the paravirtualized I/O model, the TD software must ensure that the data buffers passed to/from functions invoked using TDG.VP.VMCALL are placed in the TD's shared memory space.

14.4. Direct Device Assignment (DDA) and SRIOV

The VMM may assign devices directly to the guest TD. The addresses mapping the MMIO resources of such devices must be mapped in the shared memory space of the TD. When submitting data buffers to these devices, the guest TD must locate the data buffers in shared memory such that the directly assigned device can move data in/out of such buffers using DMA. The data buffers placed in shared memory should be programmed in IOMMU page tables.

The SRIOV virtual function devices assigned to guest TD also follow the DDA guidelines stated above with respect to MMIO and data buffers. The control plane of the virtual function would use the soft or hard mechanism to configure the virtual functions:

- The soft mechanism would use para-virtualization to configure the virtual function.
- The hard mechanism would use hardware mailboxes accessed using MMIO in the shared memory region.

14.5. IOMMU – DMA Remapping

The IOMMU uses the VT-d remapping tables to translate GPA in the DMA from device to an HPA. The VT-d remapping tables will reflect the mapping of memory used by I/O devices in the guest TD. The programming of the VT-d remapping tables and management will be done by the VMM.

Only shared GPA memory should be mapped in the VT-d tables:

- If the result of the translation results in a physical address with a TD private key ID, then the IOMMU will abort the transaction and report a VT-d DMA remapping failure.

- If the GPA in the transaction that is input to the IOMMU is private (SHARED bit is 0), then the IOMMU may abort the transaction and report a VT-d DMA remapping failure, even if the translated physical address is with a non-private HKID. This is intended to support debug wherein a TD or VMM could program a bad GPA into the device.

14.6. Shared Virtual Memory (SVM)

Shared Virtual Memory enables applications to access buffers directly accessed by the devices. The VT-d tables help provide the mechanism to map application buffers using the first-level and second-level page tables to provide applications access to the same memory accessed by devices.

SVM should be avoided because VT-d tables can only map shared memory.

15.Debug and Profiling Architecture

The Intel TDX module debug architecture includes the following debug facilities:

On-TD Debug: Facilities for debugging a guest TD using software that runs inside the TD

Off-TD Debug: Facilities for debugging a guest TD, configured in debug mode, using software that runs outside the TD

15.1. On-TD Debug

Intel SDM, Vol. 3, 17	Debug, Branch Profile, TSC and Intel Resource Director Technology (Intel RDT) Features

15.1.1. Overview

On-TD debug is the normal mode used to debug guest TD software. A debug agent resides inside the guest TD, and it can interact with external entities (e.g., a debugger) via standard I/O interfaces. The Intel TDX module is designed to virtualize and isolate TD debug capabilities from the host VMM and other guest TDs or legacy VMs. On-TD debug can be used for production or debug TDs – i.e., regardless of the guest TD's ATTRIBUTES.DEBUG state.

Guest TDs are allowed to use almost all architectural debug features supported by the processor, e.g.:

- Single stepping
- Code, data and I/O breakpoints
- INT3
- Bus lock detection
- DR access detection
- TSX debug

However, the TDX architecture does not allow guest TDs to toggle IA32_DEBUGCTL uncore PMI enabling bit (13).

Guest TDs are allowed to use almost all architectural tracing features, e.g.:

- LBR (if allowed by the TD's XFAM, see 11.5)
- PT (if allowed by the TD's XFAM, see 11.5)
- BTS

However, the TDX architecture does not allow guest TDs to use BTM.

15.1.2. Generic Debug Handling

15.1.2.1. *Context Switch*

By design, the Intel TDX module context-switches all debug/tracing state that the guest TD is allowed to use.

- DR0-3, DR6 and IA32_DS_AREA MSR are context-switched in TDH.VP.ENTER and TD exit flows.
- RFLAGS, IA32_DEBUGCTL MSR and DR7 are saved and cleared on VM exits from the guest TD and restored on VM entry to the guest TD.
- Pending debug traps are natively saved on VM exits from the guest TD and reloaded on VM entries using the TD VMCS PDE field.

15.1.2.2. *IA32_DEBUGCTL MSR Virtualization*

Intel SDM, Vol. 3, 17.4.1	IA32_DEBUGCTL MSR

By design, IA32_DEBUGCTL access by the guest TD is restricted as follows:

- Guest TD attempts to set any of the architecturally-reserved bits 63:15 and 5:2 result in a #GP(0).
- Guest TD attempts to set TDX-disallowed values result in a #VE. This includes the following cases:
  - Enable Uncore PMI by setting bit 13 to 1 (see 15.4 below).
  - Enable BTM by setting bits 7:6 to 0x1 (see details in 15.1.3 below).
- Uncore PMI is virtualized as disabled; bit 13 is read as 0 (see 15.4 below).

15.1.3. Debug Feature-Specific Handling

The following table discusses how specific debug features are handled.

Table 15.1: Debug Feature-Specific Handling

Debug Feature	How the Feature is Controlled	Handling
Hardware Breakpoints	• DR7, DR0-3 and DR6	No special handling: DRs are context-switched.
General Detect	• DR7 bit 13 (GD)	No special handling: DR7 is context-switched.
TSX Debug	• DR7 bit 11 (RTM) • IA32_DEBUGCTL bit 15 (RTM)	No special handling: DR7 and IA32_DEBUGCTL are context-switched.
Single Stepping	• RFLAGS bits 18 (Trap Flag) and 16 (Resume Flag) • IA32_DEBUGCTL bit 1 (BTF)	No special handling: RFLAGS and IA32_DEBUGCTL are context-switched.
Bus-Lock Detection	• IA32_DEBUGCTL bit 2 (BUS_LOCK_DETECT)	No special handling: IA32_DEBUGCTL is context-switched.
Software Breakpoints (INT1, INT3)	None	No special handling: software breakpoints are stateless.
Branch Trace Message (BTM)	• IA32_DEBUGCTL bits 6 (TR) and 7 (BTS)	Not allowed: when a guest TD attempts to set IA32_DEBUGCTL[7:6] to 0x1, the Intel TDX module injects a #VE (see 15.1.2 above). In debug mode (ATTRIBUTES.DEBUG == 1), the host VMM is allowed to activate BTM by setting the above bits to 0x1.
Branch Trace Store (BTS)	• IA32_DEBUGCTL bits 6 (TR), 7 (BTS), 8 (BTINT), 9 (BTS_OFF_OS) and 10 (BTS_OFF_USR)	No special handling: IA32_DEBUGCTL and IA32_DS_AREA are context-switched. **Notes:** • The guest TD can configure BTS to raise PMI on buffer overflow (by setting BTINT = 1). However, since PMIs are virtualized by the host VMM, the guest TD should be ready to handle spurious, delayed and dropped PMIs. See Perfmon discussion in 15.2 below. • BTS may allow the guest TD to hang the machine if BTS record generation causes a #PF or a #GP(0), because the act of getting to the exception handler may deliver another BTS. **It is highly recommended that the host VMM enables notification TD exit**, as described in 11.12.4.
Processor Trace (PT)	• IA32_RTIT_CONTROL • Requires VMM's consent on TD initialization by setting TD_PARAMS.XFAM[8] to 1	PT state handling as part of the extended feature set state is discussed in 11.5.
Architectural Last Branch Records (LBRs)	• IA32_LBR_CONTROL • Requires VMM's consent on TD initialization by setting TD_PARAMS.XFAM[15] to 1	LBR state handling as part of the extended feature set state is discussed in 11.5.
Non-Architectural LBRs	• IA32_DEBUGCTL bit 0 (LBR)	Guest TD attempt to set IA32_DEBUGCTL[0] is ignored by the CPU.

15.2. On-TD Performance Monitoring

Intel SDM, Vol. 3, 18	Performance Monitoring

15.2.1. Overview

The host VMM controls whether a guest TD can use the performance monitoring ISA using the TD's ATTRIBUTES.PERFMON bit – part of the TD_PARAMS input to TDH.MNG.INIT (see the [TDX Module ABI]).

By design, if a guest TD is allowed to use performance monitoring:

- The guest TD enumerates native Perfmon capabilities via CPUID leaf 0x0A.
- The guest TD is allowed to use all Perfmon ISA. This includes CR4.PCE, the RDPMC instruction and the Perfmon MSRs (see 15.2.2 below).
- Perfmon state is context-switched by the Intel TDX module across TD entry and exit transitions.

Context-switching the Perfmon state has a performance impact. TD entry and exit latencies are longer than when a guest TD is not allowed to use Perfmon.

By design, if a guest TD is not allowed to use performance monitoring:

- The guest TD enumerates no Perfmon capabilities. CPUID leaf 0x0A returns all 0s.
- The guest TD is not allowed to use Perfmon ISA.
- Perfmon state is not context-switched across TD entry and exit transitions.

Regardless of Perfmon enabling, per the design:

- IA32_DS_AREA MSR is context-switched across TD entry and exit transitions.
- Counter freeze control (IA32_DEBUGCTL bit 12) is context-switched across TD entry and exit transitions.
- The uncore PMI enable bit (IA32_DEBUGCTL bit 13) is preserved during SEAM mode execution, including Intel TDX module and guest TD execution. This bit is virtualized to the guest TD as 0, and the TD is prevented from setting it. See 15.4 below for details.

See also 15.1 above.

The Intel TDX module is designed to support the following performance monitoring capabilities:

- Architectural performance monitoring version 5, described in [Intel SDM, Vol. 3, 18.2.5)
- Exactly 8 performance monitoring counters (IA32_PMC0 through IA32_PMC7)
- Exactly 4 fixed counters (IA32_FIXED_CTR0 through IA32_FIXED_CTR3)
- Some non-architectural MSRs (see 15.2.2 below)

15.2.2. Performance Monitoring MSRs

Perfmon uses the following MSRs:

Table 15.2: Performance Monitoring MSRs

MSR	Comments	Enumeration	Reference
IA32_PMCx	multiple MSRs	CPUID(0x0A).EAX[15:8] The Intel TDX module requires the CPU to support 8 counters.	
IA32_PERFEVTSELx	multiple MSRs	CPUID(0x0A).EAX[15:8]	
MSR_OFFCORE_RSPx	2 MSRs, model-specific		
IA32_FIXED_CTRx	multiple MSRs	IA32_FIXED_CTRx is supported if (x < CPUID(0x0A).EDX[4:0]) or if (CPUID(0x0A).ECX[x] == 1). The Intel TDX module requires the CPU to support counters 0 through 3.	[Intel SDM, Vol. 3, 18.2.5.2]
IA32_PERF_METRICS		IA32_PERF_CAPABILITIES[15]	

MSR	Comments	Enumeration	Reference
IA32_PERF_CAPABILITIES			
IA32_FIXED_CTR_CTRL			
IA32_PERF_GLOBAL_STATUS			
IA32_PERF_GLOBAL_CTRL			
IA32_PERF_GLOBAL_STATUS_RESET			
IA32_PERF_GLOBAL_STATUS_SET			
IA32_PERF_GLOBAL_INUSE			
IA32_PEBS_ENABLE	model-specific		
MSR_PEBS_DATA_CFG	model-specific		
MSR_PEBS_LD_LAT	model-specific		
MSR_PEBS_FRONTEND	model-specific		
IA32_A_PMCx	multiple MSRs	CPUID(0x0A).EAX[15:8], IA32_PERF_CAPABILITIES[13] The Intel TDX module requires the CPU to support 8 counters.	[Intel SDM, Vol. 3, 18.2.6]

MSR virtualization is described in 11.7.

15.2.3. Performance Monitoring Interrupts (PMIs)

By design, when a guest TD is allowed to use Perfmon, it can also configure the counters to raise PMI on overflow. When such a TD counter overflows, the physical interrupt or an NMI configured by the host VMM into the local APIC is delivered. This interrupt or NMI causes a VM exit, and it is delivered as a TD exit to the host VMM. The host VMM is then expected to inject the PMI into the guest TD, either as a virtual interrupt using the posted interrupt mechanism (see 11.9.3), or as virtual NMI using the NMI injection interface (see 11.9.5).

Since the host VMM is not trusted, the guest TD must be ready to handle spurious, delayed or dropped PMIs. Thus, it is recommended for the guest TD to use PEBS instead of PMIs in order to record TD state at counter overflows.

Uncore PMIs are discussed in 15.4 below.

15.3. Off-TD Debug

A guest TD is defined as **debuggable** if its ATTRIBUTES.DEBUG bit is 1. In this mode, the host VMM can use Intel TDX functions to read and modify TD VCPU state and TD private memory, which is not accessible when the TD is non-debuggable.

A debuggable TD is, by nature, untrusted. Since the TD's ATTRIBUTES are included in the TDREPORT_STRUCT, the TD's debuggability state is visible to any third party to which the TD attests.

A debuggable TD can't be migrated; its ATTRIBUTES.MIGRATABLE bit must be 0.

The applicable Intel TDX functions are listed in Table 15.3 below. Note that some of the functions can access non-secret guest TD state regardless of the DEBUG attribute. The lists of state information that can be read and/or written in non-DEBUG and in DEBUG modes are detailed in the referenced sections.

Table 15.3: Off-TD Debug Interface

Intel TDX Function	ATTRIBUTES.DEBUG = 0	ATTRIBUTES.DEBUG = 1
TDH.MNG.RD **TDH.MNG.WR**	N/A	Access secret and non-secret TD-scope state in TDR and TDCS.
TDH.MEM.SEPT.RD	Read Secure EPT entry	Read Secure EPT entry

Intel TDX Function	ATTRIBUTES.DEBUG = 0	ATTRIBUTES.DEBUG = 1
TDH.VP.RD **TDH.VP.WR**	Access non-secret TD VCPU state in TDVPS (including TD VMCS)	Access secret and non-secret TD VCPU state in TDVPS (including TD VMCS).
TDH.MEM.WR/ **TDH.MEM.RD**	N/A	Access TD-private memory.
TDH.PHYMEM.PAGE.RDMD	Read page metadata (PAMT information)	Read page metadata (PAMT information).

15.3.1. Modifying Debuggable TD's State, Controls and Memory

When the TD is debuggable, the off-TD debugger can:

- Read and modify TDVMCS fields that contain guest state, VM entry load controls, VM exit save controls, and VM execution controls.
- Read and modify TDVPS fields that contain additional TD VCPU's state (e.g. extended register state).
- Read and modify a per-VCPU copy of the TD's extended feature mask (XFAM), such that more extended register state would be saved to TDVPS on TD exit and restore from TDVPS on TD entry.

This may cause the next VM entry into the TD VCPU to fail due to bad guest state. It may also generate VM exits that wouldn't have happened otherwise (e.g., VM exit due to a #PF within the TD). In non-debuggable TD such VM exits are not expected, and thus treated as fatal TDX module error and lead to shutdown. In debuggable TDs, however, such VM exits are expected and cause TD exit.

Specifically, the TDX module handling of TD VM exits is *extended* as follows:

```
1. If this TD VM exit might happen on non-debuggable TDs:
    1.1. Do "standard" handling (may result a TD exit).
    1.2. If an exception is pending to be injected into the TD:
        1.2.1. If the TD is debuggable and its exception bitmap is programmed to intercept that exception:
            1.2.1.1. TD exit to the VMM, as if the exception has been raised during TD execution.
    1.3. Resume the TD (may inject an exception).
2. Else (an unexpected VM exit happened):
    2.1. If the TD is debuggable then TD exit.
    2.2. Else handle this as a fatal error.
```

In any case, the security of other guest TDs running in production mode is not impacted.

15.3.2. Preventing Guest TD Corruption of DRs

The host-side debugger may need to have full control over guest DRs to help prevent their corruption by the guest TD. To do so, the debugger can do the following:

- Use TDH.VP.WR to set the TD VMCS GUEST_DR7 field's Global Detect bit.
- Set the TD VMCS exception bitmap execution control to intercept debug exceptions.

15.4. Uncore Performance Monitoring Interrupts (Uncore PMIs)

By design, neither the Intel TDX module itself not its guest TDs are allowed to use Uncore PMIs. The state of IA32_DEBUGCTL MSR bit 13 (ENABLE_UNCORE_PMI) is preserved across SEAMCALL, SEAM root and non-root mode and SEAMRET, except for very short time periods immediately after SEAMCALL and VM exit.

16. Machine Check Handling

16.1. Machine Check Architecture Background

The **machine-check architecture (MCA)** provides a mechanism for detecting and reporting hardware (machine) errors. These include system bus errors, ECC errors, parity errors, cache errors and TLB errors. MCA consists of a set of model-specific registers (MSRs) that are used to set up machine checking, and it includes additional banks of MSRs used for recording errors that are detected. The processor signals the detection of an uncorrected machine-check error by generating a **machine-check exception (#MC)**, which is an abort class exception. The implementation of the machine-check architecture does not ordinarily permit the processor to be restarted reliably after generating a machine-check exception. However, the machine-check-exception handler can collect information about the machine-check error from the machine-check MSRs.

In native virtualized platforms, this machine-check exception handler is expected to be implemented in the Virtual Machine Monitor (VMM) or in a control-OS – both of which are considered to be outside the TCB for Trust Domains. Processors on which TDX will be supported also can report information on corrected machine-check errors and deliver a programmable interrupt for software to respond to MC errors – referred to as **corrected machine-check error interrupt (CMCI)**. Intel64 processors supporting MCA and CMCI also support an additional enhancement to support software recovery from certain **uncorrected recoverable machine check errors**.

16.2. Security Objectives for Machine Check Handling

TDX architecture aims to provide resiliency against confidentiality and integrity attacks by software and limited hardware attacks. Towards this goal, the TDX architecture helps enforce the enabling of memory integrity for all private HKIDs. The CPU memory controller computes the integrity check value (MAC) for the data (cache line) during writes, and it stores the MAC with the memory as meta-data. A 28-bit MAC is stored in the ECC bits

In addition to the MAC meta-data, the memory controller also maintains a 1-bit **TD Owner** meta-data for cache-lines belonging to pages assigned to a TD for use as TD private memory. The TD Owner bit is set for TD cache lines and is clear for non-TD cache lines, and it is also covered by ECC and included in MAC calculation.

By design, checking of memory integrity is performed during memory reads. Memory integrity errors on an integrity checking failure, which can occur due to inadvertent corruption of data or due to malicious corruption, are logged by the memory controller as **UCNA (uncorrected no-action required) UCR errors** – the cache line is poisoned, and a value of 0 is returned to the core. In addition, if the TD Owner bit was set, the memory controller marks the key itself as poisoned; any subsequent reads using that key on the same memory controller also return a poisoned and zeroed data to the code.

On a subsequent consumption (read) of the poisoned data by software, there are two possible scenarios:

Scenario 1: Core determines that the execution can continue, and it treats poison with exception semantics signaled as a **#MCE (Machine Check Exception)** or **MSMI (Machine-check System Management Interrupt)**.

Scenario 2: Core determines execution cannot continue, and it does an unbreakable shutdown (e.g., long flows).

The Intel TDX module programs the logical processors to handle both scenarios for two cases:

- When a guest TD (in SEAM non-root mode) is executing on a logical processor
- When the Intel TDX module (in SEAM root mode) is executing on a logical processor

This is described in the following sections.

Hence, the security objectives for the Intel TDX module for Machine Check handling are:

- Corruption of TD private data or Intel TDX module memory must be detectable before the decrypted corrupted data are consumed by the guest TD or the Intel TDX module.
- Host software must not be able to **repeatedly** cause machine-checks during Intel TDX module or guest TD operation.
- Host software must not be able to speculatively or non-speculatively access TD private memory to detect if a prior corruption attempt was successful in finding an integrity collision or failed and received zero-data.
- On an integrity violation machine-check, the affected guest TD and the key corresponding to its affected HKID must be unusable for normal operation of the TD – i.e., the TD may only be torn down.

16.3. Corrected Machine Check Interrupt (CMCI)

CMCI is delivered as a normal interrupt. If delivered during guest TD operation, this interrupt causes a VM exit, and Intel TDX module performs a TD exit to the host VMM. If delivered during Intel TDX module operation, this interrupt remains pending until either SEAMRET to the host VMM or until VM entry to a guest TD.

16.4. Handling #MC during Guest TD Operation

An MC HARD event results in an unbreakable shutdown.

An MC KIND event that occurs during guest TD operation (in SEAM non-root mode) cannot be delivered directly by the CPU to the guest TD (as an #MC exception) for recovery attempts. That is because the guest TD software will have no avenue to translate HPAs logged in MCA banks to GPAs needed for guest TD handling. Hence, an MC KIND event during guest TD execution is considered a fatal event for that TD, and the Intel TDX module is designed to prohibit further TD entry. The TD exits to the host VMM, indicating a TDH.VP.ENTER completion with a non-recoverable TD state. The exit reason and exit qualification are reported in GPRs. The host VMM then can analyze the #MC details and reclaim the TD memory, as described in 16.7 below.

16.5. Virtualization of Machine Check Capabilities and Controls to the Guest TD

Although the guest TD is not allowed to handle machine check event, the following virtualization is used in order to allow possible pare-virtualization behavior, e.g., future handling of MC KIND event by the TD.

- The values of CPUID(1).EDX[7] (MCE) and CPUID(1).EDX[14] (MCA), as seen by the guest TD, are 1.
- The value of CR4[6] (MCE), as seen by the guest TD, is 1. Guest TD attempt to set this bit to 0 results in a #VE.
- Guest TD accesses to MSRs 0x179 (IA32_MCG_CAP), MSRs 0x17A, 0x17B, 0x4D0 (IA32_MCG_*), MSRs 0x281 through 0x29D (IA32_MCx_CTL2) and MSRs 0x400 through 0x473 (IA32_MCx_*) result in a #VE.

16.6. Handling #MC during Intel TDX Module Operation

A machine check (kind) event that occurs during Intel TDX module operation (in SEAM root mode) forces a shutdown on a current LP. Shutdown also prevents subsequent SEAMCALL on any LP.

16.7. Reclaiming Memory after a Machine Check Event

A machine check can occur due to non-malicious cases – e.g., due to a bit flip caused by cosmic rays. Hence, it is a functional requirement to be able to reclaim memory for a guest TD that was stopped due to a machine check event.

16.4 above described how the Intel TDX module is designed to handle #MC during guest TD operation. At the same time, the platform broadcasts the MC event to other LPs. Other TDs executing on other LPs will similarly be marked as FATAL. Eventually, all LPs are executing in the host VMM's #MC handler, where the VMM can interrogate the MC status registers. The host VMM can reclaim memory assigned to TDs in a FATAL state using the normal TD teardown flow (TDH.VP.FLUSH, TDH.PHYMEM.CACHE.WB, TDH.MNG.KEY.FREEID, TDH.PHYMEM.PAGE.RECLAIM).

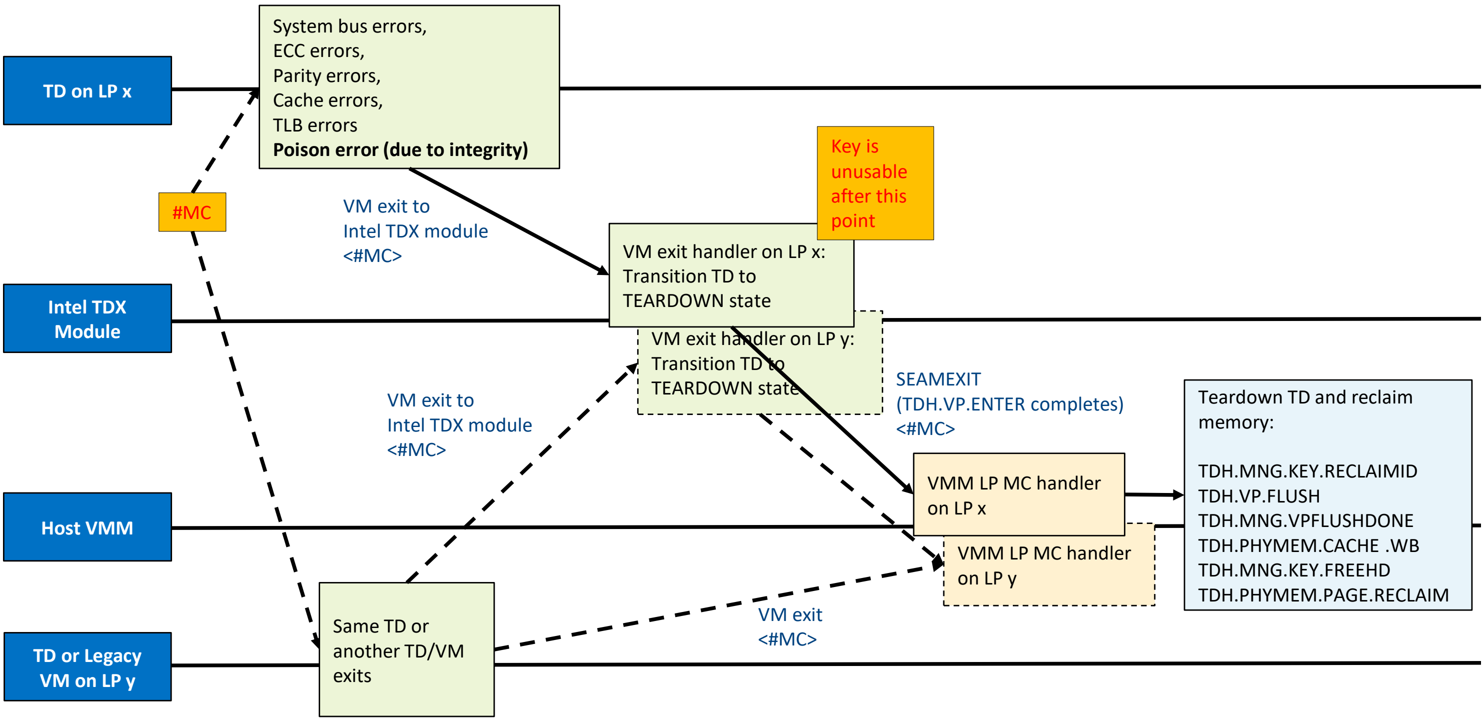

TD on LP x
Intel TDX Module
Host VMM
TD or Legacy VM on LP y
System bus errors,
ECC errors,
Parity errors,
Cache errors,
TLB errors
Poison error (due to integrity)
#MC
VM exit to Intel TDX module <#MC>
Key is unusable after this point
VM exit handler on LP x: Transition TD to TEARDOWN state
VM exit handler on LP y: Transition TD to TEARDOWN state
VM exit to Intel TDX module <#MC>
SEAMEXIT (TDH.VP.ENTER completes) <#MC>
Teardown TD and reclaim memory:
TDH.MNG.KEY.RECLAIMID
TDH.VP.FLUSH
TDH.MNG.VPFLUSHDONE
TDH.PHYMEM.CACHE .WB
TDH.MNG.KEY.FREEHD
TDH.PHYMEM.PAGE.RECLAIM
VMM LP MC handler on LP x
VMM LP MC handler on LP y
Same TD or another TD/VM exits
VM exit <#MC>

Figure 16.1: Example of Machine Check Handling and TD Memory Reclamation

17.Side Channel Attack Mitigation Mechanisms

17.1. Checking CPU Vulnerabilities to Known Attacks

On TDX module initialization (TDH.SYS.INIT and TDH.SYS.LP.INIT), the TDX module reads the IA32_ARCH_CAPABILITIES MSR to check that the following bits are set, indicating that the CPU is not vulnerable to a list of known attacks:

- Bit 0 (RDCL_NO)
- Bit 1 (IBRS_ALL)
- Bit 3 (SKIP_L1DFL_VMENTRY)
- Bit 5 (MDS_NO)
- Bit 6 (IF_PSCHANGE_MC_NO)
- Bit 8 (TAA_NO)

17.2. Branch Prediction Side Channel Attacks Mitigation Mechanisms

Branch predictions cached by the CPU before entering a guest TD should not impact the behavior of that TD. The Intel TDX module helps assure that by applying CPU mechanisms to isolate the branch predictions of each guest TD from branch predication done outside its execution.

17.3. Single-Step and Zero-Step Attacks Mitigation Mechanisms

17.3.1. Description

Single-step attacks, zero-step attacks and EPT fault attacks are techniques that provide an adversary with access to a class of powerful, low-noise side channel attacks. They do so by exploiting control over hardware such as fine resolution APIC timers, and using TDX module memory management interface functions.

- **Single-Step Attacks** involve timing pin-based events such as interrupts, NMI, SMI and INIT to interrupt the guest TD execution after every instruction executed in the guest TD. This allows the attacker to examine the state of the machine following each instruction execution in interesting regions of code, and use side channels to leak information used by that region of code.
- **EPT Fault Attacks** involve causing EPT violations or EPT misconfigurations to infer the control flow of execution inside a guest TD. Such control flow inference coupled with other side channel techniques, such as branch shadowing, can be used as a side channel to leak information from the guest TD.
- **Zero-Step Attacks** involve using an EPT fault on targeted instructions in a guest TD with an intent to glean side channel information from speculative execution past the faulting instruction. Such instructions are called “replay anchors”, as every resumption of the TD execution leads to the same EPT fault and thus the same speculative execution with the same stimulus to be replayed repeatedly, such that noise in side-channel observation of that speculative execution can be reduced.

The Intel TDX module provides mechanisms to help assist in mitigating single and zero step attacks. For single step attacks, the TDX module detects when a TD VCPU gets interrupted soon (~4K cycles) after it was entered, and continues to provide execution opportunities to the TD VCPU for a small random number of instructions before the interruption is delivered to the host VMM. For zero step attacks, the Intel TDX module counts Secure EPT faults. After a pre-determined number of such EPT violations occur on the same instruction, the TDX module starts tracking the GPAs that caused Secure EPT faults and fails further host VMM attempts to enter the TD VCPU unless previously faulting private GPAs are properly mapped in the Secure EPT.

17.3.2. Host VMM Expected Behavior

No change is required to the host VMM’s normal memory management behavior:

- The host VMM should block (TDH.MEM.RANGE.BLOCK) TD private pages and remove them (TDH.MEM.PAGE.REMOVE) only after the guest TD has explicitly relinquished the ownership of that page through a software protocol between the VMM and the TD. Such a protocol is implemented by the balloon driver mechanism employed by guest Linux kernel to allow the host VMM to overcommit a guest VM assigned memory.
- The host VMM can block TD private pages and perform the following GPA-to-HPA mapping updates without coordination with the guest TD:
  - Physical page relocation (TDH.MEM.PAGE.RELOCATE)

- Mapping merge or split (TDH.MEM.PAGE.PROMOTE, TDH.MEM.PAGE.DEMOTE)
- Unblock (TDH.MEM.RANGE.UNBLOCK)

A guest TD VCPU attempt to access such pages while they are blocked results in an EPT violation TD exit. A well-behaved host VMM should not re-enter the TD until the mapping operation is done. Failing to do so will immediately result in another EPT violation and the TD VCPU won't make any progress.

17.3.3. Guest TD Interface and Expected Guest TD Operation

The TDX module provides the guest TD with a notification facility, by which the guest TD can get notified when excessive Secure EPT violations are raised by the same TD instruction. This mechanism allows the guest TD to employ its own policies. The guest TD enables this notification by setting bit 0 of TDCS.NOTIFY_ENABLES field, using TDG.VM.WR. When this bit is set, the Intel TDX module raises #VE exception when more than a pre-determined number of Secure EPT violations are detected on the same instruction, with #VE information containing EPT violation details. This allows the guest TD to implement its advanced defenses beyond the basic defense done by the TDX module.

As part of its normal memory management behavior, the guest TD should track its GPA space allocation and should only accept (TDG.MEM.PAGE.ACCEPT) PENDING pages that it expects to be added (TDH.MEM.PAGE.AUG) by the host VMM. Failing to do so would make the TD vulnerable to attacks, e.g., the host VMM could zero-out a page by removing it and adding a new one at the same GPA.

Thus, when the guest TD attempts to access a page and a #VE is raised indicating an EPT violation, the expected guest TD's #VE handler behavior is as follows:

- If this page is not known to the guest TD as owned by it, i.e., it was not added at TD build time (TDH.MEM.PAGE.ADD) and has not been added dynamically (TDH.MEM.PAGE.AUG) and accepted (TDG.MEM.PAGE.ACCEPT), the guest TD can accept this page normally.
- Otherwise, this may indicate an attack and the guest TD can employ its own policy. For example, the guest TD may halt if this page is one of the pages expected to be resident when a security critical workload is executing, or signal the current running application so that the application would employ application-specific defenses.

The guest TD's #VE handler, as well as its virtual NMI handler, should not have any secrets that are susceptible to leakage.

The Intel TDX module does not provide protection against attacks when accessing **shared** pages. The guest TD should treat shared memory access as communicating with a potential attacker, and not do any secure processing while accessing to shared memory.

18. UPDATED: General Aspects of the Intel TDX Interface Functions

18.1. Concurrency Restrictions and Enforcement

18.1.1. Explicit Concurrency Restrictions

Intel TDX functions may specify concurrency restrictions on accessing one or more resources, as described below. In most cases, the restriction applies for the duration of the instruction execution. However, in some cases, the restriction applies for a longer duration. For example, TDH.VP.ENTER requires shared access to the TD-scope logical control structures TDR and TDCS, and it also requires shared access to TDVPS – the VCPU-scope logical control structure which applies during TDX non-root operation through TD Exit.

Table 18.1: Concurrency Restrictions of Intel TDX Functions or Flows

Concurrency Restriction	Description	Examples
Exclusive Access	During the period when an LP has an exclusive access to a certain resource, any attempt by another LP to concurrently execute an instruction that requires either an exclusive or a shared access to the same resource will fail.	• TDH.VP.CREATE requires an exclusive access to the TDVPR page.
Shared Access	During the period when an LP has a shared access to a certain resource, any attempt by another LP to concurrently execute an instruction that requires an exclusive access to the same resource will fail. No such restriction exists on another LP that attempts to concurrently execute an instruction that requires a shared access.	• TDH.VP.CREATE requires a shared access to the TDR page. • TDH.PHYMEM.CACHE.WB requires a shared access to the KOT.

Software is expected to comply with the specified concurrency restrictions. The Intel TDX module helps enforce them (using internal locks) for proper TDX operation.

Table 18.2: Concurrency Restrictions Cross-Table

		Logical Processor Y		
	Concurrency Restriction	**Exclusive**	**Shared**	**None**
Logical Processor X	**Exclusive**	Not Allowed	Not Allowed	Allowed
	Shared	Not Allowed	Allowed	Allowed
	None	Allowed	Allowed	Allowed

Intel TDX functions do not wait on a resource that requires an exclusive or a shared access. If the resource is busy, the function fails immediately.

18.1.2. Implicit Concurrency Restrictions

In some cases, Intel TDX functions and whole flows (e.g., TD Entry through TD Exit) may have **implicit** exclusive or shared access to resources. This means that the access restriction is implied by the architecture, but no direct enforcement is made by the flow itself.

An important case is TDX non-root mode. TDH.VP.ENTER acquires shared locks on the TD's TDR and TDCS control structures and on the VCPU's TDVPS control structure. These shared locks are released only on TD exit. Thus, during all the time the LP is in the logical TDX non-root mode, including during TDCALL leaf functions, the LP has implicit shared access to TDVPS, TDR and TDCS.

18.1.3. Transactions

In some cases, Intel TDX module flows update some state as a transaction. They first read the current state, then do some calculations and eventually attempt to update the state using an atomic operation (e.g., LOCK CMPXCHG) to check that the state has not changed and set its new value. If that check fails, an Intel TDX module interface function may fail with a TDX_OPERAND_BUSY status.

18.1.4. NEW: Concurrency Restrictions with Host Priority

This is a variant on explicit concurrency restrictions, where the host VMM side is given priority. A new HOST_PRIORITY flag is added to locks protecting resources that may be accessed by the host VMM and a guest TD. Both mutexes and shared/exclusive locks can be enhanced with host priority.

A new HOST_PRIORITY flag is added to shared/exclusive locks protecting resources that may be accessed by the host VMM and a guest service TD.

18.1.4.1. *Host-Side (Host VMM) Operation*

Lock operations process the HOST_PRIORITY bit as follows:

- A SEAMCALL (host-side) function that fails to acquire a lock in exclusive mode sets the lock's HOST_PRIORITY bit and returns a TDX_OPERAND_BUSY status to the host VMM. It is the host VMM's responsibility to re-attempt the SEAMCALL function until is succeeds; otherwise, the HOST_PRIORITY bit remains set, preventing the guest TD from acquiring the lock.
- A SEAMCALL (host-side) function that succeeds to acquire a lock in either exclusive or shared mode clears the lock's HOST_PRIORITY bit.

18.1.4.2. *Guest-Side (Service TD) Operation*

A TDCALL (guest-side) function that attempt to acquire a lock fails if HOST_PRIORITY is set to 1; a TDX_OPERAND_BUSY status is returned to the guest.

Currently, all applicable guest-side flows are short; once a lock is acquired, the flow releases it after a short period.

18.2. Memory and Resource Operands Access

Intel SDM, Vol. 3, 11.5.2	Precedence of Cache Controls
Intel SDM, Vol. 3, 11.11	Memory Type Range Registers (MTRRs)
Intel SDM, Vol. 3, 11.12	Page Attribute Table (PAT)

18.2.1. Overview

In this section, we discuss Intel TDX functions' memory and resource operands access from the following perspectives:

- Access semantics (shared, private, opaque and hidden)
- Explicit vs. implicit accesses
- Operand address specification (host-physical address, guest-physical address)
- Actual memory access (read or write) vs. memory reference

18.2.1.1. *Access Semantics*

Access semantics, as used in this document, convey the intended purpose of the access. Intel TDX functions are designed to use one of the following access semantics when accessing their memory and/or platform resource parameters:

Table 18.3: Access Semantics Definition

Access Semantics	Description	Intel TDX Module Usage
Shared	Memory is accessed using one of the shared HKIDs (in the range 0 to MAX_MKTME_HKIDS - 1). This is mostly used for memory parameters accessed by the VMM.	• Source page of TDH.MEM.PAGE.ADD • Memory operands of TDCALL leaf functions
Private	The memory is mapped in the TD's private GPA space. Memory accessed using the target TD's private HKID (in the range MAX_MKTME_HKIDS - 1 to MAX_HKIDS - 1). Such memory pages can be mapped in the TD's private GPA space.	• TD private pages • Secure EPT pages
Opaque	Memory is addressable by the host VMM, but its content is not directly accessible to software or devices. Memory is encrypted using either the Intel TDX global private key (for TDR) or the TD's ephemeral private key (for other control structures).	• TDR • TDCX • TDVPR
Hidden	Access is to an Intel TDX module internal resource. That resource is not directly addressable as a memory operand to software or devices.	• KOT • WBT

Note that on guest-side (TDCALL) functions, shared vs. private semantics is determined by the GPA provided as an operand to the function. A specific TDCALL leaf function may or may not impose a private or a shared access – e.g., TDG.MEM.PAGE.ACCEPT requires a private GPA, while TDG.MR.REPORT may work with either a private GPA or a shared GPA.

18.2.1.2. *Explicit vs. Implicit Access*

An **explicit memory access** is defined as an access where the memory location is provided as explicit operand to an Intel TDX function. The address may be provided directly in a GPR or indirectly via some address field in a software-accessible memory data structure.

The HKID for accessing the memory can be inferred by the instruction – implicitly or explicitly from the explicitly provided access.

An **implicit memory access** is defined as an access to a platform physical memory address, or to some other resource, that is not passed as an operand of an instruction (either directly or indirectly) but is implied by use of the Intel TDX function. TDX architecture helps guarantee that an implicit access is performed correctly, or a proper error action is taken.

18.2.1.3. *Memory Operand Address Specification*

Host-side Intel TDX functions (SEAMCALL leaf functions) memory operands are specified using their **host-physical address (HPA)**, their **guest-physical address (GPA)**, or when a GPA-to-HPA mapping is done (e.g., TDH.MEM.PAGE.ADD) by **both HPA and GPA**.

In most cases, HPA for private or opaque access semantics must specified with all HKID bits set to 0.

Guest-side Intel TDX functions (TDCALL leaf functions) memory operands are specified using their **guest-physical address (GPA)**.

18.2.1.4. *Memory Type*

18.2.1.4.1. Memory Type for Private and Opaque Accesses

The memory type for **private** and **opaque** access semantics, which use a private HKID, is WB.

18.2.1.4.2. Memory Type for Shared Accesses

Intel SDM, Vol. 3, 28.2.7.2 Memory Type Used for Translated Guest-Physical Addresses

The memory type for **shared** access semantics, which use a shared HKID, is determined as described below. Note that this is different from the way memory type is determined by the hardware during non-root mode operation. Rather, it is a best-effort approximation that is designed to still allow the host VMM some control over memory type.

- For **shared access during host-side (SEAMCALL) flows**, the memory type is determined by MTRRs.
- For **shared access during guest-side flows (VM exit from the guest TD)**, the memory type is determined by a combination of the Shared EPT and MTRRs.
  - If the memory type determined during Shared EPT walk is WB, then the effective memory type for the access is determined by MTRRs.
  - Else, the effective memory type for the access is UC.

18.2.1.5. *Actual Memory Access vs. Memory Reference*

In some cases, Intel TDX functions only **reference** memory – i.e., use its address, but no actual access is done.

In other cases, Intel TDX functions **access** the memory – i.e., perform read or write (but not execute) operations.

18.2.1.6. *Summary Table*

Table 18.4: Memory Access Summary

Explicit/ Implicit	Intel TDX Function	Access Semantics	Address Operand	HKID Derivation	Memory Type	Example
Explicit	Host-Side (SEAMCALL Leaf)	Shared	HPA	Derived HPA operand's HKID bits	From MTRR	SRCPAGE operand of TDH.MEM.PAGE.ADD
		Private	HPA	TD's HKID	WB	Target page of TDH.PHYMEM.PAGE.RECLAIM
			GPA	TD's HKID	WB	CHUNK operand of TDH.MR.EXTEND
			HPA and GPA	TD's HKID	WB	Target page of TDH.MEM.PAGE.ADD
		Opaque	HPA	TD's HKID or Intel TDX global HKID	WB	TDVPR operand of TDADDVPR
	Guest-Side (TDCALL Leaf)	Shared	GPA	From Shared EPT	From Shared EPT and MTRR	REPORTDATA operand of TDG.MR.REPORT
		Private	GPA	TD's HKID	WB	Target page of TDG.MEM.PAGE.ACCEPT
Implicit	All	Private/ Opaque	N/A	TD's HKID or Intel TDX global HKID	WB	TDCS access by TDH.VP.ENTER
		Hidden	N/A	N/A	N/A	KOT access by TDH.MNG.KEY.CONFIG

18.3. UPDATED: Register Operands and CPU State Convention

Intel SDM, Vol. 3, 24.9	VM-Exit Information Fields
Intel SDM, Vol. 3, App. C	VMX Basic Exit Reasons

18.3.1. Overview: Regular vs. Transition Leaf Functions

Intel TDX functions can be divided into transition functions and non-transition functions.

The **non-transition functions** are where SEAMCALL and TDCALL leaf functions behave as emulated CPU instructions from the perspective of the host VMM and the guest TD, respectively. In those cases, the meaning of input and output register operands is straightforward – similar to CPU instructions.

Transition cases are SEAMCALL(TDH.VP.ENTER) and TDCALL(TDG.VP.VMCALL) leaf functions, where a full cycle (until start of the next instruction) includes TD transitions to the guest TD or host VMM, respectively, and back to the host VMM or guest TD, respectively. In those cases, we look at the functions from the point of view of the caller. The meaning of input and output register operands is more complicated.

Both cases are explained in the following sections and in the function reference sections.

18.3.2. NEW: Interface Function Leaf and Version Numbers

Interface functions are selected by a leaf number, provided in RAX. A version number enables supporting multiple versions of the same function, if required for backward compatibility. Unless otherwise specified, the default version number is 0.

Table 18.5: Intel TDX Interface Functions Leaf and Version Numbers in RAX

Bits	Field	Description
15:0	Leaf Number	Selects the SEAMCALL or TDCALL interface function
23:16	Version Number	Selects the SEAMCALL or TDCALL interface function version
63:24	Reserved	Must be 0

18.3.3. UPDATED: Interface Function Completion Status

Intel TDX function completion status is returned in RAX. The status is structured to provide as many details to software about error conditions as practically possible. At the same time, the status enables software to ignore details that it does not need. Software may parse the completion status at three detail levels, as described below.

18.3.3.1. *Least Detailed Level: Success/Warning/Error*

At this simplest level, software can differentiate among three cases:

Table 18.6: Intel TDX Interface Functions Completion Status in RAX at the Least Detailed Level

RAX Value	Meaning	Description
0	Success	Function completed successfully
Positive (0x00000000_00000001 – 0x7FFFFFFF_FFFFFFFF)	Informational / Warning	Function completed successfully, but with some informational or warning code – e.g., TDH.PHYMEM.PAGE.RECLAIM of a TDCX page that is already not VALID
Negative (0x80000000_00000000 – 0xFFFFFFFF_FFFFFFFF)	Error	Function aborted due to some error

18.3.3.2. UPDATED: Medium Detailed Level: Class, Recoverability and Fatality

At this level, software can understand the following information:

Class: The class of error or warning – e.g., Resource Busy

Recoverability Hint: Whether the function can be **retried** after some conditions have been corrected – e.g., if some resource was busy due to a concurrency error, retrying the function may succeed.

Fatality Hint: Whether the TD has entered a state where it can only be torn down

18.3.3.3. UPDATED: Most Detailed Level

At this level, software can understand more details of an error that happened – e.g., if TDH.VP.ADDCX fails, software may understand if it is due to a wrong number of TDCX pages or due to the VCPU already being initialized.

Table 18.7: Intel TDX Interface Functions Completion in RAX at the Most Detailed Level

Bits	Name	Description
63	ERROR	Instruction aborted due to error. 0: Indicates that the function completed successfully – possibly with some warnings. 1: Indicates that the function aborted due to some error.
62	NON_RECOVERABLE	Recoverability hint – applicable only when ERROR is 1. 0: Indicates that the function may possibly be retried after some conditions have been corrected. 1: Indicates that the error is probably not recoverable.
61	FATAL	Fatality hint – applicable only for SEAMCALL, if ERROR is 1. 0: Indicates that the TD can continue its normal lifecycle. 1: Indicates that the TD entered a state where it can only be torn down.
60	HOST_RECOVERABILITY_HINT	As a TDH.VP.ENTER output, indicates a TDCALL that resulted in a trap-like TD exit for which the host VMM needs to provide a recoverability hint in the following TD entry. On the following TDH.VP.ENTER, the host VMM provides a hint to the guest TD, which is the output of the TDCALL: 0: The host VMM hints that the guest-side function may possibly be retried (e.g., the host may have corrected some conditions). 1: The host VMM hints that the error is probably not recoverable.
59:48	RESERVED	Reserved – set to 0
47:40	CLASS	Class of the function completion status
39:32	DETAILS_L1	Details of the function completion status
31:0	DETAILS_L2	Additional details of the function completion status – e.g., includes: • Implicit or explicit operand identifier • CPUID leaf or sub-leaf • MSR index • VMCS field code • VM exit reason • CMR index • TDMR index

Refer to the [TDX Module ABI] for a list of function completion codes.

18.3.4. CPU State Preservation Convention

18.3.4.1. *TDH.VP.ENTER*

TDH.VP.ENTER is a special case. In addition to explicit output operands discussed in 0 below, TDH.VP.ENTER is not designed to preserve the extended CPU state that the TD may use according to TDCS.XFAM.

The host VMM is expected to save any state it needs before calling TDH.VP.ENTER. Details are provided in the TDH.VP.ENTER leaf function definition (see the [TDX Module ABI]).

18.3.4.2. *Other Interface Functions*

All Intel TDX functions except TDH.VP.ENTER are designed to preserve the CPU state not explicitly defined as output.

Most interface functions preserve the AVX, AVX2 and AVX512 state. There are some exceptions, as described in the specific function definitions:

- TDG.VP.VMCALL may use some XMM registers to pass information to and from the host VMM.
- Some interface functions may reset AVX, AVX2 and AVX512 state to the architectural INIT state.

18.3.5. Transition Cases: TD Entry and Exit

18.3.5.1. *TD Entry: TDH.VP.ENTER*

Transfer of Host VMM State to TD Guest

By design, in the case of a TDH.VP.ENTER leaf function that follows a previous TDG.VP.VMCALL, the RCX input parameter of the previous TDG.VP.VMCALL is used as a bitmap. It selects the GPRs (from RBX, RDX, RBP, RDI, RSI and R8 through R15) and XMM registers whose value is transferred to the guest TD as-is. RAX is set to 0. See the TDG.VP.VMCALL description in the [TDX Module ABI].

The rest of the CPU state is restored from the TD VCPU state as saved on TDG.VP.VMCALL.

Output State (Back to the Host VMM)

On completion of TDH.VP.ENTER, a success – indicated by the ERROR bit (RAX[63]) being 0 – means that TD Entry into the TD guest was successful. The TD guest ran for some time and then exited to the Intel TDX module. That exit can be due to execution of TDG.VP.VMCALL) or due to an asynchronous exit (e.g., an EPT Violation). The Intel TDX module then executes SEAMRET, transferring control to the instruction following TDH.VP.ENTER. In this case, the DETAILS field (RAX[31:0]) format is designed to be the same as the VMX **Exit reason**.

If the completion of TDH.VP.ENTER (i.e., exit from the TD guest) was due to TDCALL(TDG.VP.VMCALL), then the RCX input parameter of TDG.VP.VMCALL is designed to be used as a bitmap. It selects the GPRs (from RBX, RDX, RBP, RDI, RSI and R8 through R15) and XMM registers whose value is passed to the host VMM as the output of TDH.VP.ENTER. RCX itself is passed as-is to the output of TDH.VP.ENTER, and RAX[31:0] indicates the **VMCALL** exit reason (see below). See the TDG.VP.VMCALL description in the [TDX Module ABI].

If the completion of TDH.VP.ENTER was due to another reason, then other VMX-like Exit Information fields are provided in other GPRs. Details are provided in the TDH.VP.ENTER leaf function definition (see the [TDX Module ABI]).

By design, any GPRs and extended states that do not return values as described above are set to synthetic values. If the VMM uses any of them, it must explicitly save them before TDH.VP.ENTER and restore them afterward.

18.3.5.2. *TD Synchronous Exit: TDG.VP.VMCALL*

Transfer of TD Guest State to Host VMM

In the case of a TDG.VP.VMCALL leaf function, the RCX input parameter of TDG.VP.VMCALL is designed to be used as a bitmap. It selects the GPRs (from RBX, RDX, RBP, RDI, RSI and R8 through R15) and XMM registers whose value is passed to the host VMM as the output of TDH.VP.ENTER. RCX itself is passed as-is to the output of TDH.VP.ENTER.

RAX provides TDH.VP.ENTER completion status (see above). All other CPU state components, including GPRs and XMM registers not selected by RCX, are saved in TDVPS and set to fixed values (see the [TDX Module ABI]). The value of RCX itself is also saved to TDVPS.

Output State (Back to the Guest TD)

On completion of TDG.VP.VMCALL, a success – indicated by the ERROR bit (RAX[63]) being 0 – means that a SEAMRET into the VMM was successful. The VMM ran for some and then executed TDH.VP.ENTER successfully (possibly on another LP). The Intel TDX module executed VMRESUME successfully, transferring control to the instruction following TDCALL.

In this case, the RCX input parameter of TDG.VP.VMCALL is designed to be used as a bitmap. It selects the GPRs (from RBX, RDX, RBP, RDI, RSI and R8 through R15) and XMM registers whose value reflects their state as input to TDH.VP.ENTER. All other CPU states, including GPRs and XMM registers not selected by RCX, are restored from TDVPS.

18.4. UPDATED: Metadata Access Interface

18.4.1. Introduction

Metadata access interface is the architecture that allows representing TDX metadata, i.e., TD non-memory state and TDX module control state, in a way that is independent of the way it is stored. It does this by hiding the memory format of TDX control structures and allowing abstraction of data, as follows:

- The actual fields stored in the TD control structures may be different than their abstracted representation. E.g., a TDVPS field may be provided as a GPA to TDH.VP.WR, while internally stored as an HPA.
- Access to a TD metadata field may **trigger some operation**. E.g., writing the TD VMCS's "posted-interrupt descriptor address" control triggers the verification of related control and may enable posted interrupt handling.
- TD metadata fields may be completely **virtual**, i.e., there may be no actual control structure fields represented by them.

Metadata abstraction is used in the following cases:

- Read of TDX Module information by the host VMM and guest TD using the following SEAMCALL and TDCALL functions:
  - Single Field Read: TDH.SYS.RD, TDG.SYS.RD
  - All Fields Read: TDH.SYS.RDALL, TDG.SYS.RDALL
- Read and write of TDR, TDCS and TDVPS control structures by the host VMM using the following SEAMCALL functions:
  - Single Field Access: TDH.MNG.RD, TDH.MNG.WR, TDH.VP.RD, TDH.VP.WR
- Read and write of TDR, TDCS and TDVPS control structures by the guest TD using the following TDCALL functions:
  - Single Field Access: TDG.VM.RD, TDG.VM.WR, TDG.VP.RD, TDG.VP.WR
- Read and write of TDR and TDCS a service TD using the following TDCALL functions:
  - Single Field Access: TDG.SERVTD.RD, TDG. SERVTD.WR
- For **TD migration**, export and import of TD metadata by the host VMM using the following SEAMCALL functions:
  - State Export: TDH.EXPORT.STATE.IMMUTABLE, TDH.EXPORT.STATE.TD, TDH.EXPORT.STATE.VP
  - State Import: TDH.IMPORT.STATE.IMMUTABLE, TDH.IMPORT.STATE.TD, TDH.IMPORT.STATE.VP

18.4.2. Metadata Fields and Elements

Metadata fields are identified by **field identifiers (MD_FIELD_ID)**. A field identifier contains a **FIELD_CODE** and other information. A detailed description and MD_FIELD_ID values are defined in tables provided in the [TDX Module ABI]. Metadata fields size may be up to 128 bytes.

For the purpose of metadata abstraction interface, fields are divided into multiple **field elements**; the size of each element can be 1, 2, 4 or 8 bytes. Elements in a field have consecutive field codes, incremented by 1 or 2 as encoded in by the field identifier's INC_SIZE.

Figure 18.1 below shows an example of a SHA384 fields (e.g., TDCS.MRCONFIGID), whose size is 48B. When access using the metadata access functions, this field is divided into six 8-byte elements.

	Element 0	Element 1	Element 2	Element 3	Element 4	Element 5
FIELD_CODE	X	X + 1	X + 2	X + 3	X + 4	X + 5
Content	Bytes 7:0	Bytes 15:8	Bytes 23:16	Bytes 31:24	Bytes 39:32	Bytes 47:40

Figure 18.1: Example of a 48 Byte TDCS.MRCONFIGID Field Composed of Six 8 Byte Elements

A detailed definition of a field identifier is provided in the [TDX Module ABI].

18.4.3. Arrays of Metadata Fields

Metadata fields can be organized in arrays. Figure 18.2 below shows an example of an array of 4 fields, each composed of 1 element. In this case, fields in the array have consecutive field codes, incremented by 1 or 2 as encoded in by the field identifier's INC_SIZE field.

Array Index	Field Code	Content
0	X + 0	Array[0]
1	X + 1	Array[1]
2	X + 2	Array[2]
3	X + 3	Array[3]

Figure 18.2: Example of an Array of 4 Single-Element Fields

Figure 18.3 below shows an example where each field is composed of multiple elements. TDCS.RTMR is such a case. The base FIELD_ID of each field in the array is incremented by the number of elements in each field, multiplied by 1 or 2 as encoded in by the field identifier's INC_SIZE field.

Array Index	Base FIELD_ID	Element 0's FIELD_ID	Element 1's FIELD_ID	Element 2's FIELD_ID	Element 3's FIELD_ID	Element 4's FIELD_ID	Element 5's FIELD_ID
0	X + 0	X + 0	X + 1	X + 2	X + 3	X + 4	X + 5
1	X + 6	X + 6	X + 7	X + 8	X + 9	X + 10	X + 11
2	X + 12	X + 12	X + 13	X + 14	X + 15	X + 16	X + 17
3	X + 18	X + 18	X + 19	X + 20	X + 21	X + 22	X + 23

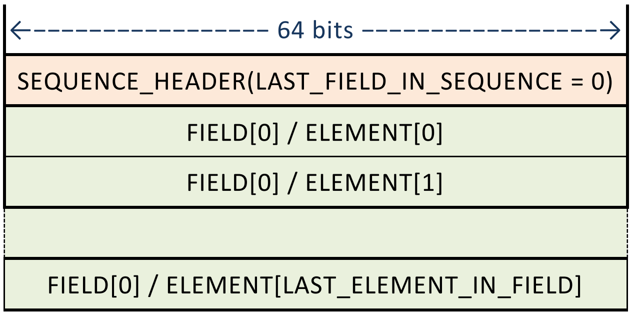
Figure 18.3: Example of an Array of Four 48 Byte TDCS.RTMR Fields, Each Composed of 6 Elements

18.4.4. NEW: Metadata Field Sequences

Field sequences contain one or more whole metadata fields, each composed of one or more elements. A sequence is composed of a sequence header and one or more values.

- All fields in a sequence have the same CONTEXT_CODE, CLASS_CODE and size.
- Each element is a sequence occupies 8 bytes, even if its size is 1, 2 or 4 bytes. When a sequence is used as an output of the TDX module, the upper bytes beyond the element size are zeroed-out. When a sequence is used as an input of the TDX module, the upper bytes are ignored.
- The FIELD_CODEs of each element in a sequence are consecutive.
- A field sequence may contain a write mask, which applies to each element value in the sequence. This is applicable when the sequence is used for writing bit fields, e.g., VMCS execution controls.
- A sequence always contains whole fields, i.e., if a field is composed of multiple elements, the sequence contains all of them.

A **field sequence header** contains the initial field code and other information – for a detailed description see the [TDX Module ABI].



64 bits

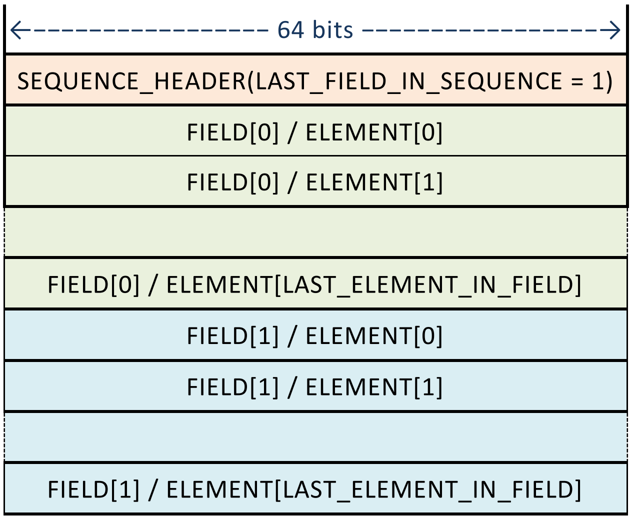
SEQUENCE_HEADER(LAST_FIELD_IN_SEQUENCE = 0)

FIELD[0] / ELEMENT[0]

FIELD[0] / ELEMENT[1]

FIELD[0] / ELEMENT[LAST_ELEMENT_IN_FIELD]

Figure 18.4: Example of a Metadata Field Sequence with One Field Composed of Multiple Elements



64 bits

SEQUENCE_HEADER(LAST_FIELD_IN_SEQUENCE = 1)

FIELD[0] / ELEMENT[0]

FIELD[0] / ELEMENT[1]

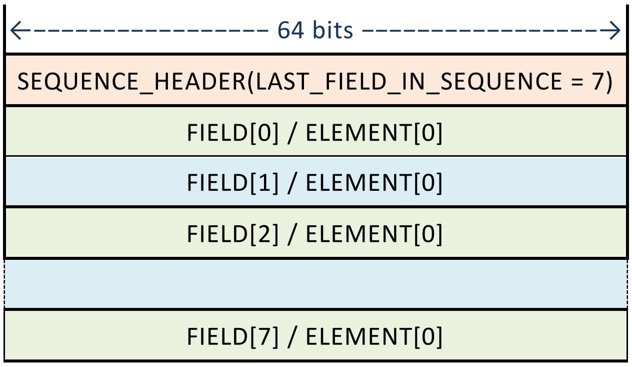
FIELD[0] / ELEMENT[LAST_ELEMENT_IN_FIELD]

FIELD[1] / ELEMENT[0]

FIELD[1] / ELEMENT[1]

FIELD[1] / ELEMENT[LAST_ELEMENT_IN_FIELD]

Figure 18.5: Example of a Metadata Field Sequence with 2 Fields Composed of Multiple Elements



64 bits

SEQUENCE_HEADER(LAST_FIELD_IN_SEQUENCE = 7)

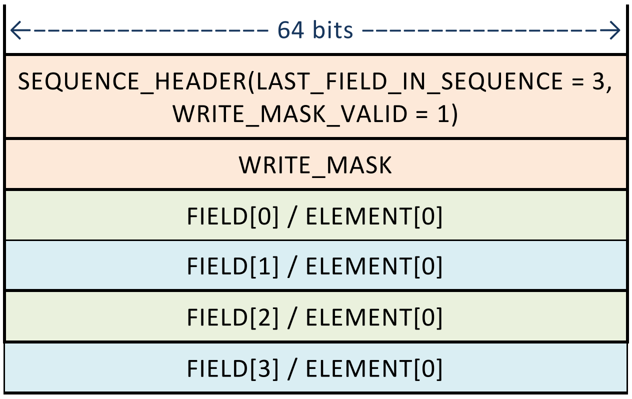
FIELD[0] / ELEMENT[0]

FIELD[1] / ELEMENT[0]

FIELD[2] / ELEMENT[0]

FIELD[7] / ELEMENT[0]

Figure 18.6: Example of a Metadata Field Sequence with 7 Fields Composed of a Single Element



64 bits

SEQUENCE_HEADER(LAST_FIELD_IN_SEQUENCE = 3, WRITE_MASK_VALID = 1)

WRITE_MASK

FIELD[0] / ELEMENT[0]

FIELD[1] / ELEMENT[0]

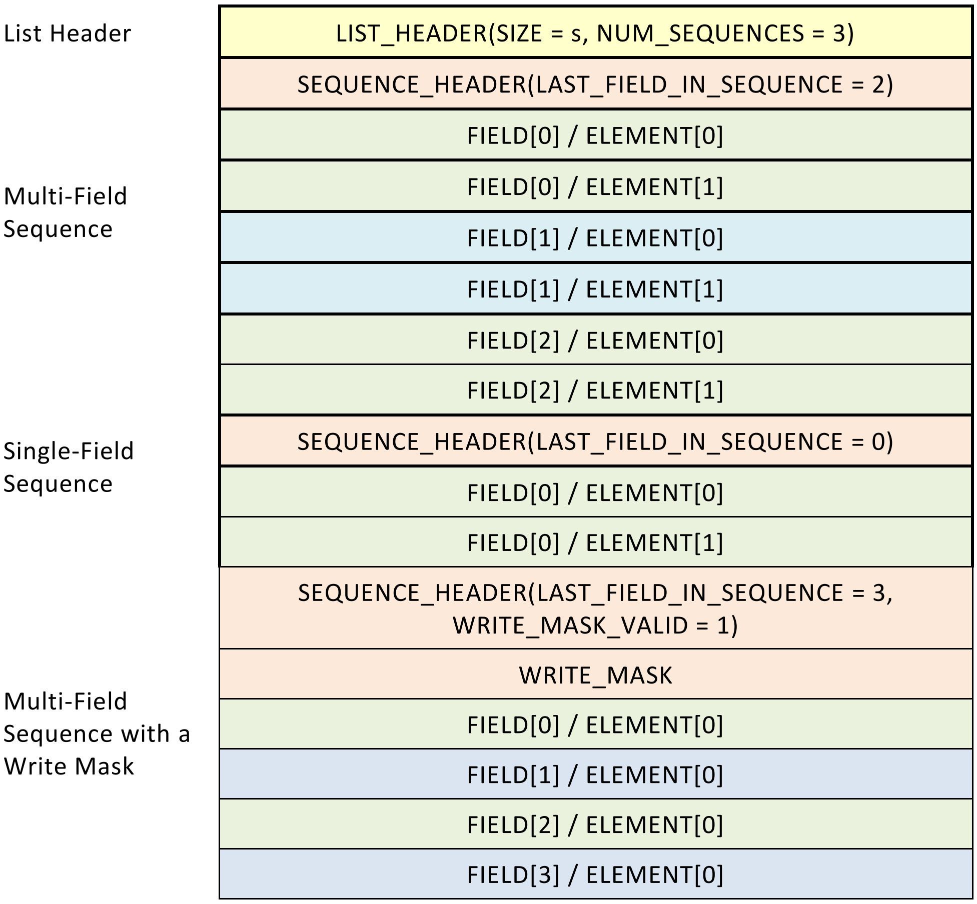
FIELD[2] / ELEMENT[0]

FIELD[3] / ELEMENT[0]

Figure 18.7: Example of a Metadata Field Sequence with a Write Mask

18.4.5. NEW: Metadata Lists

A metadata list is composed of a list header and one or more field sequences. The list header specifies list buffer size in bytes and the number of sequences. Metadata lists are used, e.g., for exporting VCPU metadata by THD.EXPORT.STATE.VP.



List Header

LIST_HEADER(SIZE = s, NUM_SEQUENCES = 3)

Multi-Field Sequence

SEQUENCE_HEADER(LAST_FIELD_IN_SEQUENCE = 2)

FIELD[0] / ELEMENT[0]

FIELD[0] / ELEMENT[1]

FIELD[1] / ELEMENT[0]

FIELD[1] / ELEMENT[1]

FIELD[2] / ELEMENT[0]

FIELD[2] / ELEMENT[1]

Single-Field Sequence

SEQUENCE_HEADER(LAST_FIELD_IN_SEQUENCE = 0)

FIELD[0] / ELEMENT[0]

FIELD[0] / ELEMENT[1]

Multi-Field Sequence with a Write Mask

SEQUENCE_HEADER(LAST_FIELD_IN_SEQUENCE = 3, WRITE_MASK_VALID = 1)

WRITE_MASK

FIELD[0] / ELEMENT[0]

FIELD[1] / ELEMENT[0]

FIELD[2] / ELEMENT[0]

FIELD[3] / ELEMENT[0]

Figure 18.8: Metadata List Example

The metadata list header format is defined in the [TDX Module ABI].

18.5. Latency of the Intel TDX Interface Functions

The Intel TDX module runs with interrupts disabled (including NMI and SMI). To support proper system responsiveness, most TDX module interface functions are designed to have a short latency. However, there are infrequent cases where the latency of some interface functions may be longer than normal, as listed below.

- Host-side interface functions that are invoked a limited number of times during TDX module lifecycle. The interface functions below are known to have longer than normal latencies:
    - TDH.SYS.INIT
    - TDH.SYS.LP.INIT
    - TDH.SYS.KEY.CONFIG
- Host-side interface functions that are invoked a limited number of times during TD . The interface functions below are known to have longer than normal latencies:
    - TDH.MNG.KEY.CONFIG
    - TDH.MNG.INIT
    - TDH.VP.INIT
- TDH.VP.ENTER may have a long latency if the single/zero step attack mitigation (described in 17.3) is activated due to a suspected attack.